CADE-21

The 21^{st} Conference on Automated Deduction

4th International Verification Workshop

VERIFY'07

Editor:

Bernhard Beckert

Bremen, Germany, July 15–16, 2007

CADE-21 Organization:

Conference Chair: Michael Kohlhase (Jacobs University Bremen)
Program Chair: Frank Pfenning (Carnegie Mellon University)
Workshop Chair: Christoph Benzmüller (University of Cambridge)
Local Organization: Event4 Event Management

Preface

The VERIFY workshop series aims at bringing together people who are interested in the development of safety and security critical systems, in formal methods, in the development of automated theorem proving techniques, and in the development of tool support. Practical experiences gained in realistic verifications are of interest to the automated theorem proving community and new theorem proving techniques should be transferred into practice. The overall objective of the VERIFY workshops is to identify open problems and to discuss possible solutions under the theme "What are the verification problems? What are the deduction techniques?".

This volume contains the research papers presented at the *4th International Verification Workshop* (VERIFY'07) held July 15–16, 2007 in Bremen, Germany. This workshop was the 4th in a series of international meetings since 2002. It was affiliated with the *21st Conference on Automated Deduction* (CADE-21).

Each paper submitted to the workshop was reviewed by three referees, and an intensive discussion on the borderline papers was held during the online meeting of the Program Committee. 13 research papers were accepted based on originality, technical soundness, presentation, and relevance. I wish to sincerely thank all the authors who submitted their work for consideration. And I would like to thank the Program Committee members and other referees for their great effort and professional work in the review and selection process. Their names are listed on the following pages.

In addition to the contributed papers, the program included three excellent keynote talks. I am grateful to Prof. Cesare Tinelli (The University of Iowa, USA), Prof. Tobias Nipkow (TU München, Germany), and Prof. Aaron Stump (Washington University in St. Louis, USA) for accepting the invitation to address the workshop.

July 2007 Bernhard Beckert

Program Chair and Organiser

Bernhard Beckert	University of Koblenz-Landau, Germany

Program Committee

Serge Autexier	DFKI & University Saarbrücken, Germany
Yves Bertot	INRIA Sophia Antipolis, France
Bruno Dutertre	SRI International, USA
Reiner Hähnle	Chalmers University, Gothenburg, Sweden
Dieter Hutter	DFKI Saarbrücken, Germany
Andrew Ireland	Heriot-Watt University, Edinburgh, UK
Deepak Kapur	University of New Mexico, USA
Joost-Pieter Katoen	RWTH Aachen, Germany
Joseph Kiniry	University Dublin, Ireland
Heiko Mantel	RWTH Aachen, Germany
Fabio Massacci	University of Trento, Italy
Stephan Merz	INRIA Lorraine, France
Till Mossakowski	University of Bremen, Germany
Lawrence C. Paulson	University of Cambridge, UK
Wolfgang Reif	University of Augsburg, Germany
Julian Richardson	Powerset Inc., USA
Luca Viganò	University of Verona, Italy
Christoph Walther	TU Darmstadt, Germany

Steering Committee

Serge Autexier	DFKI & University Saarbrücken, Germany
Heiko Mantel	RWTH Aachen, Germany

Additional Referees

Dominik Haneberg
Holger Grandy
Kurt Stenzel

Table of Contents

Invited Talks

Research Papers

Reflecting Linear Arithmetic: From Dense Linear Orders to Presburger Arithmetic

Tobias Nipkow

Institut für Informatik, Technische Universität München
http://www.in.tum.de/~nipkow

Abstract

This talk presents reflected quantifier elimination procedures for both integer and real linear arithmetic. Reflection means that the algorithms are expressed as recursive functions on recursive data types inside some logic (in our case HOL), are verified in that logic, and can then be applied to the logic itself. After a brief overview of reflection we will discuss a number of quantifier elimination algorithms for the following theories:

- Dense linear orders without endpoints. We formalize the standard DNF-based algorithm from the literature.
- Linear real arithmetic. We present both a DNF-based algorithm extending the case of dense linear orders and an optimized version of the algorithm by Ferrante and Rackoff [3].
- Presburger arithmetic. Again we show both a naive DNF-based algorithm and the DNF-avoiding one by Cooper [2].

We concentrate on the algorithms and their formulation in Isabelle/HOL, using the concept of *locales* to allow modular definitions and verification. Some of the details can be found in joint work with Amine Chaib [1].

References

1. A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. Technical report, Institut für Informatik, Technische Universität München, 2006. Submitted for publication.
2. D. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.
3. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Computing*, 4:69–76, 1975.

Lightweight Verification with Dependent Types

Aaron Stump

Computer Science and Engineering Dept.
Washington University in St. Louis

Abstract

Dependent types, studied for many years in Logic, have recently been gaining attention in Functional Programming Languages for expressing rich properties as types. A simple example is a type $\langle list\ A\ n \rangle$, for lists of length n holding objects of type A. A more complex example is $\langle trm\ G\ T \rangle$, for terms in some object language which have object-language type T in context G. Dependently typed programming languages seek to support static verification of code manipulating such data types, by statically enforcing the constraints the data types impose. The verification is lightweight in the sense that the aim is typically to verify preservation of datatype properties, rather than full functional specifications of programs.

This talk will explore dependently typed programming in the context of Guru, a new dependently typed programming language under development at Washington University in St. Louis. Guru lifts the restriction to terminating programs which is commonly required by dependently typed programming languages (such as Coq, Epigram, and ATS, to name just a few). This is done by the novel technical feature of strictly separating program terms from proofs, and types from formulas, thus going counter to the commonly used Curry-Howard isomorphism. We will consider dependently typed programming in Guru via several examples: tree operations which are statically verified to preserve the binary search tree property, and compilation of simply typed object programs which is statically verified to preserve the programs' object-language type.

Trends and Challenges in Satisfiability Modulo Theories

Cesare Tinelli*

Department of Computer Science
The University of Iowa
tinelli@cs.uiowa.edu

Abstract

Satisfiability Modulo Theories (SMT) is concerned with the problem of determining the satisfiability of first-order formulas with respect to a given logical theory T. A distinguishing feature of SMT is the use of inference methods tailored to the particular theory T. By being theory-specific and restricting their language to certain classes of formulas (such as, typically but not exclusively, ground formulas), such methods can be implemented into solvers that are more efficient in practice than general-purpose theorem provers. SMT techniques have been traditionally developed to support deductive software verification, but they have also applications in model checking, certifying compilers, automated test generation, and other formal methods.

This talk gives an overview of SMT and its applications, and highlights some long-standing challenges for a wider applications of SMT techniques within formal methods, as well as some fresh challenges introduced by new potential uses. A major challenge is providing adequate model generation features for disproving verification conditions.

* The author's research described in this talk was made possible with the partial support of grants #0237422 and #0551646 from the National Science Foundation and a grant from Intel Corporation.

Formal Device and Programming Model for a Serial Interface

Eyad Alkassar[1,⋆], Mark Hillebrand[2,⋆], Steffen Knapp[1,⋆], Rostislav Rusev[1,⋆], and Sergey Tverdyshev[1,⋆]

[1] Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
{eyad, sknapp, rusev, deru}@wjpserver.cs.uni-sb.de
[2] German Research Center for Artificial Intelligence (DFKI GmbH), Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
mah@dfki.de

Abstract. The verification of device drivers is essential for the pervasive verification of an operating system. To show the correctness of device drivers, devices have to be formally modeled. In this paper we present the formal model of the serial interface controller UART 16550A. By combining the device model with a formal model of a processor instruction set architecture we obtain an assembler-level programming model for a serial interface. As a programming and verification example we present a simple UART driver implemented in assembler and prove its correctness. All models presented in this paper have been formally specified in the Isabelle/HOL theorem prover.

1 Introduction

The Verisoft project [1] aims at the pervasive modeling, implementation, and verification of complete computer systems, from gate-level hardware to applications running on top of an operating system. The considered systems employ various devices, e.g., a hard disk controller for persistent storage, a time-triggered bus controller for communication in a distributed system, and a serial interface for user interaction via a terminal. The drivers controlling these devices are part of the operating system and proving their correctness is critical to proving the correctness of the system as a whole.

Here we consider a system which the user may control with a terminal connected via a serial interface. To prove the functional correctness of the serial interface device driver it is not sufficient to argue only about the driver code; the serial interface itself and its interaction with the processor have to be formally modeled, too. In this paper we present for the first time a formal model of a serial interface and its programming model at the assembler language level. Furthermore, as an informal example, we present a serial interface driver and sketch its correctness proof with respect to our models.

The remainder of this paper is structured as follows. In Sect. 2 we discuss previous and related work. In Sect. 3 we sketch the instruction set architecture of the VAMP

⋆ Work of the first author was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'. Work of the third author was supported by the International Max Planck Research School for Computer Science (IMPRS). Work of all but the fourth author was supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

processor [2, 3] and show how memory-mapped devices can be integrated into this architecture. In Sect. 4 we present the formal model of a UART 16550A controller and formalized environmental and software conditions. To informally demonstrate the utility of the framework, in Sect. 5 we present a simple driver written in assembler, which writes several words to the serial interface. We sketch its correctness proof.

2 Previous and Related Work

For the pervasive verification of computer systems, as done in the Verisoft project, devices must be modeled at different system layers. Some results of Verisoft's completed or ongoing work in the context of devices and their drivers have already been published. One subproject of Verisoft deals with the verification of a FlexRay-like shared serial bus interface to be used in distributed automotive systems. To verify such a system we need to argue formally about low-level clock synchronization [4], gate-level implementation, and driver real-time properties of the FlexRay interface. All these arguments will finally be combined into one formal pervasive correctness proof. A paper-and-pencil style description of this ongoing effort can be found in Knapp and Paul [5].

In another Verisoft subproject the formal pervasive verification of a general-purpose computer system is attempted. In this context, Hillebrand *et al.* [6] presented paper-and-pencil formalizations of a system with devices for the gate and the assembler level. For a hard disk as a specific device, correctness arguments justifying these models and a correctness proof for a disk driver were given on paper. Here we *formalize* large portions of [6] for a communication device.

So far almost all other device related verification approaches have either aimed at the correctness of a gate-level implementation or at showing safety properties of drivers.

In approaches of the former kind, simulation- and test based techniques are used to check for errors in the hardware designs. In particular, [7–10] deal with UARTs in that manner. Berry *et al.* [8] specified a UART model in a synchronous language and proved a set of safety properties regarding FIFO queues. From that model a hardware description can be generated (either in RTL or software simulation) and run on a FPGA.

In approaches of the latter kind the driver code is usually shown to guarantee certain API constraints of the operating system and hence cannot cause the system to crash. For example, the SLAM project [11] provides tools for the validation of safety properties of drivers written in C. Lately, the success of the SLAM project led to the deployment of the Static Driver Verifier (SDV) as part of the Windows Driver Foundation [12]. SDV automatically checks a set of 65 safety rules concerning Windows Driver API for given C driver programs. Hallgren *et al.* [13] modeled device interfaces for a simple operating system implemented in Haskell. Three basic memory-mapped I/O primitives were specified: read, write, and a test for valid region.

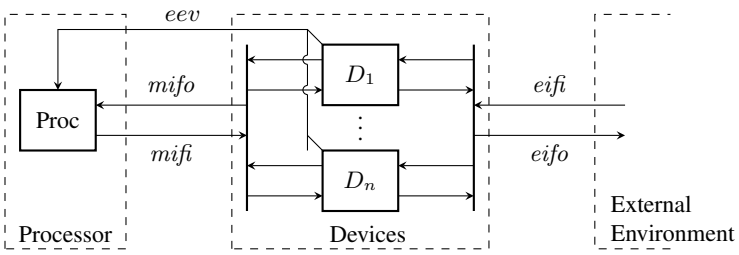


Fig. 1. Overview: A System with Processor and Devices

However, the only correctness property being stated is the disjointness of the device address spaces.

In contrary to all mentioned approaches, we aim at the formalization and *functional* verification a (UART) driver interacting with a device. Thus, it is not sufficient to argue about the device or the programming model alone. Similar in scope is the 'mini challenge' proposed by Holzmann [14], which deals with the formal verification of a file system for a Flash device. Apparently, formalizing the device and its interaction with the driver is also part of the challenge. However, no details have yet been published.

3 Processor and Devices

In this section we define the instruction set architecture (ISA) of a processor with memory-mapped devices as depicted in Fig. 1.

Compared to regular ISA definitions we have the following differences: in addition to the processor state, the state space of the combined architecture also includes devices states. Processor and devices may interact (i) by the processor issuing memory operations to special memory regions (device addresses) and (ii) by the device causing interrupts. Additionally, devices can make computational steps on their own when interacting with an external environment (e.g., a network). Therefore we model the computation of ISA with devices as an interleaved computation.

Note that at the hardware level processor and devices run in parallel and not interleaved. This requires some non-trivial extensions of the formal hardware correctness proof, which we will report on elsewhere.

3.1 Processor

A processor configuration c_{P} is a tuple consisting of (i) two program counters $c_{\mathrm{P}}.pc$ and $c_{\mathrm{P}}.dpc$ implementing delayed branching, (ii) general purpose, floating point, and special purpose register files $c_{\mathrm{P}}.gpr$, $c_{\mathrm{P}}.fpr$, $c_{\mathrm{P}}.spr$, and (iii) a byte addressable memory $c_{\mathrm{P}}.m$.

Devices are mapped into the processor memory. Thus by simple read and write operations the processor can access them. In addition devices can signal an interrupt to the processor via an external event signal (cf. Fig. 1).

Let DA denote the set of memory addresses mapping to devices, which are disjoint from regular physical memory addresses. The processor indicates an access to an address in DA via the memory interface input $mifi$ and receives the device's response on the memory interface output $mifo$; this naming convention is from the point of view of the devices.

Formally, let the predicates $lw(c_{\mathrm{P}})$ and $sw(c_{\mathrm{P}})$ indicate load and store word instructions and let $ea(c_{\mathrm{P}})$ and $RD(c_{\mathrm{P}})$ denote the address and affected processor register for such operations (see Müller and Paul [15] for full definitions).

The memory interface input has the following four components: (i) the read flag $mifi.rd = lw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a load from a device address, (ii) the write flag $mifi.wr = sw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a store to a device address, (iii) the address $mifi.a = ea(c_{\mathrm{P}})$ is set to the effective address, with $ea[14:12]$ specifying the accessed device and $ea[11:2]$ specifying the accessed device port (we support up to eight devices with up to 1024 ports of width 32 bits), and finally (iv) the data input $mifi.din = c_{\mathrm{P}}.gpr[RD(c_{\mathrm{P}})]$ is set to the store operand.

The memory interface output $mifo \in \{0,1\}^{32}$ contains the device's response for a load operation on a device.

The processor model is defined by the output function ω_{P} and the next state function δ_{P}. The former takes a processor state c_{P} and computes a memory interface input $mifi$ to the device as defined above. The latter takes a processor state c_{P}, a device output $mifo$, and an external event (interrupt) vector eev (where $eev[i]$ is set iff device D_i indicates an interrupt). It returns the next state of the processor c'_{P}.

3.2 Devices

The configurations of all devices are combined in a mapping c_{D} from an index D_i to the corresponding device configuration.

Our device model is sequential in the sense that a device may progress either due to a processor access or an input from the external environment. To distinguish both cases we extend the set of device indices by the processor index P and denote this set by PD.

The device transition function δ_{D} specifies the interaction of the devices with the processor and the external environment. It takes a processor-device index $idx \in PD$, an input from the external environment $eifi$, an input from the processor $mifi$, and a combined device configuration c_{D}. It returns a new device configuration c'_{D}, an output to the processor $mifo$, and an external output $eifo$.

Depending on the input index idx and the memory input $mifi$, the transition function δ_{D} is defined according to the following three cases:

- If $idx \neq P$, a step of the device idx is triggered by the external input $eifi$. In this case δ_{D} ignores the given $mifi$.
- If $idx = P \wedge (mifi.wr \vee mifi.rd)$, a device step is triggered by a processor device access. In this case δ_{D} ignores the given $eifi$ and produces an arbitrary $eifo$. The device being accessed as well as the access-type is specified by the given $mifi$.
- Otherwise the processor does not access any device. In this case, δ_{D} does nothing.

The device output function ω_{D} computes the external event vector eev for the processor based on the current device configurations.

3.3 Combined System

By combining the processor and device models we obtain a model for the overall system with devices as depicted in Fig. 1. This model allows interaction with an external environment via $eifi$ and $eifo$ whereas the communication between processor and devices is not visible from the outside anymore.

A configuration c_{PD} of the combined model consists of a processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$ and device configurations $c_{\mathrm{PD}}.c_{\mathrm{D}}$.

Similarly to the previous models, we define a transition function δ_{PD} and an output function ω_{PD}. Both functions take the same three inputs: a processor-device index idx, a configuration c_{PD}, and an external input $eifi$.

We introduce some more notation for the transition and the output function. Let $mifi = \omega_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}})$ be the memory interface input from the processor to the devices. Let $(c'_{\mathrm{PD}}.c_{\mathrm{D}}, mifo, eifo) = \delta_{\mathrm{D}}(idx, c_{\mathrm{PD}}.c_{\mathrm{D}}, eifi, mifi)$ denote the updated device configuration, the memory output to the processor, and the external output. Let $eev = \omega_{\mathrm{D}}(c'_{\mathrm{PD}}.c_{\mathrm{D}})$ denote the external event vector, which is computed based on the updated device configuration. Finally, if $idx = P$ then $c'_{\mathrm{PD}}.c_{\mathrm{P}}$ denotes the updated processor configuration, i.e., $c'_{\mathrm{PD}}.c_{\mathrm{P}} = \delta_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}}, eev, mifo)$. Otherwise $c'_{\mathrm{PD}}.c_{\mathrm{P}}$ denotes the unchanged processor configuration, i.e., $c'_{\mathrm{PD}}.c_{\mathrm{P}} = c_{\mathrm{PD}}.c_{\mathrm{P}}$.

The transition function δ_{PD} returns the new configuration, $\delta_{\mathrm{PD}}(idx, eifi, c_{\mathrm{PD}}) = c'_{\mathrm{PD}}$. The output function ω_{PD} simply returns the output to the external environment, $\omega_{\mathrm{PD}}(idx, c_{\mathrm{PD}}, eifi) = eifo$.

3.4 Model Run

A model run is computed by the function run_{PD}, which executes a number of steps in the combined model. It takes as inputs an initial configuration c^0_{PD}, a number of steps i, an external input sequence $eifiseq \in \mathbb{N} \to eifi$, and a computational sequence $seq_{\mathrm{PD}} \in \mathbb{N} \to PD$, which designates the interleaving of the processor and device steps. It returns an updated configuration c'_{PD} and an external output sequence $eifoseq \in \mathbb{N} \to eifo$.

The run function run_{PD} is defined by recursive application of δ_{PD}. For the base case, i.e., $i = 0$, we set $run_{\mathrm{PD}}(0, seq_{\mathrm{PD}}, eifiseq, c^0_{\mathrm{PD}}) = (c^0_{\mathrm{PD}}, \langle\rangle)$.

For $i+1$, let $(c_{\mathrm{PD}}, \mathit{eifoseq}) = \mathit{run}_{\mathrm{PD}}(i, \mathit{seq}_{\mathrm{PD}}, \mathit{eifiseq}, c^0_{\mathrm{PD}})$ denote configurations and outputs after executing i steps. To execute the $(i+1)$-th step, we apply the transition and output function of the combined model one more time. Let $c'_{\mathrm{PD}} = \delta_{\mathrm{PD}}(\mathit{seq}_{\mathrm{PD}}(i), \mathit{eifiseq}(i), c_{\mathrm{PD}})$ and $\mathit{eifo} = \omega_{\mathrm{PD}}(\mathit{seq}_{\mathrm{PD}}(i), \mathit{eifiseq}(i), c_{\mathrm{PD}})$. We define $\mathit{run}_{\mathrm{PD}}(i+1, \mathit{seq}_{\mathrm{PD}}, \mathit{eifiseq}, c^0_{\mathrm{PD}}) = (c'_{\mathrm{PD}}, \mathit{eifoseq} \circ (\mathit{eifo}, \mathit{seq}_{\mathrm{PD}}(i)))$.

4 Serial Interface (UART 16550A)

The *universal asynchronous receiver / transmitter* (UART) is a chip for communication over a serial interface. In the following we will simply speak of a serial interface.

A serial interface provides a computing device with the capability to send data via a few copper wires to another serial interface. This facility is used for non-network communication, e.g., with terminals, modems, and other computers.

In this section we describe the driver programmer's model of the serial interface chip UART 16550A [16]. Briefly summarized, the processor can send or receive data byte-wise. In case of a send this byte is stored in a FIFO queue, called the transmitter buffer, and later on sent to the external environment. A receiver buffer stores all incoming bytes from the environment. Reads from the programmer are served in FIFO manner, too. The UART provides the programmer with two methods to access status information for both buffers: either by interrupts or by polling special ports.

The transmitter and receiver queue are bounded in size (16 bytes); thus they may overrun. The receiver queue may overrun if the speed of incoming data from the environment exceeds the speed at which the processor can handle it. The transmitter queue may overrun if the processor writes data into the transmitter buffer faster than the serial interface can send out the data to the environment. Another error related to the queue size occurs when an empty receiver queue is read.

Our model handles the three error cases as follows: (i) overruns in the receiver queue are treated according to the UART specification, i.e., new incoming bytes are dropped, (ii) writing to a full transmitter queue and reading the empty receiver buffer are not allowed in our model, they are excluded through explicit software conditions.

Discharging these software conditions for a particular driver is tricky, and obviously it would be desirable to find easier accessible programming rules, e.g., write only if the transmitter buffer signals that it is empty. Proving that a concrete system will never let the receiver queue overrun, is even harder. A programmer would typically rely on run-time estimations, which ensure that the environment does not send data faster than the driver code can handle (when running on a certain machine). However, our model does not provide any real-time bounds on computations, and hence proving correctness of a concrete system would either require further assumptions over the environment (e.g., as part of a communication protocol between two serial interfaces) or correctness criteria which tolerate overruns.

For brevity, in this paper we do not go into detail regarding the *memory control register* and the *memory status register*. These registers are used to address additional

wires connected to some external modem and to configure hardware flow control. The remainder of this section is structured as follows. In Sect. 4.1 the configuration and the ports of the serial interface is detailed. The transition function δ_{u} of the serial interface is split into two logical parts: a processor- and an environment sided transition function. The first part is specified in Sect. 4.2, the second in Sect. 4.3. Finally in Sect. 4.4 all required software and environment restrictions are stated and different ways of discharging them are discussed.

4.1 Configuration

In the definition of the UART we use FIFO queues C_T of maximum size 16 for types T. For example, we use queues of type $C_{\mathbb{B}^8}$ to send and receive data.

We model these queues by cyclic buffers with head and tail pointers hd and tl. The buffer content ct maps indices to elements of type T. The number of queue entries is denoted as len.

Queues with $len = 0$ and $len = 16$ are called empty and full. The head element of a queue is accessed by $head(b) = b.ct(b.hd)$. Queues are manipulated by the operations $push$ and pop. The function $push$ adds a new byte to the queue at the tail pointer. It is only defined for non-full queues. We set $push(bdin, b) = b'$ where $b'.ct[b.tl] = bdin$, $b'.tl = (b.tl + 1) \bmod 16$ and $b'.len = b.len + 1$. The function pop deletes the element pointed to by the head pointer. It is only defined for non-empty queues. We set $pop(b) = b'$ where $b'.hd = (b.hd + 1) \bmod 16$ and $b'.len = b.len - 1$. A configuration of a serial interface c_{u} is a record with the following components:

1. The *transmitter holding buffer* $thb \in C_{\mathbb{B}^8}$ is a FIFO byte queue of size 16. Input bytes from the external environment are stored in chronological order. The transmitter holding buffer can be read byte-wise by the programmer.

2. The *receiver buffer* $rb \in C_{\mathbb{B}^8}$ is a FIFO byte queue of size 16. It can be written byte-wise by the programmer.

3. Interrupt driven mode configuration. The serial interface generates four types of interrupts (mapped to a single interrupt line). For each type two kinds of flags are maintained in the configuration: one indicating whether the interrupt type is enabled or disabled and the other indicating whether a corresponding interrupt is still pending.

- The *received data available interrupt* is generated, when the number of bytes in the receive buffer exceeds its interrupt trigger level. This level is computed as $itl(x) = 7x[1] + 3x[0] \cdot (x[1] + 1) + 1 \in \{1, 4, 8, 14\}$ for $x = c_{\mathrm{u}}.rbitl \in \mathbb{B}^2$. The component $c_{\mathrm{u}}.erdai \in \mathbb{B}$ indicates if the interrupt is enabled or not, while $c_{\mathrm{u}}.rdai \in \mathbb{B}$ indicates if the interrupt is currently pending.
- The *transmitter holding buffer empty interrupt* is generated if the transmitter buffer is empty. The component $c_{\mathrm{u}}.ethrei \in \mathbb{B}$ indicates if the interrupt is enabled or not, while $c_{\mathrm{u}}.threi \in \mathbb{B}$ indicates if it is currently pending.
- The *receiver line status interrupt* is generated if certain transmission errors occur. These are *overrun*, *parity*, *framing*, and *breaking* errors. The components $c_{\mathrm{u}}.oe$

specifies if an overrun in one of the two queues occurred. The *parity*, *framing*, and *breaking* errors are linked to particular bytes in the receiver queue. Their occurrence is saved in the 3-bit FIFO queue $c_{\mathrm{u}}.trerr \in C_{\mathbb{B}^3}$. For example, 110 encodes a parity and a framing error in the corresponding byte of the receiver queue.

– The *timeout interrupt* is generated if for a certain period of time no data was received or read from the receiver queue. The UART sets this timeout to the time needed to receive four bytes. This interrupt type can be used by the programmer to ensure that no data is forgotten in the receive queue after the input stream ended. The component $c_{\mathrm{u}}.toi \in \mathbb{B}$ indicates if the interrupt is currently pending. This interrupt is enabled iff the received data available interrupt is.

4. Polling mode configuration. The serial interface can also be operated in polling mode, in which the driver can check the status of the buffers by reading special ports. These ports map to three boolean configuration components: the data ready flag $c_{\mathrm{u}}.dr \in \mathbb{B}$, the empty transmitter holding buffer flag $c_{\mathrm{u}}.ethb$, and the empty data holding registers flag $c_{\mathrm{u}}.edhr$.
It is possible to mix interrupt and polling modes, e.g., the programmer could be informed about incoming data by an interrupt and then read the receiver buffer as long as it is non-empty.

5. Word length configuration. The following components can be set by the programmer, but do not affect the modeled behavior of our device. Nevertheless we need to model them: when connecting two serial interfaces the word length must be configured equally on both sides.
The serial interface uses a timer with a 115.2 kHz frequency; the *baud rate* is computed as $115200/c_{\mathrm{u}}.div$. Due to port overloading the programmer has to set a so-called *Divisor Latch Access Bit* $c_{\mathrm{u}}.dlab \in \mathbb{B}$ before accessing the $c_{\mathrm{u}}.div$ field.
The low-level encoding of the transmitted data (including error protection) is configured via the *word length* $c_{\mathrm{u}}.wl \in \mathbb{B}^2$, the stop bit length $c_{\mathrm{u}}.sbl \in \mathbb{B}$, and the parity select $c_{\mathrm{u}}.ps \in \mathbb{B}^3$. We omit details here.

The UART has eleven different registers, which are mapped to eight different addresses. Hence, some addresses are used in different contexts. They map to different registers either depending on the access type (read / write operation) or depending on the value of the divisor latch access bit $c_{\mathrm{u}}.dlab$ (see Table 1).

4.2 Processor-Side Transitions

As already mentioned, the transition function δ_{u} of the serial interface is split into two logical parts: a processor-side and an environment-side transition function.

The processor-side transition function $\delta_{\mathrm{u}}^{\mathrm{mem}}$ defines the behavior of the serial interface when communicating with the processor. Given a current configuration of the serial interface and an input from the processor, it computes an updated serial interface configuration and an output to the processor, i.e., $\delta_{\mathrm{u}}^{\mathrm{mem}}(c_{\mathrm{u}}, mifi) = (mifo, c'_{\mathrm{u}})$.

Table 1. Ports of the serial interface

UART Register / Buffer	Port	Abbreviation	Access Type
Transmitter Holding Buffer	0	THB_{p}	Write, $dlab = 0$
Receiver Buffer	0	RB_{p}	Read, $dlab = 0$
Divisor Latch Low Byte	0	$DLLB_{\mathrm{p}}$	Read / Write, $dlab = 1$
Interrupt Enable Register	1	IER_{p}	Read / Write
Divisor Latch High Byte	1	$DLHB_{\mathrm{p}}$	Read / Write, $dlab = 1$
Interrupt Identification Register	2	IIR_{p}	Read
FIFO Control Register	2	FCR_{p}	Write
Line Control Register	3	LCR_{p}	Read / Write
Line Status Register	5	LSR_{p}	Read

Note that the transition function $\delta_{\mathrm{u}}^{\mathrm{mem}}$ is only partially defined because some processor accesses to the device are considered illegal and lead to an undefined device configuration. Later on we will formulate software conditions excluding all these cases.

In the following we abbreviate a read access to port x by $rd(mifi, x) = mifi.rd \wedge mifi.a = x$ and a write access to port x by $wr(mifi, x) = mifi.wr \wedge mifi.a = x$. Although in general we allow devices to have ports of width 32 bit, the serial interface only has ports of width 8 bit. Hence, only the lower 8 bits of $mifi.din$ and $mifo$ are significant. In the following we assume that $mifo$ will be zero-padded by the device and omit these extra bits here.

Configuration Updates. If the bit $dlab$ is cleared and the processor reads the port receiver buffer having the address RB_{p} and the receiver queue of the serial interface is not empty then its first byte is popped. Furthermore the queue maintaining transmission errors for received bytes is updated, too:

$$rd(mifi, RB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 0 \wedge c_{\mathrm{u}}.rb.len > 0 \implies (c'_{\mathrm{u}}.rb = pop(c_{\mathrm{u}}.rb)) \wedge (c'_{\mathrm{u}}.trerr = pop(c_{\mathrm{u}}.trerr))$$

The processor writes the byte to be transmitted into the port transmitter holding buffer. If the corresponding queue is not full, the written byte is pushed into it:

$$wr(mifi, THB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.thb.len < 16 \implies c'_{\mathrm{u}}.thb = push(c_{\mathrm{u}}.thb, mifi.din[7:0])$$

Pending interrupt flags, raised by the device, are cleared if the processor reads the corresponding ports. Reading the receiver buffer clears the received data available- and the time-out interrupt. Similarly reading the interrupt identification register or reading the transmitter holding buffer clears the transmitter holding buffer empty interrupt. Finally the receiver line status interrupt is cleared by reading the line status register:

$$\begin{aligned} rd(mifi, RB_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.rdai = 0 \wedge c'_{\mathrm{u}}.toi = 0 \\ rd(mifi, THB_{\mathrm{p}}) \vee rd(mifi, IIR_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.threi = 0 \\ rd(mifi, LSR_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.rlsi = 0 \end{aligned}$$

By writing the port interrupt enable register, the programmer can specify which interrupt types are enabled, i.e., which can be raised by the device. Since the timeout interrupt is enabled when the received data available interrupt is, only three bits are relevant. The other five bits are ignored:

$$wr(mifi, IER_{\mathrm{p}}) \implies$$
$$(c'_{\mathrm{u}}.erdai = mifi.din[0]) \wedge (c'_{\mathrm{u}}.ethrei = mifi.din[1]) \wedge (c'_{\mathrm{u}}.erlsi = mifi.din[2])$$

The transmit or receive FIFO can be cleared manually by the programmer through writing the FIFO control register FCR_{p}. The bit zero of the FCR_{p} indicates if FIFOs should be used at all. If it is cleared no buffers will be used.

Setting bits one and two will clear the receiver and transmitter buffers, resp.:

$$wr(mifi, FCR_{\mathrm{p}}) \wedge mifi.din[1] = 1 \implies c'_{\mathrm{u}}.rb.len = 0 \wedge c'_{\mathrm{u}}.rb.hd = c_{\mathrm{u}}.rb.tl$$
$$wr(mifi, FCR_{\mathrm{p}}) \wedge mifi.din[2] = 1 \implies c'_{\mathrm{u}}.thb.len = 0 \wedge c'_{\mathrm{u}}.thb.hd = c_{\mathrm{u}}.thb.tl$$

Bit three indicates if DMA is supported. In this paper we do not deal with DMA. Bit four and five of the FCR_{p} are reserved.

If the received data available interrupt is enabled, the last two bits of the FCR_{p} encode at what length of the receive queue an interrupt is generated. Hence, the two bits map to the $rbitl$ component of the serial interface:

$$wr(mifi, FCR_{\mathrm{p}}) \implies c'_{\mathrm{u}}.rbitl = mifi.din[7:6]$$

The first two bits of the line control register are mapped to the transmission word length wl, bit two relates to the stop bit length sbl, bits three to five map to the parity select ps, bit seven is set to access the two divisor bytes, and bit six is reserved:

$$wr(mifi, LCR_{\mathrm{p}}) \implies (c'_{\mathrm{u}}.wl = mifi.din[1:0]) \wedge$$
$$(c'_{\mathrm{u}}.sbl = mifi.din[2]) \wedge (c'_{\mathrm{u}}.ps = mifi.din[5:3]) \wedge (c'_{\mathrm{u}}.dlab = mifi.din[7])$$

By writing the divisor latch high byte register $DLHB_{\mathrm{p}}$ and the divisor latch low byte register $DLLB_{\mathrm{p}}$ the divisor div is set:

$$wr(mifi, DLLB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 1 \implies c'_{\mathrm{u}}.div[7:0] = mifi.din[7:0]$$
$$wr(mifi, DLHB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 1 \implies c'_{\mathrm{u}}.div[15:8] = mifi.din[7:0]$$

Generated Output. The predicate is_int indicates if for a given configuration of the serial interface c_{u} at least one of the four interrupts types is pending:

$$is\_int(c_{\mathrm{u}}) = c_{\mathrm{u}}.threi \vee c_{\mathrm{u}}.rdai \vee c_{\mathrm{u}}.rlsi \vee c_{\mathrm{u}}.toi$$

In case of a write operation the 32-bit wide data output $mifo$ is irrelevant and therefore set to zero. In case of a read operation the first 24 bits of the output are filled with zeros since the serial interface operates byte-wise.

If the processor reads from the receiver buffer and the receive queue is not empty, the first byte is taken from the queue and returned to the processor:

$$rd(mifi, RB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 0 \wedge c_{\mathrm{u}}.rb.len > 0 \implies \\ mifo = head(c_{\mathrm{u}}.rb) \wedge c'_{\mathrm{u}}.rb = pop(c_{\mathrm{u}}.rb)$$

If the processor reads port $DLLB_{\mathrm{p}}$, the lower eight bits of the divisor component div are returned. If it reads port $DLHB_{\mathrm{p}}$, the upper eight bits of the divisor component div are returned:

$$rd(mifi, DLLB_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 1) \implies mifo = c_{\mathrm{u}}.div[7:0] \\ rd(mifi, DLHB_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 1) \implies mifo = c_{\mathrm{u}}.div[15:8]$$

When reading the interrupt enable register the output encodes the four flags indicating which interrupt types are enabled:

$$rd(mifi, IER_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 0) \implies mifo = 0^5 \circ c_{\mathrm{u}}.erlsi \circ c_{\mathrm{u}}.ethrei \circ c_{\mathrm{u}}.erdai$$

The type of the interrupt that caused the eev flag to be set can be checked by reading the interrupt identification register. For $rd(mifi, IIR_{\mathrm{p}})$ we define $mifo = 1100 \circ is \circ \neg is\_int(c_{\mathrm{u}})$ where the three interrupt status bits $is \in \mathbb{B}^3$ are defined as follows:

$$is = \begin{cases} 011 & \text{if } c_{\mathrm{u}}.rlsi \\ 010 & \text{if } \neg c_{\mathrm{u}}.rlsi \wedge (c_{\mathrm{u}}.rdai \vee c_{\mathrm{u}}.toi) \\ 110 & \text{if } \neg c_{\mathrm{u}}.rlsi \wedge \neg(c_{\mathrm{u}}.rdai \vee uart.toi) \\ 001 & \text{if } \neg uart.rlsi \wedge \neg(uart.rdai \vee uart.toi) \wedge uart.threi \end{cases}$$

The line status register is a read-only register which encodes the polling mode information of the transmitter and the receiver queues. Furthermore, in case of a transmission error (i.e., in case of *line status interrupt*), this register provides the error type. Remember that the component $c_{\mathrm{u}}.trerr$ stores parity, framing and break errors of all bytes in the receiver queue. When reading LSR_{p}, the errors occurred in the head of the queue are reported in bits two, three and four. Let $errQ$ denote whether at least one error occurred in any of the bytes in the queue. Reading the port LSR_{p} results in:

$$rd(mifi, LSR_{\mathrm{p}}) \implies \\ mifo = errQ \circ c_{\mathrm{u}}.edhr \circ c_{\mathrm{u}}.ethb \circ head(c_{\mathrm{u}}.trerr)[2:0] \circ c_{\mathrm{u}}.oe \circ c_{\mathrm{u}}.dr$$

The line control register can also be read out. As mentioned before it contains the parity select, word length, stop bit length, set break interrupt enable flag and the divisor latch bit components of the serial interface:

$$rd(mifi, LCR_{\mathrm{p}}) \implies mifo = c_{\mathrm{u}}.dlab \circ c_{\mathrm{u}}.ebi \circ c_{\mathrm{u}}.ps \circ c_{\mathrm{u}}.sbl \circ c_{\mathrm{u}}.wl$$

4.3 Environment-Side Transitions

We describe the interaction of the serial interface with the environment, which is given by the environment-sided transition function $\delta_{\mathrm{u}}^{\mathrm{env}}$. This function takes an input from the environment and a serial interface configuration and it returns an updated serial interface configuration and an output to the environment, i.e., $\delta_{\mathrm{u}}^{\mathrm{env}}(c_{\mathrm{u}}, eifi) = (eifo, c'_{\mathrm{u}})$.

The input from that environment is given by (i) a bit $eifi.tshrready$ indicating if the transmitter shift register is empty and hence the next byte of the transmitter queue can be sent, (ii) a bit $eifi.serdinvalid$ indicating if new and valid data was received, (iii) the serial input data $eifi.serdin$, (iv) three bits indicating parity, framing, and break error, $eifi.pe$, $eifi.fe$ and $eifi.be$, (v) and a bit $eifi.to$ indicating a time-out interrupt. Since no time is modeled, a non-deterministic input from the environment signals time-out. The only output of the serial interface to the environment is the byte being sent.

If the transmission shift register is empty $eifi.tshrready$ and the transmitter queue has data in it, $c_{\mathrm{u}}.thb.len > 0$, the first byte of the queue is sent, $head(c_{\mathrm{u}}.thb)$ to the external environment. Otherwise, a special empty output is transmitted, i.e., 0^8.

The byte written to the external environment is taken from the transmitter queue:

$$eifi.tshrready \wedge c_{\mathrm{u}}.thb.len > 0 \implies c'_{\mathrm{u}}.thb = pop(c_{\mathrm{u}}.thb)$$

If the receive queue is not full, received bytes are added to it. Furthermore the queue maintaining the parity, framing and break errors is updated for the received byte:

$$eifi.serdinvalid \wedge c_{\mathrm{u}}.rb.len < 16 \implies c'_{\mathrm{u}}.rb = push(c_{\mathrm{u}}.rb, eifi.serdin) \wedge c'_{\mathrm{u}}.trerr = push(c_{\mathrm{u}}.trerr, eifi.pe \circ eifi.fe \circ eifi.be)$$

If a new byte is received although the receive queue is full, then the new byte is dropped and an error is indicated by raising the overrun flag:

$$eifi.serdinvalid \wedge c_{\mathrm{u}}.rb.len = 16 \implies c'_{\mathrm{u}}.oe = 1$$

The interrupt pending signals are raised if (i) the transmitter queue is empty and the environment signals through $eifi.tshrready$ that the next byte can be sent, $c'_{\mathrm{u}}.threi = c'_{\mathrm{u}}.thbp.len > 0$, (ii) the length of the receiver queue reaches the specified trigger level (receive data available interrupt), $c'_{\mathrm{u}}.rdai = c_{\mathrm{u}}.rb.len \geq itl(c_{\mathrm{u}}.rbitl)$, (iii) or a framing, parity, break or an overrun error occurs (line status interrupt), $c'_{\mathrm{u}}.rlsi = eifi.fe \vee eifi.pe \vee c'_{\mathrm{u}}.oe$, (iv) or the receiver queue is non-empty and the external environment signals the occurrence of a time-out, $c'_{\mathrm{u}}.toi = eifi.to \wedge c'_{\mathrm{u}}.rb.len > 0$.

In the polling driven mode the configuration is updated similarly: (i) $c'_{\mathrm{u}}.ethb$ is set if the transmitter queue is not empty, $(c'_{\mathrm{u}}.thbp.len = 0)$, (ii) $c'_{\mathrm{u}}.dr$ is set if the receiver queue is non-empty, $(c'_{\mathrm{u}}.rb.len > 0)$, (iii) $c'_{\mathrm{u}}.edhr$ is set if both the transmitter queue and the shift register are empty, $(c'_{\mathrm{u}}.thbp.len = 0) \wedge eifi.tshrready$.

4.4 Software Conditions and Environment Restrictions

The transition function δ_{u} is not total. Undefined cases are related to over- and underruns of the queues and illegal accesses to unmodeled or write-only ports. Formally, we characterize these cases by predicates over memory input and UART configurations:

- The line-status register must not be written, $\neg wr(mifi, LSR_{\mathrm{p}})$, and the unmodeled ports MCR_{p} and MSR_{p} must not be accessed, $mifi.a \notin \{MSR_{\mathrm{p}}, MCR_{\mathrm{p}}\}$.
- The receiver buffer must not be read when empty and the transmitter buffer must not be written to when full. Formally, if $c_{\mathrm{u}}.dlab = 0$ then $rd(mifi, RB_{\mathrm{p}}) \implies c_{\mathrm{u}}.rb.len > 0$ and $wr(mifi, THB_{\mathrm{p}}) \implies c_{\mathrm{u}}.thb.len < 16$.

Only if these software conditions are met, we can assume the model to be accurate. The driver programmer is responsible for discharging them. For example, a driver which writes no more than 16 byte chunks between each two transmitter holding buffer empty interrupts, obviously fulfills the second condition.

For proving correctness of a driver implementation we need to impose further restrictions on the behavior of the environment.

Liveness. We need to assume liveness of the sending part: data in the transmitter buffer must eventually be sent, $\forall i\, \exists j > i\, .\, seq_{\mathrm{PD}}(j) = D_{\mathrm{uart}} \wedge eifiseq(j).tshrready = 1$.

Also the processor is assumed to be live, $\forall i\, \exists j > i\, .\, seq_{\mathrm{PD}}(j) = P$. While liveness can be assumed by the programmer, it has to be shown in the hardware correctness proof.

Overrunning Receiver Queue. The speed of the environment sending packets to the serial interface is not related in any sense to the speed of the processor; packets arrive completely non-deterministically. The question is: how can a driver programmer under these circumstances assure that no packets are lost due to overrunning queues?

This is a tricky task. In a first approach we might impose timing restriction on the environment. Hardware implementation details like caches, pipelining, etc. are invisible in the ISA. Thus, the numbers of instructions executed cannot be related to real time and a relation between transmission speed and ISA execution cannot be established.

Note that the problem of overrunning queues is not inherent to our way of modeling. It is a problem that a device programmer must expect and deal with in non-real-time operating systems, too. This situation leads to serious difficulties in the formalization of the correctness statements for serial interface drivers. For example, it is impossible to prove that all key presses sent from a keyboard to a serial interface are finally processed by the driver because the model contains runs in which the environment is too fast leading to a queue overrun. There are three approaches to deal with the problem:

```
 0: addi r3,r0,#Da(D_uart)  (1.1)
 4: addi r4,r0,#3           (1.2)
 8: sw LCR_p·4(r3),r4       (1.3)
12: sw IER_p·4(r3),r0       (1.4)
16: addi r0,#14, r4         (1.5)
20: sw FCR_p·4(r3),r4       (1.6)
24: lw r6, 0(r1)            (2.1)
28: sw THB_p·4(r3), r6      (2.2)
32: slri r6, r6, #8         (2.3)
36: sw THB_p·4(r3), r6      (2.4)
40: slri r6, r6, #8         (2.5)
44: sw THB_p·4(r3), r6      (2.6)
48: slri r6, r6, #8         (2.7)
52: sw THB_p·4(r3), r6      (2.8)
56: addi r1, r1, #4         (2.9)
60: lw r4, LSR_p·4(r3)      (3.1)
64: andi r4, r4, #32        (3.2)
68: beqz r4, #-12           (3.3)
72: nop                     (3.4)
76: subi r5, r5, #1         (4.1)
80: bnez r5, #-60           (4.2)
84: nop                     (4.3)
```

Fig. 2. UART driver. We assume that registers `r1` and `r2` are preset to a and n.

1. *Model overruns in specification and use software synchronization.* A widely used mechanism is called software flow control: the receiver signals the sender when ready / unable to accept new data via the special characters Xon / Xoff.

2. *Hardware synchronization.* Synchronization can also be implemented directly in hardware (called *autoflow control*), as was done for the UART 16750. Using such hardware an assumption can be introduced stating that the environment is not sending new data while the receiver buffer is still full.

3. *Worst case execution time (WCET) analysis.* Good run-time estimates require a cycle-accurate model for the target processor. Indeed, there are tools for several architectures to precisely estimate the WCET of given programs, e.g., [17]. By analyzing the serial interface driver and parts of the kernel, such as the interrupt handlers, we can compute the latency of processing data received at the serial interface. This yields a maximum baud rate under which the driver may be run safely without overruns.

5 Example: A Simple UART Driver

We construct a simple device driver and sketch its correctness proof with respect to the ISA of Sect. 3. The driver writes n words from the processor's memory, starting at address a, to the serial interface with index D_{uart} and base address $Da(D_{\mathrm{uart}})$. Its code is shown in Fig. 2; its size is approximately an order of magnitude smaller than the code of a realistic driver for the UART 16550A. We use a MIPS-like syntax. GPRs, immediates, and register-indexed memory operands are denoted as $\mathtt{r}k$, $\#l$, and $m(\mathtt{r}n)$. Lines are prefixed with an offset to a certain code base address cba. Arrows indicate jump targets; all jumps are executed with one delay slot.

To state the driver correctness, we use the auxiliary function $purge$. For a device index idx and an external output sequence $eifoseq$ it returns the sub sequence of external outputs for device idx.

Let seq_{PD} and $eifiseq$ denote a computational sequence and an external input sequence fulfilling the liveness assumption. Let c^0_{PD} denote an initial configuration which starts with the execution of the driver, $c^0_{\mathrm{P}}.dpc = cba$, and where the word

count and start address are stored in the first two registers, i.e., $c^0_{\mathrm{P}}.gpr[1] = a$ and $c^0_{\mathrm{P}}.gpr[2] = n$.

Furthermore let c^i_{PD} and $eifoseq^i$ denote the reached state and generated output after the execution $run_{\mathrm{PD}}(i, seq_{\mathrm{PD}}, eifiseq, c^0_{\mathrm{PD}})$ of some i steps of the combined system.

Theorem 1 (Functional correctness). *There exists some step number e, after which the driver finished execution and the n words from the processor's memory are output to the external environment:* $purge(eifoseq^e, D_{\mathrm{uart}}) = c^0_{\mathrm{P}}.m_{4\cdot n}(a)$

Proof. The main part of the code is the *outer loop* in parts (2) to (4). It is traversed n times, sending a word over the serial interface in each iteration. Before iteration $j < n$ and after iteration $j = n$ of the loop after a certain number $s(j)$ of steps the following invariants have to hold: (i) j words have been written to the environment, $purge(eifoseq^{s(j)}, D_{\mathrm{uart}}) = c^0_{\mathrm{P}}.m_{4\cdot j}(a)$, (ii) the first address not yet copied and the number of remaining words are stored in $gpr[1]$ and $gpr[4]$, and (iii) the device has an empty transmitter holding buffer, interrupts disabled, and a cleared $dlab$ flag. The existence of $s(j)$ and the invariants are shown by induction over j.

Initially, the device invariant is established by code part (1) writing the ports LCR_{p}, IER_{p}, and FCR_{p}. For $j > 0$, correctness of the code that copies a word from memory to the transmitter holding buffer, part (2), and the polling loop, part (3) have to be shown. After part (2), $4 - c_{\mathrm{u}}.thb.len$ bytes have been transmitted; the other bytes will have been transmitted after the polling loop exits. To show termination of these parts, the liveness condition over the computational and external sequence has to be applied.

6 Conclusion and Future Work

We have presented the detailed formal model of a serial interface controller, the UART 16550A [16]. By combining this model with the formal model of a processor ISA, we obtained a formal model of a processor in which the UART may be accessed as a memory-mapped device. All presented models have been specified in the theorem prover Isabelle/HOL [18]. The formalized ISA resembles the DLX instruction set architecture that was taken as a specification for the VAMP processor [2, 3].

Our Isabelle/HOL formalization defines a precise programming model for device drivers and may be used as the basis of an integrated, self-contained formal driver verification environment. Thus, it is relevant for both device programmers and verification engineers.

For the programmer, the model is a succinct description of the visible state of the device and its interaction with the external environment and the processor. Moreover, *environmental conditions*, which the programmer may assume, and *software conditions*, which the programmer must satisfy, precisely define the rules for implementing

a functionally correct device driver. An example of such a driver, transmitting data via a serial interface, was given in Sect. 5.

In addition, the model may be used by the verification engineer to develop mathematical software correctness proofs and to check them with a computer-aided verification system. A sketch of such a proof was given in Sect. 5. In contrast to related work, the high level of detail in our device models even allows the verification of complex properties like functional correctness rather than just control or protocol properties.

Our further work in this area can be split into two parts. First, we plan to formalize and extend the implementation and correctness proofs from Sect. 5 to cover data reception and successful communication between two serial interfaces. Second, in the broader context of the attempted system verifications in Verisoft, the scope of our modelling and verification effort needs to be extended to cover all system layers from the gate-level hardware of the VAMP processor [2, 3] with devices up to user-level device drivers for a variety of standard devices (e.g., serial interface, hard disk [6], FlexRay-like bus controller). The final result of this effort is a stack of computational models with device support; adjacent layers in this model stack will be related to each other by simulation theorems.

References

1. The Verisoft Consortium: The Verisoft Project. http://www.verisoft.de/ (2003)
2. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Geist, D., Tronci, E., eds.: CHARME'03. Volume 2860 of LNCS. Springer (2003) 51–65
3. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In Borrione, D., Paul, W., eds.: CHARME'05. Volume 3725 of LNCS. Springer (2005) 301–316
4. Schmaltz, J.: A formal model of lower system layer. In: FMCAD'06, IEEE/ACM Press (2006) 191–192
5. Knapp, S., Paul, W.: Pervasive verification of distributed real-time systems. In Broy, M., Grünbauer, J., Hoare, T., eds.: Software System Reliability and Security. Volume 9 of IOS Press, NATO Security Through Science Series. (2007) To appear.
6. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD '05, IEEE Computer Society (2005) 309–316
7. Cohen, B.: Component design by example: A step-by-step process using VHDL with UART as vehicle. VhdlCohen (2000)
8. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD, IEEE Computer Society / ACM (2003) 433–440
9. ALDEC – The Design Verification Company: UART nVS. http://www.aldec.com/products/ipcores/_datasheets/nSys/UART_nVS.pdf (2006)
10. Rashinkar, P., Paterson, P., Singh, L.: System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell, MA, USA (2001)
11. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In Dwyer, M.B., ed.: SPIN. Volume 2057 of LNCS. Springer (2001) 103–122
12. Microsoft Corporation: SDV: Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx (2004)
13. Hallgren, T., Jones, M.P., Leslie, R., Tolmach, A.P.: A principled approach to operating system construction in Haskell. In Danvy, O., Pierce, B.C., eds.: ICFP, ACM (2005)
14. Holzmann, G.J.: New challenges in model checking. http://www.easychair.org/FLoC-06/holzmann_25mc_floc06.pdf (2006) Symposium on 25 years of Model Checking, Seattle, USA. Invited talk.

15. Müller, S., Paul, W.: Computer Architecture: Complexity and Correctness. Springer (2000)
16. National Semiconductor: PC16550D – universal asynchronous receiver / transmitter with FIFO's. `http://www.national.com/ds.cgi/PC/PC16550D.pdf` (2005)
17. Ferdinand, C., Heckmann, R.: Verifying timing behavior by abstract interpretation of executable code. In Borrione, D., Paul, W., eds.: CHARME'05. Volume 3725 of LNCS. Springer (2005) 336–339
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

A Mechanization of Phylogenetic Trees

Mamoun Filali
filali@irit.fr

IRIT CNRS
Université Paul Sabatier
118 Route de Narnonne
F-31062 Toulouse France

Abstract. We study the mechanization of phylogenetic trees in higher order logic. After characterizing trees within such a logic, we state how to reason and to compute about them. We introduce the so called generative partitions and relations whose purpose is to allow the reconstruction of a tree from its leaves. After introducing tree transformations, we define the graft operation. and consider sufficient conditions for the preservation of the generative partitions or relations after a graft. It follows that we can reconstruct a tree given its set of leaves and its generative relation which has been preserved along the growth of the tree. We apply this result to the reconstruction of a distributed computation.
keywords: HOL, tree structure, verification, ISAR.

1 Introduction

This paper gives a definitional formalization, in higher order logic (HOL), of phylogenetic trees. We also formalize how to reason and compute on such trees. We define the notion of a generative relation, that aims at characterizing information which enables to rebuild a tree. Finally, we propose a reconstruction algorithm based on the set of leaves and a generative relation. The correctness of the algorithm is established. We introduce an operation, the graft, that allows to represent the growth of a tree. a graft are stated. We illustrate such a reconstruction through the so-called leaf vectors and a concrete generative relation. It should be stressed that our study is not only concerned with the proposal of an original algorithm but also by the formal definitions and proofs within a logical framework.

The rest of this paper is organized as follows: Section 2 gives the representation and the basic operations. Section 3 introduces the graft operation and studies its reconstruction properties.Section 4 presents a concrete example where we apply the reconstruction algorithm. Section 5 contains the conclusions and related works.

2 A phylogenetic tree representation and basic operations

In this section, we introduce the formal representation of phylogenetic trees; For such a representation, we consider how to reason about it and how to compute on it. We rely mainly on basic set theory. However, rather than working

with set theory only, we use type theoretic reasoning also. We have done the mechanization within the Isabelle logical framework [13]. Actually, we have used the Isabelle/Isar[1] [19] environment which goal is to assist in the development of human-readable proof documents composed by the user and checked by the machine.

2.1 Notations and basic definitions

In this section, we recall the basic set theory and order notions, we will use.We hope that the name of the definitions and their formal expression are self explanatory. We have used the definitions given in [6]. Moreover, we express them in the Isabelle syntax [13]. For each definition, first, we give its signature, then its formal expression. For instance, we have used the following definitions:

```
"Maximal ≜ λ S. {m ∈ S. ∀ m' ∈ S. m ⊆ m' ⇒ m = m'}"

"Down ≜ λ(S,e). {s ∈ S. s ⊆ e}"

"PDown ≜ λ(S,e). {s ∈ S. s ⊂ e}"

--{* proper partition*}
"PPartition ≜ λ (n,S).Partition(n,S) ∧ (∀ e ∈ S. e ⊂ n)"

"A//r ≜ ⋃x ∈ A. {r``{x}}"  -- {* set of equiv classes *}
```

In Isabelle, the reflexive transitive closure of relation `r`, denoted `r^*`, is introduced as an inductive data type [2]. Its introduction rules are `rtrancl_refl` which specifies that every couple `(a,a)` belongs to the transitive closure, and `rtrancl_into_rtrancl` which specifies that if `(a,b)` belongs to `r^*` and `(b,c)` belongs to `r`, then `(a,c)` belongs also to `r^*`.

```
inductive "r^*"
intros
  rtrancl_refl : "(a, a) ∈ r^*"
  rtrancl_into_rtrancl :
   "(a, b) ∈ r^* ⟹ (b, c) ∈ r ⟹ (a, c) ∈ r^*"
```

With respect to the proofs, we have used the Isabelle/Isar format. A proof is established by a sequence of intermediate results which has to be proved recursively or already established. Eventually, results are justified either as axioms of the logic or by rules of the logic. With respect to proofs, Isar promotes the

[1] "Isar" abbreviates "Intelligible semi-automated reasoning".

so called "declarative style" [18] which is closer to the usual mathematical reasoning than the procedural format. Let us mention that, basically, Isar supports natural deduction but also supports calculational reasoning [7].

As an example, the following statement which consists in assumptions[2] (**assumes**), a conclusion (**shows**) and a proof script (**proof**) establishes that the union of two hierarchies (see section 2.2) is also a hierarchy. A basic statement of the proof has the format:

from ⟨facts⟩ **have** label $'$:$'$ ⟨proposition⟩ **by** ⟨method⟩

which aim is to establish `proposition` from `facts` by applying `method`.

```
theorem Hierarchy_union:
 assumes h1:"H1 ∈ Hierarchy"
 assumes h2:"H2 ∈ Hierarchy"
 assumes s: "∀ n1 ∈ H1. ∀ n2 ∈ H2. SDS(n1,n2)"
 shows "(H1 ∪ H2) ∈ Hierarchy"
proof −
 from h1 h2 have e: "∅ ∉ H1 ∪ H2" by (unfold
    Hierarchy_def, blast)
 from s have "∀ n1 ∈ H1. ∀ n2 ∈ H2. SDS(n2,n1)"
   by (auto simp only: SDS_def)
 from this have "∀ n1 ∈ H2. ∀ n2 ∈ H1. SDS(n1,n2)" by
    auto
 from s this h1 h2 have sds: "∀ n1 ∈ H1 ∪ H2. ∀ n2 ∈ H1 ∪
     H2. SDS(n1,n2)"
   by (unfold Hierarchy_def, blast)
 from h1 h2 have f: "finite (H1 ∪ H2)" by (unfold
    Hierarchy_def, auto)
 from h1 h2 have "∀ n ∈ H1 ∪ H2. finite n" by(unfold
    Hierarchy_def, auto)
 from e sds f this show ?thesis by (unfold Hierarchy_def,
     blast)
qed
```

2.2 Hierarchies and trees

Our mechanization is based on the introduction of phylogenetic trees starting from the basic notions of set theory. For such a purpose, we first consider hierarchies [3] and then introduce trees as restricted hierarchies. Along with these hierarchies, we give some general definitions that will be used later.

[2] Sometimes assumptions are also called preconditions.

The basic idea of the following representations is to infer a structure from the relations between its elements; the structure is not encoded directly. Such a content based encoding is motivated by the fact that our basic concern is the reconstruction starting from *some* of the elements, namely the leaves, of the tree structure.

Hierarchies. We first introduce a generic `graph` as a set of nodes. A `node` is a set of generic elements.

```
types
  'e graph = "('e set) set" -- {* generic graph *}
  'e node = "('e set)" -- {* generic node *}
```

Hierarchies are finite graphs which elements are finite and non empty and obey to the `SDS`: "Subset Disjoint Subset" relation:

```
  "SDS ≜ λ(s1,s2). s1 ⊆ s2 ∨ s1 ∩ s2 = ∅ ∨ s2 ⊆ s1"

  "Hierarchy ≜ {H.    finite(H)
                  ∧ (∀ n ∈ H. finite(n))
                  ∧ ∅ ∉ H
                  ∧ (∀ n1 ∈ H. ∀ n2 ∈ H. SDS (n1,n2))
               }"
```

In the following, we give the formal definitions that will be used.

```
  "Leaves ≜ λ h. {l ∈ h. PDown(h,l) = ∅}"

  "ROOT ≜ λ h. ⋃ h"

  "Subtrees ≜ λ t. image (λ e. Down(t,e)) (Maximal(t))"

  -- {* proper subtrees *}
  "PSubtrees ≜ λ t. Subtrees (t - Maximal(t))"

  -- {* roots of proper subtrees, child nodes *}
  "R1 ≜ λ t. image ROOT (PSubtrees(t))"

 "Sigma ≜ λ S. {⋃ (⋃ S)} ∪ (⋃ S)"
```

Due to the lack of space, we do not state all the established results. We will give them on the fly when needed.

Trees and phylogenetic trees. Starting from hierarchies, we first define a tree as a hierarchy with its `ROOT` as the single maximal element :

```
"Tree ≜ {h ∈ Hierarchy. Maximal(h) = {ROOT(h)}}"
```

Then, we introduce phylogenetic trees as trees which nodes are either leaves or the union of all its subnodes:

```
"Phylo ≜
  {t ∈ Tree. ∀ n ∈ t. n ∈ Leaves(t) ∨ n = ⋃ PDown(t,n)}"
```

With respect to phylogenetic trees, we just mention the following equality that will allow us to say that the reconstruction can proceed starting from the leaves, while the statement of the reconstruction theorem is over the root. Actually, for a phylogenetic tree t, we have: $\mathrm{ROOT}(t) = \bigcup \mathrm{Leaves}(t)$. Moreover, we will rely on the following result about the union of phylogenetic trees:

```
lemma phylo_union:
 assumes t1: "t1 ∈ Phylo"
 assumes t2: "t2 ∈ Phylo"
 assumes u: "ROOT(t2) ∈ Leaves(t1)"
 shows "t1 ∪ t2 ∈ Phylo"
proof ... qed
```

Examples. The figure 1 illustrates the representation of phylogenetic trees. For instance, with respect to the previous definitions and the tree let `t2`, we have:

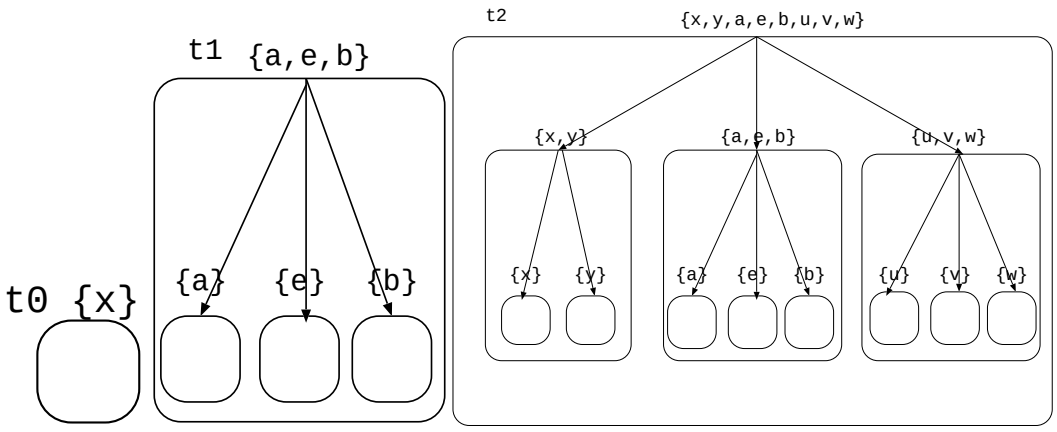


Fig. 1. phylogenetic trees

$$
\begin{aligned}
t2 = \quad & \{\{x, y, a, e, b, u, v, w\}, \{x, y\}, \{x\}, \{y\} \\
& , \{a, e, b\}, \{a\}, \{e\}, \{b\}, \{u, v, w\}, \{u\}, \{v\}, \{w\}\} \\
\text{Leaves}(t2) = \quad & \{\{x\}, \{y\}, \{a\}, \{e\}, \{b\}, \{u\}, \{v\}, \{w\}\} \\
\text{ROOT}(t2) = \quad & \{x, y, a, e, b, u, v, w\} \\
\text{R1}(t2) = \quad & \{\{x, y\}, \{a, e, b\}, \{u, v, w\}\}
\end{aligned}
$$

The decomposition and induction theorems. In order to reason about phylogenetic trees, we first introduce a decomposition theorem: a tree is either a singleton containing its `ROOT`, or the sum (`Sigma`) of its proper subtrees.

```
theorem phylo_cases:
  assumes t: "t ∈ Phylo"
  shows "t = {ROOT(t)} ∨ t = Sigma (PSubtrees(t))"
proof ... qed
```

We state the induction theorem about phylogenetic trees as follows:

```
theorem phylo_induct:
  assumes b: "∀ e. P({e})"
  assumes r: "∀ T ∈ domSigma. (∀ t ∈ T. t ∈ Phylo ∧ P(t))
       ⇒ P(Sigma(T))"
  shows "∀ t ∈ Phylo. P(t)"
proof ... qed
```

where `domSigma` specifies the set of trees which can be "joined" to form a phylogenetic tree:

```
  "domSigma ≜
     {S. S ≠ ∅ ∧ finite(S) ∧ S ⊆ Tree
        ∧ (∀ t1 ∈ S. ∀ t2 ∈ S. t1 ≠ t2 ⇒ ⋃ t1 ∩ ⋃ t2 = ∅)
        ∧ PPartition(⋃ ⋃ S, image ROOT S)
     }"
```

2.3 Transformations

The basic property of the studied transformations is to preserve the underlying structure while transforming the nodes.

Hierarchy transformations and preservation theorem. First, we introduce general transformations which basic property is to preserve the cardinality of a set of nodes.

```
 "G_tr ≜ λ g. {tr. ∀ n1 ∈ g. ∀ n2 ∈ g.
                  (tr(n1) = tr(n2)) = (n1 = n2)}"
```

A hierarchy transformation is a general transformation which preserves the relations between the nodes of a hierarchy:

```
{* hierarchy transformations set*}
"H_tr ≜ λ h.
 {tr ∈ G_tr(h). (∀ n ∈ h. finite(n) ⇒ finite(tr(n)))
   ∧(∀ n ∈ h. n ≠ ∅ ⇒ tr(n) ≠ ∅)
   ∧(∀ n1 ∈ h. ∀ n2 ∈ h. n1 ⊆ n2 ⇒ tr(n1) ⊆ tr(n2))
   ∧(∀ n1 ∈ h. ∀ n2 ∈ h. n1 ∩ n2 =∅ ⇒ tr(n1) ∩ tr(n2) =∅)
 }"
```

A hierarchy is preserved by a hierarchy transformation:

```
theorem hierarchy_trans:
assumes t: "t ∈ Hierarchy"
assumes tr: "tr ∈ H_tr(t)"
shows "image tr t ∈ Hierarchy"
proof ... qed
```

A tree is also preserved by a hierarchy transformation.

Phylogenetic transformations and preservation theorem. Intuitively, when a phylogenetic transformation is applied to a non-leaf node, the decomposition into its descendant nodes is preserved. The characterizing property of a phylogenetic transformation is expressed as follows:

```
 "P_tr ≜ λ h. { tr ∈ H_tr(h). ∀ n ∈ h.
   n ∈ Leaves(h) ∨ tr(n) = ⋃ (image tr (R1(Down(h,n))))}"
```

Then, we state the preservation theorem:

```
theorem phylo_trans:
 assumes t: "t ∈ Phylo"
 assumes tr: "tr ∈ P_tr(t)"
 shows "image tr t ∈ Phylo"
proof ... qed
```

Example. A `Mutation` is a transformation that concerns the nodes up a graph node: `gp`, such "up" nodes contain `gp`, and a mutation is expressed as follows:

```
  "Mutation ≜ λ(gp,R).λ n. if gp ⊆ n then (n − gp) ∪ R else n"
```

We show that the `Mutation` transformation is a phylogenetic transformation:

```
theorem Mutation_P_tr:
  assumes h: "h ∈ Phylo"
  assumes g: "g ∈ Phylo"
```

```
  assumes pre: "PreGraft(h,gp,g)"
  shows "Mutation(gp,ROOT(g)) ∈ P_tr(h)"
proof ... qed
```

where `PreGraft` (We will use this predicate as the precondition of the `Graft` operation.) is defined as follows:

```
  "PreGraft ≜ λ(h,gp,g). h ∈ Hierarchy ∧ (((ROOT h) ∩ (ROOT g)) = ∅) ∧
                  gp ∈ Leaves(h) ∧ g ∈ Hierarchy ∧ g ≠ ∅"
```

2.4 Generative partitions and relations

One of our concerns is the reconstruction of a phylogenetic tree starting from the set of its leaves. The basic idea of such a reconstruction is to partition the leaves according to its direct proper subtrees and to apply recursively the reconstruction to each of the sets of the partition. These successive partitions define the sets which are generated by a generative partition.

Generative partitions. Given a phylogenetic tree `h`, `GP` is called a generative partition of `h`, if each node is either a leaf or partitioned according the direct sub-roots of `n` (`Down(h,n)` is the subtree of `h` which root is `n`).

```
"GenerativePartition≜ λ(h,GP). ∀ n ∈ h.
  GP(n) =(if n ∈ Leaves(h) then {n} else R1(Down(h,n)))"
```

Generative relations. Semantically, the generative relation is a symmetric relation of which the transitive closure is a generative partition. The motivation for introducing generative relations is to make local the reasoning about the of growth the tree and consequently easier than a global one. First, we define `R2P` which converts a relation to the partition function given by its reflexive and transitive closure: a node `n` is partitioned by the classes of the corresponding equivalence relation.

```
 "R2P(r) ≜ λ n. (n // ((r(n))^*))"

"GenerativeRelation ≜ λ (h,GR).
        (∀ n ∈ h. GR(n) ⊆ n × n ∧ sym(GR(n)))
     ∧ GenerativePartition(h, R2P(GR))"
```

2.5 The reconstruction function and theorem

We introduce the auxiliary function `Reconstruct`. Its definition is set up in order to be accepted as a well defined function by Isabelle: since it is a recursive function that is not primitive, we have to provide a `measure` that decreases

at each call. The condition of the `if` expression ensures it. The `reconstruct` function is a curryfied version of `Reconstruct`.

```
recdef Reconstruct "measure (λ (GP,s). card s)"
"Reconstruct(GP,s) =
  (if (finite s) ∧ (∀ s' ∈ GP(s). s' ⊂ s) then
     Sigma (image (λ e. Reconstruct(GP,e)) (GP s))
   else {s})"
(hints simp add: psubset_card_mono)

"reconstruct(GP) ≜ λ s. Reconstruct(GP,s)"
```

The theorem characterizing the reconstruction is stated as follows:

```
theorem generative_partition_reconstruction:
shows
 "∀ GP. ∀ t ∈ Phylo. GenerativePartition(t,GP)
                      ⇒ (reconstruct(GP)(ROOT(t)) = t)"
proof ... qed
```

This theorem is established thanks to the induction theorem over phylogenetic trees (2.2).

2.6 Discussion

In this section, we discuss the definition of phylogenetic trees that has been elaborated. With respect to the structural point of view: a phylogenetic tree can be defined as either a singleton node or as the *Sigma* of its subtrees. Such a set based construction is not admitted by most of the type theory based logical frameworks [9, 1, 4, 13]. In fact, in such frameworks a tree is usually recursively defined through the *list* of its subtrees, or through a map of its subtrees from a given index type. We have tried to work with each of these representations. Their main drawback is to break the underlying natural confluence. For instance, with such representations, inserting a subtree after actually removing it, does not yield the original tree. Such a confluence is fundamental for establishing naturally our reconstruction result. Otherwise, we would have to introduce modulo relations in order to not distinguish between trees of which subtrees are identical but not in the same order.

3 The graft operation

In our setting, the graft operation models the growth of a tree. As its name suggests, the graft operation consists in grafting a tree at a given node. In this study, we consider a restricted version: grafting occurs at singleton nodes only.

3.1 Graft decomposition

Let `h` be a graph, `gp` a node of `h` where the graft should occur and `g` the graph to graft. We express the graft through two basic operations: first, `h` is transformed through a `Mutation`, second, `g` is grafted through the union ($\cup$) operation. Such a decomposition is illustrated by figure 2. The `Graft` is expressed as follows:

```
"Graft ≜ λ(h,gp,g).(image (Mutation (gp,ROOT(g))) h) ∪ g"
```

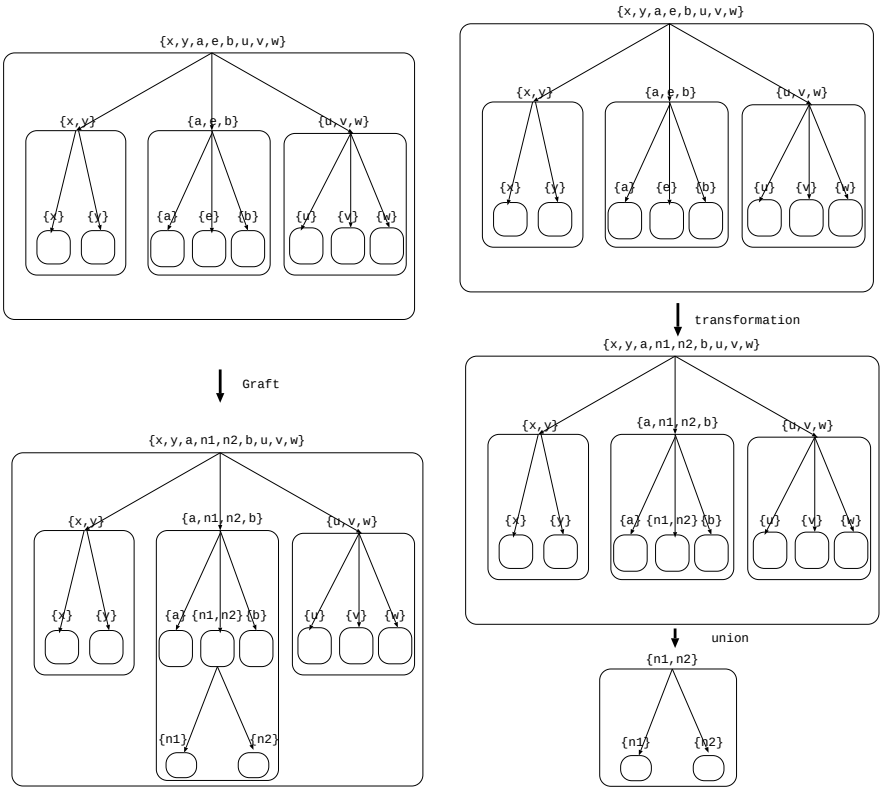


Fig. 2. Decomposition of a graft

We show that the grafted tree is also a phylogenetic tree. The proof is established thanks to the decomposition of the graft operation; we first establish that `Mutation` is a phylogenetic transformation, then thanks to the union theorem, `g` being phylogenetic, it follows that the grafted tree is phylogenetic.

```
theorem graft_phylo:
 assumes h: "h ∈ Phylo"
 assumes g: "g ∈ Phylo"
 assumes pre: "PreGraft(h,gp,g)"
 shows "Graft(gp,g)(h) ∈ Phylo"
proof ... qed
```

3.2 Reconstructing a graft through a generative partition

This section states a general result about the preservation of a generative partition `GP`. In fact, we have a precondition about the partitioning of the mutated nodes.

```
theorem generative_partition_graft_phylo:
  assumes h: "h ∈ Phylo"
  assumes g: "g ∈ Phylo"
  assumes pre: "PreGraft(h,gp,g)"
  assumes gp_h: "GenerativePartition(h,GP)"
  assumes gp_g: "GenerativePartition(g,GP)"
  assumes gp_tr:
    "∀ n ∈ h. GP(Mutation(gp,ROOT(g))(n)) =
       (if n ∈ Leaves(h) then {Mutation(gp,ROOT(g))(n)}
        else image (Mutation(gp,ROOT(g))) (GP(n)))"
  shows "GenerativePartition(Graft(h,gp,g),GP)"
proof ... qed
```

3.3 Reconstructing a graft through a generative relation

In the same way, a generative relation can be preserved while extending a tree through a graft. Thanks to this preservation: a tree, growing through graft operations, will always be reconstructible from its leaves through its invariant generative relation.

A simplified mutation: the basic update `upd`. For the purpose of our application, we consider the following node transformation:

```
"upd ≜ λ (l,N). λ S. if l ∈ S then S − {l} ∪ N else S"
```

Since we have: $\mathrm{upd}(l, N) = \mathrm{Mutation}(\{l\}, N)$, `upd` inherits the property of `Mutation`; then it is a phylogenetic transformation.

Moreover, in order to simplify the proof obligations for establishing that a generative relation is preserved after a graft, we have elaborated sufficient conditions that should be established by the update function. Due to the lack of space, we do not detail them.

We have established the preservation of the generative relation for the graft of a so called canonical tree which consists of a root and a set of leaves:

```
"Canonic ≜ λ N. {N} ∪ (⋃ e ∈ N. {{e}})"
```

We have the following invariant theorem establishing the preservation of a generative relation when grafting a canonic tree:

```
lemma generative_relation_graft_phylo:
 assumes t: "t ∈ Phylo"
 assumes up: "{l} ∈ t"
 assumes gp1: "GenerativeRelation(t,GR)"
 assumes gr_m: "∀ n. GR(n) ⊆ n × n ∧ sym(GR(n))"
 assumes terminal: "Terminal(t)"
 assumes N: "∀ n ∈ t. N ∩ n = ∅"
 assumes N_e: "N ≠ ∅ ∧ finite(N)"
 assumes gp2: "GenerativeRelation(Canonic(N),GR)"
 assumes gr_tr:
  "∀ n ∈ t. l ∈ n ⇒ r_upd(l,N)(n)(GR(n),GR(upd(l,N)(n)))"
  shows "GenerativeRelation(Graft({l},Canonic(N))(t),GR)"
proof ... qed
```

4 Application

As an application of phylogenetic trees, we consider distributed diffusing computations. In the initial state, one site (or process) multi-casts a message to a subset of other nodes. Then, all nodes share the same behavior: when a message is received, the receiver performs a computation step and, possibly, multi-casts a message to a subset of other nodes.

We are interested in the following problem: how to reconstruct the global history of such a computation, after its termination[3], from information gathered during the computation. For such a diffusing computation, the control flow is a tree in which the nodes are the computation steps and the edges are the message communications. Our algorithm consists in collecting an encoded representation of these leaves. From this leaves set, we apply the reconstruction algorithm based upon a generative relation defined on the computation as a phylogenetic tree.

4.1 Control tree encoding

We define an encoding for the control tree. The nodes generated during the computation (temporary leaves) are encoded as vectors. At each site, a local counter is incremented by $p - 1$ each time a computation step multi-casts p messages. Thus, one[4] plus the sum of the local counters represents the number of the control tree leaves. Moreover, the value of the counter of site s is the maximum of the vectors component at the index s.

[3] Such a reconstruction is usually used for debugging purposes.

[4] We have to take into account the initial states where the counters are all null and the tree consists of one leaf node.

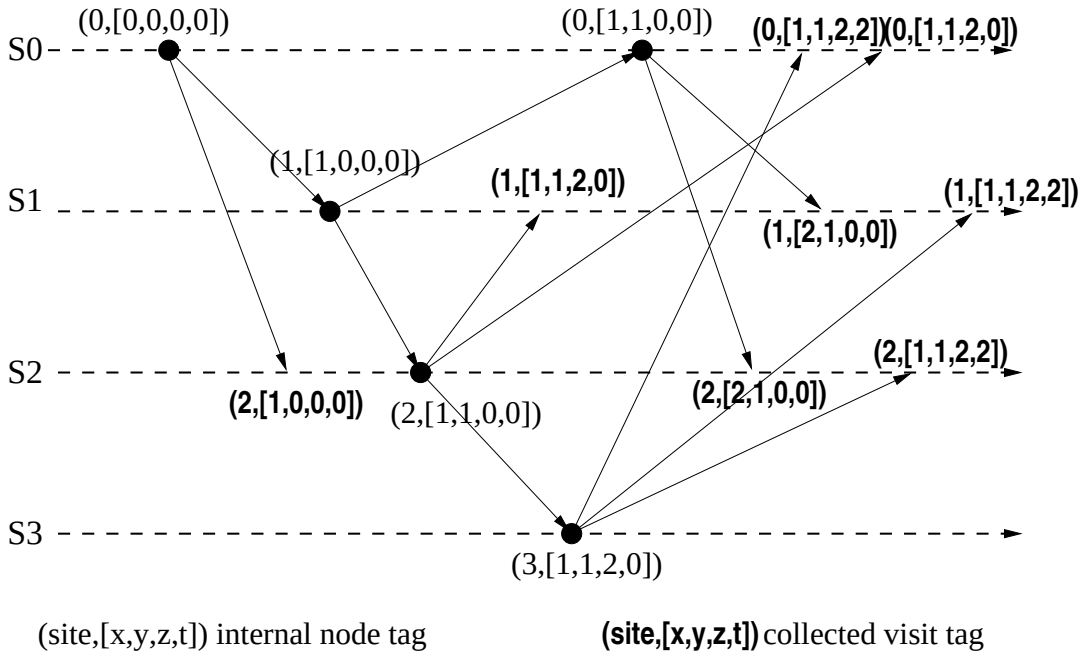


Fig. 3. Visit tags encoding

We associate a "visit tag" to each node. This tag is composed of the site of the node and a natural integers vector. This vector V has a size N, corresponding to the number of sites, and is assigned the local counter values of the sites it has visited. Figure 3 shows the tagging of the nodes of a diffusing computation with this encoding.

The state space of all the application is modeled by a global type `State`. It contains the fields related to the network, the local computations and the collector. The computation is concerned by the following fields:

- The field `lcounter` implements the local counter of each site;
- The field `collected` records the visit tags of the computation leaves.

Two transitions are considered and each of them is launched when a message is received:

- `ReceiveAndEnd` describes a computing step without further message multicast. In this case, the visit tag contained in the received message is sent to the collector;
- `ReceiveAndSplit` describes a computing step terminated by a message multicast. In this case, a new tag is created: it holds the destination site (d) and a vector which is identical to the tag vector of the splitting node ($m.V$), except for the splitting site (`self`) component, which gets the new local counter value $lcounter[self]$ assigned by this computation step. No message is sent to the collector.

With respect to phylogenetic trees, the diffusing computation is seen as a tree. A `ReceiveAndEnd` assigns to a node the definitive leaf status. While a `ReceiveAndSplit` extends a tree with new leaves. We interpret it as a `Graft` operation. Then, for validation purposes, we have an auxiliary variable `auxTree` for recording the growth of such a "superposed" tree: we prove that at termination, this auxiliary tree and the reconstructed tree are the same.

4.2 Termination detection and reconstruction

We introduce a collector process to gather vectors: a vector is sent to the collector when a process performs a computation step without multi-casting a new message. In such a case, this step generates a leaf with respect to the control flow of the computation. Then, with respect to phylogenetic trees, the collected tagged messages are in fact leaves of the phylogenetic tree superposed to the diffusing computation (and recorded in the auxiliary variable `auxTree`).

The reconstruction of the control tree can only start when the global computation is terminated. Several distributed algorithms can solve the termination problem, especially, thanks to a collector process[12]. However, the encoding itself provides a simple criterion for termination detection [8]: a computation is terminated when the number of collected leaves is equal to one plus the sum of the elements in the maximum of the collected visit vectors[5]:

$$\mid \text{collected} \mid = 1 + \sum_{s \in \text{Site}} \max_{v \in collected} v.V[s]$$

The generative relation for the diffusing computation as a phylogenetic tree is defined as follows:

```
"gr ≜ λ n. {(v1,v2). v1 ∈ n ∧ v2 ∈ n ∧
              (if V(v1) = min_on(n) ∨ V(v2) = min_on(n)
                 then (v1 = v2)
               else (∃ s. V(v1)(s) = V(v2)(s) ∧ s ≠ w(v1) ∧
                      s ≠ w(v2) ∧ V(v1)(s) ≠ min_on(n)(s)))
            }"
```

We derive the correctness of the reconstruction through the following invariant:

```
"ReconstructionInvariant ≜ λ st. auxTree(st) =
   reconstruct(R2P(gr))(collected(st) ∪ network(st))"
```

Then, when termination is reached, the network is empty, and the reconstruction applied to the collected messages gives the computation tree.

[5] |_| denotes the cardinality of _ .

5 Conclusion

In this paper, we have proposed a mechanization of phylogenetic trees. Starting from basic set theory, we have introduced phylogenetic trees through hierarchies and trees. Then, we have defined generic transformations. We note that sets based representations, although already suggested in the literature[10], are not widely used in computer science. To the best of our knowledge, the representation of a tree through the set of its leaves together with a generative partition or relation, as well as the study of dedicated transformations, are original. We have given a concrete example, where such notions have been applied to tree reconstruction and shown how such a reconstruction could be validated. It is interesting to remark that thanks to theorem proving techniques, such a validation was possible; actually we have considered an unknown number of nodes and unbounded natural vectors. Usual model checking techniques cannot handle such problems.

Most of our results have been proved formally within Isabelle. In fact, our trees are "unordered" trees. Such a data type could be considered as an inductive data type where `Sigma` would play the role of a constructor; however, due to the negative occurrence [6] most of the logical frameworks (HOL [9], Isabelle [13], PVS [4], Coq [1]) do not support such a definition schema. Vos and Swiestra [17] have studied restrictions for accepting inductive data types with negative occurrences; since our trees are finite, we could have reused their work. This work is not known to be available within the Isabelle framework. An alternative way would have been to introduce "unordered" trees through an equivalence relation [14] over ordered trees where subtrees are constructed with a list. It would be interesting to compare the subsequent developments of phylogenetic trees, generative relations and partitions.

With respect to the formalization of trees and biology related results, numerous works have been published. Among the more recent, we can cite [16] who consider the problem of tree inclusion in a categorical setting. [11] reviews basic network models for reasoning about biology; he notices that applications to biology of existing tools from algebra is just beginning. To the best of our knowledge, the mechanization of these works has not been considered yet. We think that our work could be reused as a starting point for establishing algorithms correctness but also for the correctness of their proposed proofs[7].

[6] The negative occurrence is due to the fact that the parameter of `Sigma`, considered as a constructor, is a set of trees.

[7] It is interesting to remark that the analysis of the algorithm of [5] is reported to be incorrect in [15].

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. http://coq.inria.fr.
2. S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Springer-Verlag, editor, *Theorem Proving in Higher Order Logics*, volume 1690, pages 19–36, 1999.
3. S. Bocker and A. W. Dress. A note on maximal hierarchies. *Advances in Mathematics*, (151):270–282, 2000.
4. S. Crow, S. Owre, J. Rushby, N. Shankar, and S. Mandayam. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton*, http://www.csl.sri.com/pvs, April 1995.
5. J. Culberson and P. Rudnicki. A fast algorithm for constructing trees from distance matrices. *Information Processing Letters*, 30(4):215–220, may 1989.
6. B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.
7. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1989.
8. M. Filali, P. Mauran, G. Padiou, P. Quéinnec, and X. Thirioux. Refinement based validation of a distributed termination detection algorithm . In *FMPPTA'2000 , Cancun*, volume 1800 of *Lecture Notes in Computer Science*, pages 1027–1036. Springer-Verlag, may 2000.
9. M. Gordon and T. Melham. *Introduction to HOL*. Cambridge University Press, 1994.
10. D. E. Knuth. *The art of computer programming Fundamental algorithms*, volume 1. Addison-Wesley, 1969.
11. R. Laubenbacher. Algebraic models in Systems biology. In H. Anai and K. Horimoto, editors, *Algebraic Biology 2005 - Computer Algebra in Biology*, pages 33–40. Universal Academic press, Tokyo, Japan, 2005.
12. F. Mattern. Global quiescence detection based on credit distribution and re covery. *Information Processing Letters*, 30(4):195–200, Feb. 1989.
13. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
14. L. C. Paulson. Defining functions on equivalence classes. *ACM Transactions on Computational Logic*, 7(4):658–675, 2006.
15. L. Reyzin and N. Srivastava. On the longest path algorithm for reconstructing trees from distance matrices. *Information Processing Letters*, 101(1):98–100, january 2007.
16. F. Rosello and G. Valiente. An algebraic view of the relation between largest common subtrees and smallest common supertrees. *Theoretical Computer Science*, (362):33–53, 2006.
17. T. E. Vos and S. D. Swierstra. Inductive data types with negative occurrences in HOL. In *Workshop on Thirty Five years of Automath, Edinburgh, UK*, 2002.
18. M. Wenzel and F. Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29:389–411, 2002.
19. M. M. Wenzel. *Isar – a generic interpretative approach to readable proof documents*. Number 1690 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

Combinations of Theories and the Bernays-Schönfinkel-Ramsey Class

Pascal Fontaine

LORIA, Nancy University, France

Abstract. The Bernays-Schönfinkel-Ramsey (BSR) class of formulas is the class of formulas that, when written in prenex normal form, have an $\exists^*\forall^*$ quantifier prefix and do not contain any function symbols. This class is decidable. We show here that BSR theories can furthermore be combined with another disjoint decidable theory, so that we obtain a decision procedure for quantifier-free formulas in the combination of the BSR theory and another decidable theory.
The classical Nelson-Oppen combination scheme requires theories to be stably-infinite, ensuring that, if a model is found for both theories in the combination, models agree on cardinalities and a global model can be built. We show that combinations with BSR theories can be much more permissive, even though BSR theories are not always stably-infinite. We state that it is possible to describe exactly all the (finite or infinite) cardinalities of the models of a given BSR theory. For the other theory, it is thus only required to be able to decide if there exists a model of a given cardinality.
With this result, it is notably possible to use some set operators, operators on relations, orders — any operator that can be expressed by a set of BSR formulas — together with the usual objects of SMT solvers, notably integers, reals, uninterpreted symbols, enumerated types.

1 Introduction

Many techniques for the formal verification of information systems generate *verification conditions*, i.e. formulas encapsulating parts of the reasoning about the systems. The deduction tools validating these verification conditions should accept expressive languages, and should require a minimal amount of human interaction. Combination of theories is the method behind SMT-solvers (SMT for satisfiability modulo theories) to build decision procedures for very expressive languages, containing interpreted symbols from several decidable theories. Usually the theory embedded in the solvers is a static combination of linear arithmetic, uninterpreted symbols, list operators, bit-vectors,... For instance, it is possible to combine a decision procedure for integer linear arithmetic and a decision procedure for the empty theory (i.e. a decision procedure for equality and uninterpreted symbols) into a decision procedure to study formulas like

$$x \leq y \wedge y \leq x + f(x) \wedge P(h(x) - h(y)) \wedge \neg P(0) \wedge f(x) = 0.$$

The Bernays-Schönfinkel-Ramsey (BSR) class is a wide decidable class of formulas; any set of function-free universal formulas is indeed decidable. We consider here this class of formulas as a component in a combination of theories.

The classical Nelson-Oppen combination scheme [11, 16] requires every theory in the combination to be stably-infinite, i.e. every quantifier-free formula satisfiable in the theory should have a model with infinite cardinality. BSR theories are not, in general, stably-infinite: as an example, consider the BSR theory $\forall x \forall y (x = y)$ that only accepts models on a domain with one element. The classical combination result is not suitable in our case.

It has already been mentioned [17] that a BSR theory can be combined with a theory $\mathcal{T}$ provided

- if a set of ground literals L is $\mathcal{T}$-satisfiable, then the minimal cardinality of $\mathcal{T}$-models for L can be computed;
- $\mathcal{T}$ only has finite models.

We show here that this last strong requirement is not necessary; BSR theories can in fact be combined with any other decidable theory $\mathcal{T}$ (with or without infinite models, stably infinite or not), provided that, if a set L of ground literals is satisfiable in $\mathcal{T}$, it is possible to determine if there exists a $\mathcal{T}$-model of a given finite or infinite cardinality.

Motivations: the incentive for the procedure we present in Section 6 is double. First, the requirement we impose on the theory $\mathcal{T}$ is fulfilled by many decidable theories; using results in this paper it is possible to extend many decidable quantifier-languages (for instance, mixing uninterpreted symbols with linear arithmetic on reals and integers) with new interpreted predicates defined by a BSR theory. The BSR theory is not required to be stably-infinite. The other theory is not required to have only finite models.

The second motivation for such a general combination of theories is that the $\mathcal{T}$-satisfiability of quantifier-free formulas containing operators on sets, relations,... can be reduced to studying the satisfiability of sets of literals in the combinations of $\mathcal{T}$ and a BSR theory (see Sections 3 and 4). In Section 5, we show that there is a straightforward implementation of this method when $\mathcal{T}$ is the empty theory. Good results have been obtained with our prototype on translations of some problems from the SET domain of the TPTP library. When $\mathcal{T}$ is not the empty theory, we can fall back to the general decision procedure in Section 6. This decision procedure relies on the computation of model cardinalities of BSR theories. We show in Section 7 that it is possible to know exactly the cardinalities of BSR theories, and, in particular, we prove that it is possible to compute if a BSR theory has an infinite model or not.

For convenience, the results in this paper are presented in an *unsorted* framework, although most SMT-solvers work on a many-sorted logic (see for instance [5]). The results can easily be transferred to a many-sorted framework, at an expense of heavier notations.

2 Notations

A first-order language is a tuple $\mathcal{L} = \langle \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$ such that $\mathcal{V}$ is a enumerable set of variables, $\mathcal{F}$ and $\mathcal{P}$ are sets of functions and predicates (we refer to "symbols" for the union of $\mathcal{F}$ and $\mathcal{P}$). Every function and predicate symbol is assigned an arity. Nullary predicates are propositions, and nullary functions are constants. The set of terms on language $\mathcal{L}$ is defined in the usual way. A ground term is a term without variables. An atomic formula is either $t = t'$ where t and t' are terms, or a predicate symbol applied to the right number of terms. Formulas are built from atomic formulas, connectors ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$), and quantifiers ($\forall$, $\exists$). A formula with no free variable is closed. A theory is a set of closed formulas. Two theories are disjoint if no predicate (except the equality) or function symbol is interpreted in both theories.

An interpretation $\mathcal{I}$ for a first-order language assigns a set of elements D to the domain, a total function $\mathcal{I}[f]$ on D with appropriate arity to every function symbol f, a predicate $\mathcal{I}[p]$ on D with appropriate arity to every predicate symbol p, and an element $\mathcal{I}[x]$ to every variable x. By extension, an interpretation gives a value in D to every term, and a truth value to every formula. A model for a formula (or a theory) is an interpretation that makes the formula (resp. every formula in the theory) true. A formula is satisfiable if it has a model. It is unsatisfiable otherwise. A formula G is $\mathcal{T}$-satisfiable if it satisfiable in the theory $\mathcal{T}$, that is, if $\mathcal{T} \cup \{G\}$ is satisfiable. A $\mathcal{T}$-model of G is a model of $\mathcal{T} \cup \{G\}$. A formula G is $\mathcal{T}$-unsatisfiable if it has no $\mathcal{T}$-model.

The cardinality of an interpretation (or model) is the cardinality of the domain of this interpretation. The restriction of a predicate p on domain D to domain $D' \subseteq D$ is the predicate p' with domain D' such that p and p' have the same truth value for all arguments in D'.

A conjunctive (disjunctive) normal form is a conjunction of clauses, i.e. a conjunction of disjunctions of literals, (resp. a disjunction of conjunctions of literals). It is always possible to transform a quantifier-free formula into a logically equivalent conjunctive (disjunctive) normal form. A formula is universal if it is of the form $\forall x_1 \ldots \forall x_n . \varphi$ where φ is quantifier-free. A Skolem formula is a formula where all universal quantifiers appear with a positive polarity only, and all existential quantifiers appear with a negative polarity only. It is always possible to transform a given formula into an equisatisfiable Skolem formula, using Skolemization. We refer to [3] for Skolemization and conjunctive (disjunctive) normal form transformations.

3 From operators to BSR theories

Objects such as sets, relations, or arrays of bits can be viewed as predicates. For instance, sets can be unambiguously represented by their characteristic function

Equality	$\approx$	$\lambda p\, q.\ \forall x.\ p(x) \equiv q(x)$
membership	$\in$	$\lambda x\, p.\ p(x)$
$\emptyset$	$\emptyset$	$\lambda x.\ \bot$
Ω	Ω	$\lambda x.\ \top$
Enumerate	$\{a_1, \ldots a_n\}$	$\lambda x.\, (x = a_1 \vee \ldots x = a_n)$
Intersection	$\cap$	$\lambda p\, q.\ \lambda x.\ p(x) \wedge q(x)$
Union	$\cup$	$\lambda p\, q.\ \lambda x.\ p(x) \vee q(x)$
Difference	$\setminus$	$\lambda p\, q.\ \lambda x.\ p(x) \wedge \neg q(x)$
Subset	$\subseteq$	$\lambda p\, q.\ \forall x.\ p(x) \Rightarrow q(x)$

A. Sets

Equality	$\approx$	$\lambda p\, q.\ \forall x\, y\ .\ p(x,y) \equiv q(x,y)$
Transitive	Trans	$\lambda p.\ \forall x\, y\, z\ .\ [p(x,y) \wedge p(y,z)] \Rightarrow p(x,z)$
Symmetric	Sym	$\lambda p.\ \forall x\, y\ .\ p(x,y) \equiv p(y,x)$
Antisym.	ASym	$\lambda p.\ \forall x\, y\ .\ \neg p(x,y) \vee \neg p(y,x) \vee x = y$
Total	Tot	$\lambda p.\ \forall x\, y\ .\ p(x,y) \vee p(y,x)$
Reflexive	Refl	$\lambda p.\ \forall x\ p(x,x)$
Irreflexive	ARefl	$\lambda p.\ \forall x\ \neg p(x,x)$
Identity	Id	$\lambda x\, y.\ x = y$
Product	$\times$	$\lambda p\, q.\ \lambda x\, y.\ p(x) \wedge q(y)$

B. Relations

Equality	$\approx$	$\lambda p\, q.\ \forall x.\ p(x) \equiv q(x)$
Reading	read	$\lambda p\, i.\ p(i)$
Writing	write	$\lambda p\, i\, x.\ \lambda j.\ (j = i \Rightarrow x) \wedge (j \neq i \Rightarrow p(j))$

C. One-dimensional arrays of bits

Fig. 1. Operators

and operators on sets can be viewed as operators on predicates. In Figure 1, we give a few examples of set-like operators, operators on relations, operators to encode read and write operations on arrays of bits. In those examples, we assume p and q are predicates of appropriate arity and x, y, z are (first-order) variables. Notice that set-like operators can also be defined for relations; for instance, the intersection of relations is defined as $\lambda p\, q.\ \lambda x, y.\ p(x,y) \wedge q(x,y)$.

We consider formulas that are written in a first-order language augmented with the operators — defined as λ-terms given in Figure 1 — applied to the right number of objects of appropriate type.

Example 1. if A, B, C are unary predicates used to represent sets, a formula may contain $A \approx B \cap C$ which becomes, after substitution of $\cap$ and $\approx$ by their definition

$$[\lambda p\, q.\ \forall x.\ p(x) \equiv q(x)]\,(A, (\lambda p\, q.\ \lambda x.\ p(x) \wedge q(x))(B, C))\,.$$

After β-reduction, this becomes

$$\forall x.A(x) \equiv [B(x) \wedge C(x)]\,. \tag{1}$$

In general, the formulas obtained after elimination of operators mentioned in this section are first-order, but may contain quantifiers. Those quantifiers

come directly from the λ-terms; for instance the quantifier in (1) comes from the definition of $\approx$. It is easily shown however, that, if the original formula (with operators on sets, relations...) does not contain quantifiers, the resulting first-order formula is a Boolean combination of (atoms and) formulas of the form $\forall x_1 \dots x_n \, \varphi$ where φ is quantifier-free. Furthermore, quantified variables are used only as arguments of predicates, that is, no function has a quantified variable as an argument.

4 From FOL formulas to combination of theories

The formulas obtained in the previous section are Boolean combinations of quantified formulas. In this section we describe the process to reduce the $\mathcal{T}$-satisfiability problem of these quantified formulas, to the satisfiability problem for sets of literals in the union of two theories: $\mathcal{T}$ and a disjoint Bernays-Schönfinkel-Ramsey theory $L_\forall$. For the rest of the paper, we only impose one restriction on the decidable theory $\mathcal{T}$: if a set of literals is $\mathcal{T}$-satisfiable, it is possible to compute if there exists a model of a given cardinality. We also assume that all predicates occurring in operators from Figure 1 are uninterpreted for $\mathcal{T}$.

The form of the formulas issued in the previous section is such that a structural Skolemization (see for instance [3]) will never introduce Skolem functions, but only Skolem constants. We assume that the formula is Skolemized, using such a structural Skolemization. The obtained formula is a Boolean combination of universal formulas (and atoms), the universal formulas appearing with a positive polarity only.

The usual technique used in SMT-solvers to check the satisfiability of a quantifier-free formula in a theory $\mathcal{T}$ is a (loose or tight) cooperation of a Boolean satisfiability checker, and a procedure to check the satisfiability of literals within $\mathcal{T}$. This cooperation splits the problem into two parts: first, pure Boolean model searches, and second, $\mathcal{T}$-satisfiability checks for the corresponding *conjunctive sets of literals*. For simplicity, we consider here that the split is realized by converting the formula to disjunctive normal form. The formula is satisfiable if and only if at least one conjunction of literals in the disjunctive normal form is satisfiable. Now assume Ψ is the obtained formula after Skolemization. The formula is transformed into disjunctive normal form, the quantified parts being left unchanged. Since the formula has been Skolemized, the remaining (universal) quantifiers all appear with a positive polarity. Each conjunction of literals in the disjunctive normal form only contains:

- first-order literals;
- formulas of the form $\forall x_1 \dots x_n \, \varphi$, where φ is a quantifier-free formula, such that no $x_1 \dots x_n$ is used within a function;

Example 2. Suppose we want to study the satisfiability of the formula:

$$a = b \wedge f(a) \in A \wedge f(b) \notin C \wedge [f(b) \notin A \vee A \cup B \approx C \cap D]\,.$$

Substituting operators $\in$, $\cup$, $\cap$, $\approx$ by their definition and applying β-reduction, one obtains

$$a = b \wedge A(f(a)) \wedge \neg C(f(b)) \wedge \Big[\neg A(f(b)) \vee \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\Big]$$

Structural Skolemization leaves this last formula unchanged, since the sole universal quantifier appears with a positive polarity. The corresponding disjunctive normal form contains the two conjunctive sets of literals:

$$\{a = b, A(f(a)), \neg C(f(b)), \neg A(f(b))\} \tag{2}$$
$$\{a = b, A(f(a)), \neg C(f(b)), \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\} \tag{3}$$

The first set can easily be identified as being unsatisfiable. The second set only contains first-order (quantifier-free) literals and formulas of the form $\forall x_1 \ldots x_n\, \varphi$, where φ is a quantifier-free formula, such that no $x_1 \ldots x_n$ is used within a function.

In order to study the satisfiability of a set of literals in the combination of disjoint theories, one usually first computes a *separation* of the set of formulas along the languages in the disjoint theories.[1] Each part of the separation contains only the symbols from one theory in the combination; the only shared symbols are equality and variables. We apply the same technique to separate predicates that appear in quantified formulas from the rest of the symbols. For instance, the set (3) is logically equivalent (in whatever theory) to the union of the sets

$$L_g = \{a = b, y = f(a), z = f(b)\},$$
$$L_\forall = \{A(y), \neg C(z), \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\},$$

where y, z are introduced variables. In general, a separation can be built using the following method.

Algorithm: Initially, L is a set containing literals and universal formulas, and no quantified variable as argument of a function. The separation algorithm builds two sets L_g (g for ground), and $L_\forall$ (for quantified formulas and related predicates):

- for every uninterpreted predicate p that occurs in a quantified formula in L, for every occurrence $p(t_1, \ldots t_n)$ of this predicate (in a quantified formula or not), for every subterm t_i that is not a variable (shared or not), introduce a new shared variable x, add $x = t_i$ to L_g, and replace t_i by x in L. Handle similarly all occurrences of the form $t_1 = t_2$ in a quantified formula in L. This is possible since no quantified variable is used as an argument of a function;

[1] See for instance [6] for a formal presentation of the separation technique.

- for every uninterpreted predicate p that belongs to a quantified formula in L, add every literal $p(t_1, \ldots t_n)$ (or $\neg p(t_1, \ldots t_n)$) from L to $L_\forall$. The previous two steps ensure that here, $t_1, \ldots t_n$ are variables;
- add every quantified formula from L to $L_\forall$. Those formulas are universal formulas, and the previous steps ensures that they are function-free.
- finally, every literal in L that does not belong to $L_\forall$ is added to L_g.

In this algorithm:

- the computed $L_\forall$ is a set of function-free universal formulas, i.e. a BSR theory;
- the initial L is $\mathcal{T}$-satisfiable if and only if $L_g \cup L_\forall$ is also $\mathcal{T}$-satisfiable;
- the shared terms in L_g and $L_\forall$ are all variables.

To summarize, studying the $\mathcal{T}$-satisfiability of a given formula with operators as described in Figure 1 can be reduced to studying the $\mathcal{T}$-satisfiability of sets $L_g \cup L_\forall$. Another point of view is to study the satisfiability of the sets of literals L_g, in the combination of the disjoint theories $\mathcal{T}$ and $L_\forall$. In the following sections, we show that this problem is decidable, for any decidable theory $\mathcal{T}$, as long as it is possible to determine if L_g accepts a $\mathcal{T}$-model of a given cardinality.

5 Combining a BSR theory with the empty theory

The method in the previous section leads to checking the satisfiability of a set of literals L_g in the union of $\mathcal{T}$ and a BSR theory $L_\forall$ (L_g and $L_\forall$ share only variables). We assume in this section that $\mathcal{T}$ is the empty theory. That is, every function and predicate in L_g is left uninterpreted.

The classical Nelson-Oppen combination scheme cannot be used, since the theory $L_\forall$ is not necessarily stably-infinite, that is, it may be satisfiable only in finite models. For instance, if the original formula uses the "Enumerate" operator, the resulting sets of formulas may contain a formula of the form

$$\forall x.\ x = a \vee x = b \vee x = c$$

which would make $L_\forall$ non stably-infinite; the formula accepts models of cardinality at most three. However we known that the empty theory can be combined with any theory, not only stably-infinite ones [7, 17]. We now recall the combination algorithm.

Given a partition $\mathcal{P}$ of a set of terms, an *arrangement* induced by $\mathcal{P}$ is the set of all equalities between any two terms in the same class of $\mathcal{P}$, and all disequalities between any two terms in different classes in $\mathcal{P}$. For instance, the arrangement induced by $\{\{x_1, x_2\}, \{x_3\}\}$ is $\{x_1 = x_2, x_1 \neq x_3, x_2 \neq x_3\}$. Assume we have to study the satisfiability of the separation $L_1 \cup L_2$ in the combination of the stably-infinite disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$, where L_i ($i \in \{1, 2\}$) only contains symbols from $\mathcal{T}_i$ and variables. The classical result for combining stably-infinite

disjoint theories states that $L_1 \cup L_2$ is satisfiable in the combination of $\mathcal{T}_1$ and $\mathcal{T}_2$ if and only if there exists an arrangement $\mathcal{A}$ on the set of shared variables between L_1 and L_2, such that $L_i \cup \mathcal{A}$ is $\mathcal{T}_i$-satisfiable, for $i = 1$ and $i = 2$. The procedure terminates, since the set of shared variables is finite, as well as the set of arrangements. In the case where $\mathcal{T}_1$ is the empty theory, and $\mathcal{T}_2$ is any theory (not necessarily stably-infinite), the result still holds [7], but the arrangement has to be considered on a larger set of terms; the arrangement has to be considered on all terms and variables in $L_1 \cup L_2$.[2]

Applied to our present case, L_g is satisfiable in the combination of the empty theory and $L_\forall$, if and only if there exists an arrangement $\mathcal{A}$ of all ground terms and free variables in L_g such that $\mathcal{A} \cup L_g$ and $\mathcal{A} \cup L_\forall$ are both satisfiable.

Example 3. As an application, consider again the previous example:

$$\{a = b, A(f(a)), \neg C(f(b)), \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\}$$

one has to study the satisfiability of the unions of the sets

$$L_g = \{a = b, y = f(a), z = f(b)\},$$
$$L_\forall = \{A(y), \neg C(z), \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\},$$

which is equivalent to study the satisfiability of L_g in the combination of the empty theory and $L_\forall$. The combination framework then ensures that $L_g \cup L_\forall$ is satisfiable if and only if there exists an arrangement $\mathcal{A}$ of $\{a, y, z\}$ (the other terms being necessarily equal to one in this set) such that $\mathcal{A} \cup L_g$ and $\mathcal{A} \cup L_\forall$ are satisfiable. There are well known decision procedures for both satisfiability problems.

5.1 Towards an implementation

The set of formulas $\mathcal{A} \cup L_\forall$ is also a BSR theory. It is satisfiable if and only if $\mathcal{A} \cup L_{\text{inst}}$ is, where L_{inst} is a set of well-chosen instances of formulas in $L_\forall$. This leads to the following result:

Theorem 1. *Given a theory $\mathcal{T}$, a set of literals L_g, a BSR theory $L_\forall$ such that L_g and $L_\forall$ only share variables, then $L_g \cup L_\forall$ is satisfiable (in the empty theory) if and only if $L_g \cup L_{\text{inst}}$ is, where L_{inst} is a set of instances of $L_\forall$: for every formula $\forall x_1 \ldots \forall x_n \varphi(x_1, \ldots x_n)$ in $L_\forall$ ($\varphi(x_1, \ldots x_n)$ being quantifier-free), and terms or free variables $t_1, \ldots t_n$ in $L_g \cup L_\forall$, L_{inst} contains the formula $\varphi(t_1, \ldots t_n)$.*

Example 4. Applying this result on the previous example:

$$L_g = \{a = b, y = f(a), z = f(b)\},$$
$$L_\forall = \{A(y), \neg C(z), \forall x.\ [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]\},$$

[2] Another approach considers arrangements on the set of shared variables only, computes minimal cardinalities of models for the empty theory, and ensures there is a model with a larger (or equal) cardinality for the other theory [17].

gives the formula

$$\{a = b, y = f(a), z = f(b), A(y), \neg C(z), \varphi(y), \varphi(z), \varphi(a)\}$$

where $\varphi(x) = [A(x) \vee B(x)] \equiv [C(x) \wedge D(x)]$.

Deciding formulas that only contain operators like those in Figure 1, can be done easily using the capabilities implemented in any SMT-solver (for instance [1, 12]): the ability to deal with a Boolean combination of terms that only contain uninterpreted symbols. This language being handled very efficiently by modern solvers, the tools do cope well even if the number of generated instances is large. A naïve implementation can be realized by doing β-reduction, Skolemization, and instantiation as a preprocess, feeding a Boolean combination of terms with only uninterpreted symbols to the SMT-solver.

A working prototype has been implemented. We ran this prototype on the translation of problems SET008+3p, SET064+1p, SET143+3p, SET171+3p, SET580+3p, SET601+3p SET606+3p, SET623+3p, and SET609+3p from the TPTP library [15]. Unsurprisingly, these are all solved in a few milliseconds. We should however mention that it is not really relevant to compare these performances with ones of the FOL provers, since the set theories in which the problems are checked for satisfiability are not the same for both approaches. For instance, our approach implicitly assume sets cannot contain other sets, whereas no such assumption is made in the TPTP problems.

6 Combining BSR theories with arbitrary decidable theories

In the previous sections we considered formulas that contain uninterpreted symbols, as well as other symbols such as set and relation operators. We show in this section that there is a decision procedure for formulas that contain such set and relation operators and *interpreted* symbols from an arbitrary decidable theory $\mathcal{T}$, provided (1) there is a decision procedure for the arbitrary theory that is able to state if there is a model of a given cardinality (2) set and relation operators are applied on uninterpreted symbols only. We have to study the satisfiability of the set of ground literals L_g in the combination of the disjoint theories $\mathcal{T}$ and $L_\forall$, where L_g only contains symbols from $\mathcal{T}$ and variables.

Theorem 2. *Given a theory $\mathcal{T}$, a set of literals L_g, and a BSR theory $L_\forall$ such that L_g and $L_\forall$ only share variables, then L_g is satisfiable in $\mathcal{T} \cup L_\forall$ if and only if there exists an arrangement $\mathcal{A}$ of variables shared by L_g and $L_\forall$ such that $\mathcal{A} \cup L_g$ has a $\mathcal{T}$-model, and $\mathcal{A} \cup L_\forall$ has a model, both models having the same cardinality.*

This theorem is an adaptation of the general result to combine non-stably-infinite theories (see for instance [17]).

For theoretic discussions, the process of combining stably-infinite theories usually implies guessing an arrangement on a set of variables. In practice, it is equivalent, and more efficient that decision procedures exchange disjunctions of equalities (see for instance [6] for a presentation of this equivalence). We can imagine a similar treatment here for cardinalities. The decision procedures could negotiate the size of the models by exchanging constraints. For simplicity, a naïve decision procedure for the combination can be:

- build L_g and $L_\forall$ according to the method presented in section 4. Both sets only share variables, and no symbol in $L_\forall$ is interpreted by $\mathcal{T}$;
- guess an arrangement $\mathcal{A}$ on shared variables. Notice there is only a finite number of such arrangements: this guess can thus be replaced by a terminating loop;
- if the code on Fig. 2 returns "succeed" for $\mathcal{A}$, the $L_g \cup L_\forall$ is $\mathcal{T}$-satisfiable;
- If every arrangement returns "fail" for the code on Fig. 2, $L_g \cup L_\forall$ is $\mathcal{T}$-unsatisfiable.

The procedure concludes to $\mathcal{T}$-satisfiability if and only if a model is found that meets the conditions of Theorem 2. It remains to check that that every step of the code on Fig. 2 is tractable. The test on line 1 is decidable since the $\mathcal{T}$-satisfiability problem for sets of literals is decidable, and since $\mathcal{A} \cup L_\forall$ is a BSR theory (decidable fragment). The results in the following section state that it is possible to determine exactly what cardinalities are accepted for models of any BSR theory, and in particular for $\mathcal{A} \cup L_\forall$: the tests on lines 3 and 7 are decidable, and it is possible to enumerate (within finite time) the cardinalities in line 4. The tests on lines 5, 8 and 13 are possible thanks to the condition on theory $\mathcal{T}$. For the test on line 15, checking if $\mathcal{A} \cup L_g$ has a $\mathcal{T}$-model with cardinality greater or equal to k is simply reduced to checking the $\mathcal{T}$-satisfiability of $\mathcal{A} \cup L_g \cup \bigcup_{1 \leq i \leq k} \bigcup_{i < j \leq k} \{a_i \neq a_j\}$ where $a_1, \ldots a_k$ are fresh constants.

7 Cardinalities of BSR theories

The previous section states that to combine a BSR theory with another theory is mainly a matter of getting the cardinalities of the models of the BSR theory. We give now a necessary and sufficient criteria to determine if there is an infinite model for such a theory, and if not, what are the finite cardinalities for which there exists a model. For simplicity, we assume here that we have a BSR theory, with no free variables, but only constants. If this requirement is not met, one can transform the problem into an equivalent one by replacing free variables with fresh constants.

```
1:  if A ∪ L_g is T-unsatisfiable or
        A ∪ L_∀ is unsatisfiable then
2:    return fail
3:  if A ∪ L_∀ only has finite models then
4:    for each cardinality j of models of A ∪ L_∀ do
5:      if A ∪ L_g has a T-model with cardinality j then
6:        return succeed
7:  if A ∪ L_∀ has an infinite model then
8:    if A ∪ L_g has an infinite T-model then
9:      return succeed
10:   else
11:     k := the number of free variables and constants in A ∪ L_∀
12:     for each j < k do
13:       if A ∪ L_g has a T-model with cardinality j and
              A ∪ L_∀ has a model with cardinality j then
14:         return succeed
15:     if A ∪ L_g has a T-model with cardinality ≥ k then
16:       return succeed
17: return fail
```

Fig. 2. Inspecting arrangement $\mathcal{A}$

Given a BSR theory $\mathcal{T}$ using k constants, we first recall the simple result that states that, if $\mathcal{T}$ has a model of (finite or infinite) cardinality i greater than k, then it has a model for every cardinality j such that $k \leq j \leq i$. We then show that there is a number k' ($> k$), computable from $\mathcal{T}$, such that, if there is a model of cardinality greater or equal to k', then there is an infinite model. Altogether, this implies that $\mathcal{T}$ either has a model for every cardinality greater or equal to k (example in Figure 3), or there exists a j smaller than the known, finite, number k', such that $\mathcal{T}$ has a model of every cardinality between k and j, and no model of cardinality greater than j (example in Figure 3). Alternatively, one can also decide if a theory with n distinct quantified variables has an infinite model by checking if it has a n-repetitive model (see subsection 7.2).

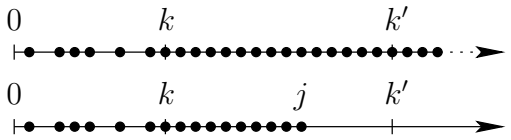


Fig. 3. Theories with infinite (above) and finite cardinalities.
A dot means there is a model with given cardinality.

7.1 BSR theories and finite models

Intuitively, the following theorem states that, given a model for a BSR theory, the elements in the domain that are not assigned to ground terms (i.e. the constants) can be eliminated, keeping it a model:

Theorem 3. *Given a model $\mathcal{M}$ for a BSR theory $\mathcal{T}$ with domain D, then $\mathcal{M}'$ such that*

- *the domain is a non-empty set $D' \subseteq D$, with $\mathcal{M}[a] \in D'$ for every constant a in $\mathcal{T}$;*
- *for every predicate p, $\mathcal{M}'[p]$ is the restriction of $\mathcal{M}[p]$ to the domain D';*

is also a model for $\mathcal{T}$

Proof. Since $\mathcal{M}$ is a model for $\mathcal{T}$, for each closed formula $\forall x_1 \dots x_n.\varphi$ in $\mathcal{T}$ (where φ is function and quantifier-free), and for all $d_1,\dots d_n \in D' \subseteq D$, $\mathcal{M}_{x_1/d_1,\dots x_n/d_n}$ is a model for φ. This also means that, for all $d_1,\dots d_n \in D'$, ${\mathcal{M}'}_{x_1/d_1,\dots x_n/d_n}$ is a model for φ, and finally that $\mathcal{M}'$ is a model for $\forall x_1 \dots x_n.\varphi$. □

Corollary 1. *Assume k is the number of constants in a BSR theory $\mathcal{T}$, or 1 if $\mathcal{T}$ has no constant. If there is a $\mathcal{T}$-model of cardinality j, there is a finite $\mathcal{T}$-model with any cardinality i with $k \leq i \leq j$. If there is an infinite $\mathcal{T}$-model, there is a $\mathcal{T}$-model with any cardinality i with $k \leq i$.*

7.2 BSR theories and infinite models

We known that a BSR theory either has models for every finite and infinite cardinality greater than k, or it only has models of finite cardinalities all smaller than a number k'. What is missing is a way to decide if one theory has an infinite model or not. If it has no infinite model, the number k' can be computed (naïvely) by checking all finite models by increasing cardinalities until k' is found.

The following definition expresses some symmetry properties of models. We later show that the existence of an infinite model is equivalent to the existence of a finite model having such symmetry properties.

Definition 1. *Let $\mathcal{M}$ be an interpretation on domain D for a BSR theory $\mathcal{T}$. Let $A = \{\mathcal{M}[a] \,|\, a \text{ is a constant in } \mathcal{T}\}$ and $B = D \setminus A$. $\mathcal{M}$ is* n-repetitive *if $|B| \geq n$ and if there exists a total order $\prec$ on elements in B such that*

- *for every $m \leq n$;*
- *for every two strictly increasing (with respect to $\prec$) series $e_1, \dots e_m$ and $e'_1, \dots e'_m$ of elements in B;*
- *for every k-ary predicate symbol p used in $\mathcal{T}$;*
- *for every $d_1, \dots d_k \in A \cup \{e_1, \dots e_m\}$;*

$\mathcal{M}[p](d_1, \ldots d_k) = \mathcal{M}[p](d'_1, \ldots d'_k)$, *with* $d'_i = e'_j$ *if* $d_i = e_j$ *for some* j, $d'_i = d_i$ *otherwise. By extension, a theory is* n*-repetitive if it has a* n*-repetitive model.*

Observe that, thanks to Theorem 3, a theory is n-repetitive if it has a n-repetitive model $\mathcal{M}$ such that $|B| = n$, in the previous definition.

Example 5. Assume $\mathcal{T}$ is a theory with constants $a_1, \ldots a_{n_0}$, unary predicates $p^1_1, \ldots p^1_{n_1}$, binary predicates $p^2_1, \ldots p^2_{n_2}$.

$\mathcal{T}$ is 1-repetitive, if and only if $\mathcal{T} \cup R_1(b)$ is satisfiable, with

$$R_1(b) =_{\text{def}} \{b \neq a_1, \ldots b \neq a_{n_0}\}.$$

In other words, a theory $\mathcal{T}$ is 1-repetitive if it accepts a model with an element in the domain that is not assigned to a constant used in $\mathcal{T}$.

$\mathcal{T}$ is 2-repetitive, if and only if

$$\mathcal{T} \cup \bigcup_{i \in \{0,1\}} R_1(b_i) \ \cup \ R_2(b_0, b_1)$$

is satisfiable, with

$$\begin{aligned} R_2(b_0, b_1) =_{\text{def}} & \{b_0 \neq b_1\} \\ & \cup \{p^1_i(b_0) \equiv p^1_i(b_1) \mid i \in [1..n_1]\} \\ & \cup \{p^2_i(b_0, b_0) \equiv p^2_i(b_1, b_1) \mid i \in [1..n_2]\} \\ & \cup \{p^2_i(a_j, b_0) \equiv p^2_i(a_j, b_1) \mid i \in [1..n_2],\ j \in [1..n_0]\} \\ & \cup \{p^2_i(b_0, a_j) \equiv p^2_i(b_1, a_j) \mid i \in [1..n_2],\ j \in [1..n_0]\} \end{aligned}$$

$\mathcal{T}$ is 3-repetitive, if and only if

$$\mathcal{T} \cup \bigcup_{i \in \{0,1,2\}} R_1(b_i) \ \cup \bigcup_{\substack{i < j \\ i,j \in \{0,1,2\}}} R_2(b_i, b_j) \ \cup \ R_3(b_0, b_1, b_2)$$

is satisfiable, with

$$\begin{aligned} R_3(b_0, b_1, b_2) =_{\text{def}} & \{p^2_i(b_0, b_1) \equiv p^2_i(b_1, b_2) \equiv p^2_i(b_0, b_2) \mid i \in [1..n_2]\} \\ & \cup \{p^2_i(b_1, b_0) \equiv p^2_i(b_2, b_1) \equiv p^2_i(b_2, b_0) \mid i \in [1..n_2]\} \end{aligned}$$

Theorem 4. *If a BSR theory* $\mathcal{T}$ *with* n *distinct quantified variables has a* n*-repetitive model with cardinality* k*, then it has (*n*-repetitive) models with any (finite or infinite) cardinality* $k' \geq k$*.*

Proof. Assume $\mathcal{M}$ is a n-repetitive $\mathcal{T}$-model of cardinality k on domain D. Let A be $\{\mathcal{M}[a] \mid a \text{ is a constant in } \mathcal{T}\}$, and $B = D \setminus A$ ($|B| = k - |A| \geq n$). Assume also that $\prec$ is the total order on B mentioned in Definition 1. Choose a strictly increasing (with respect to $\prec$) series of n distinct elements $e_1, \ldots e_n \in B$.

Let E be such that $E \cap D = \emptyset$, and $|D \cup E| = k'$. We define an interpretation $\mathcal{M}'$ on domain $D' = D \cup E$. The total order $\prec$ on B is extended to $B \cup E$. We then require that $\mathcal{M}'[a] = \mathcal{M}[a]$ for every constant a in $\mathcal{T}$, and that, for every m-ary predicate p in $\mathcal{T}$ and every $d'_1, \ldots d'_m \in D'$:

- if $|\{d'_1, \dots d'_m\} \setminus A| > n$, $\mathcal{M}'[p(d'_1, \dots d'_m)]$ does not matter;
- if $\{d'_1, \dots d'_m\} \subseteq D$, $\mathcal{M}'[p(d'_1, \dots d'_m)] = \mathcal{M}[p(d'_1, \dots d'_m)]$;
- otherwise, let $e'_1, \dots e'_n$ be a strictly increasing series including all elements in $\{d'_1, \dots d'_m\} \setminus A$. $\mathcal{M}'[p](d'_1, \dots d'_k) = \mathcal{M}[p](d_1, \dots d_k)$, with $d_i = e_j$ if $d'_i = e'_j$ for some j, $d_i = d'_i$ otherwise.

By construction, $\mathcal{M}'$ is n-repetitive.

Every formula in $\mathcal{T}$ is of the form $\forall x_1 \dots x_m.\varphi(x_1, \dots x_m)$, with $m \leq n$. For all elements $d'_1 \dots d'_m \in D'$, the truth value for $\mathcal{M}'_{x_1/d'_1, \dots x_m/d'_m}[\varphi(x_1, \dots x_m)]$ is $\mathcal{M}_{x_1/d'_1, \dots x_m/d'_m}[\varphi(x_1, \dots x_m)]$ (i.e. true), if $\{d'_1, \dots d'_m\} \subseteq D$. Otherwise, assume $e'_1, \dots e'_n$ is a strictly increasing series including all elements in $\{d'_1, \dots d'_m\} \setminus A$. Since the model $\mathcal{M}'$ is n-repetitive, then $\mathcal{M}'_{x_1/d'_1, \dots x_m/d'_m}[\varphi(x_1, \dots x_m)]$ is equal to $\mathcal{M}_{x_1/d_1, \dots x_m/d_m}[\varphi(x_1, \dots x_m)]$ (i.e. true) where $d_i = e_j$ if $d'_i = e'_j$ for some j, $d_i = d'_i$ otherwise. Finally, $\mathcal{M}'$ is a model of $\forall x_1 \dots x_n.\varphi(x_1, \dots x_m)$. □

Theorem 5. *If a BSR theory $\mathcal{T}$ has a model with a cardinality greater than a number k' computable from the theory, then it has a n-repetitive model on domain $D = A \cup B$, where $A = \{\mathcal{M}[a] \mid a \text{ is a constant in } \mathcal{T}\}$, $A \cap B = \emptyset$ and $|B| = n$.*

Proof. Assume $\mathcal{T}$ has a finite model $\mathcal{M}'$ on domain D'. We define the sets $A = \{\mathcal{M}'[a] \mid a \text{ is a constant in } \mathcal{T}\}$ and $B' = D' \setminus A$. Choose an order $\prec$ on B'. We now compute the size of B' so that there exists a n-repetitive model. A suitable k' can then be computed from $|B'|$.

Given two ordered (with respect to $\prec$) series $e_1, \dots e_m$ and $e'_1, \dots e'_m$ of elements in B', we will say that the configurations for $e_1, \dots e_m$ and $e'_1, \dots e'_m$ are the same if for every k-ary predicate p, and for every $d_1, \dots d_k \in A \cup \{e_1, \dots e_m\}$, $\mathcal{M}'[p](d_1, \dots d_k) = \mathcal{M}'[p](d'_1, \dots d'_k)$, with $d'_i = e'_j$ if $d_i = e_j$ for some j, $d'_i = d_i$ otherwise. Notice that there are only a finite number of different configurations for m elements in B': more precisely a configuration is made of at most $b = \sum_p [m + |A|]^{\text{arity}(p)}$ Boolean values, where the sum ranges on all predicates in the theory. Thus the number of different configurations is bounded by $C = 2^b$.

Understanding colors as being configurations, one can use Theorem 6 (in Appendix A) to state that, if $|B'| > f(n, N, C)$, then there exists a model of cardinality $|A| + N$ for $\mathcal{T}$ with the same configuration for any m ordered distinct elements. Recursively applying this procedure for every $m \in [1..n]$, it is possible to compute the cardinality k' of the original model so that there exists a n-repetitive model with the suitable cardinality. □

From both previous theorems:

Corollary 2. *Given a BSR theory $\mathcal{T}$ using n distinct quantified variables. $\mathcal{T}$ has an infinite model if and only if it has a n-repetitive model.*

Checking if a BSR theory $\mathcal{T}$ has an n-repetitive model is reduced to checking the satisfiability of another BSR theory $\mathcal{T}'$, basically, $\mathcal{T}$ augmented with

some quantifier-free formulas. For formulas containing operators discussed in Section 3, we have $n \leq 3$, and predicates have an arity of at most 2: $\mathcal{T}'$ is given in Example 5. If $\mathcal{T}$ does not have an infinite model, then there is a maximum cardinality j for its models. The theory accepts a model for every cardinality between the number k of constants in $\mathcal{T}$ and j. This number j is bounded by a computable number k'. Unluckily, we currently lack an efficient (if there exists) way to compute this j. A naïve process to determine this number is to try every cardinality greater than k; the process will eventually terminate. Finally notice that this inefficient process is not necessary when combining a BSR theory with theories that only have infinite models.

8 Conclusions

In Section 3, we noticed that the use of some operators to encode sets, properties on relation,... would imply to have to verify the $\mathcal{T}$-satisfiability of FOL formulas with quantifiers. It was also shown that this satisfiability problem can be reduced to the satisfiability problem of literals in the combination of the theory $\mathcal{T}$ and another decidable theory, precisely a set of Bernays-Schönfinkel-Ramsey formulas.

Combining a BSR theory with the empty theory is possible, and this is the basis to build a decision procedure for formulas that contain uninterpreted functions and predicates, some operators on sets, relations,... A prototype has been built, and the first results are promising. When formulas containing operators from Section 3 have to be studied in some decidable (non-empty) theory $\mathcal{T}$, the combination process with the BSR theory is more complicated. The method presented in Section 6 is not in itself a practical procedure: its complexity prevents a direct application. However we believe that it can be the basis for a useful tool, with implementation-oriented improvements and proper heuristics.

We mainly target the B [2] and TLA+ [9] formal methods. Those language heavily rely on some set theories, and we believe that the results in this paper can help automating the proof of some parts of the verification conditions, which often mix arithmetic symbols, uninterpreted functions, and set operators. Verification conditions generated within those formal methods are usually small, within reach of a decision procedure even if it is inefficient. For verification conditions that are not fully within the language of the decision procedure, we built a certified (through proof reconstruction [8, 10]) cooperation between a proof assistant and the automated tool. At the present time, this cooperation can be used to delegate the proof of theorems from Isabelle to our prototype implementation (see Section 5), and have the proofs rechecked by the kernel of Isabelle, ensuring consistency of the whole cooperation of both tools.

A direction for further research is to investigate how to use the knowledge and engineering embedded in state-of-the-art first order provers (for instance [14, 13, 4]) to handle the BSR theories within a combination of decision procedures.

Acknowledgments: I am grateful to Yves Guiraud, Yuri Gurevich, Stephan Merz, Silvio Ranise, Christophe Ringeissen, and Duc-Khanh Tran for the interesting discussions on this subject.

References

1. Yices: An SMT solver. Available on http://yices.csl.sri.com/.
2. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. M. Baaz, U. Egly, and A. Leitsch. Normal form transformations. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 273–333. Elsevier Science B.V., 2001.
4. P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. In S. Schulz, G. Sutcliffe, and T. Tammet, editors, *Special Issue of the International Journal of Artificial Intelligence Tools (IJAIT)*, volume 15 of *International Journal of Artificial Intelligence Tools*, 2005.
5. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Orlando, Florida, 1972.
6. P. Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, Sept. 2004.
7. P. Fontaine and E. P. Gribomont. Combining non-stably infinite, non-first order theories. In W. Ahrendt, P. Baumgartner, H. de Nivelle, S. Ranise, and C. Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 37–51, July 2005.
8. P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006.
9. L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.
10. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-light and CVC lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006.
11. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
12. R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In G. Sutcliffe and A. Voronkov, editors, *12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.
13. A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2):91–110, 2002.
14. S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, number 2083 in Lecture Notes in Artificial Intelligence, pages 370–375. Springer, 2001.
15. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
16. C. Tinelli. Cooperation of background reasoners in theory reasoning by residue sharing. *Journal of Automated Reasoning*, 30(1):1–31, Jan. 2003.
17. C. Tinelli and C. G. Zarba. Combining non-stably infinite theories. In I. Dahn and L. Vigneron, editors, *First Order Theorem Proving*, volume 86.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

A n-monochromatic theorem

We define a *n-subset* of S to be a subset of n elements of S. An *n-overgraph* in S is a set of n-subsets of S. In particular, a 2-overgraph is a (undirected) graph. The *complete* n-overgraph of S is the set of all n-subsets of S, and its *size* is the cardinality of S. A n-overgraph G is *colored* with a set of colors C if there is a coloring function that assigns an element in C to every n-subset in G. In particular, a colored 2-overgraph (that is, a colored graph), is a graph where all edges are assigned a color. A colored n-overgraph is *monochromatic* if the coloring function assigns the same color to every n-subset. A colored n-overgraph A is a *sub-overgraph* of a colored n-overgraph B if each n-subset $S \in A$ belongs to B, and if the color associated to S is the same in both colored n-overgraphs.

Theorem 6. *There exists a computable function f such that, for every set of colors C, for every $n, N \in \mathbb{N}$, and every complete n-overgraph G colored with C, if the size of G is greater or equal to $f(n, N, |C|)$, there exists a complete monochromatic n-sub-overgraph of G of size greater or equal to N.*

Proof. We proceed by induction on n and the size of C.

Notice first that $f(n, N, 1) = N$ for every n, since a n-overgraph is colored with a unique color is monochromatic. Also, $f(1, N, 2) = 2N$, since a set of $2N$ elements that have one color in a pair $\{b, w\}$ contains at least N elements of the same color.

We now consider $f(n, N, 2)$. Assume G is a complete n-overgraph in S colored by c using colors in $\{b, w\}$. We build the series S_i and e_i such that

- $S_0 = S$
- e_i is any element in S_i
- To build S_{i+1}, we consider the complete $(n-1)$-overgraph in $S_i \setminus \{e_i\}$, colored by c_{e_i}, where c_{e_i} assigns to each $(n-1)$-subset A of $S_i \setminus \{e_i\}$ the color given by c to the n-subset $A \cup \{e_i\}$. Using the induction hypothesis, if $|S_i| \geq f(n-1, x, 2)$, there is a subset $S_{i+1} \subseteq S_i \setminus \{e_i\}$ such that $|S_{i+1}| \geq x$ and the complete $(n-1)$-overgraph of S_{i+1} colored by c_{e_i} is monochromatic.

Let B be the set of e_i such that the $(n-1)$-overgraph in S_{i+1} is colored by b, and W be the set of e_i such that the $(n-1)$-overgraph in S_{i+1} is colored by w. The n-overgraphs in B and W colored by c are monochromatic. To have $|B| \geq N$ or $|W| \geq N$ it is sufficient that $|S_{2N}| = n - 1$. Defining function g to be such that $g(*) = f(n-1, *, 2)$, it is sufficient to have $|S_0| \geq g^N(N)$

It remains to define $f(n, N, |C|)$ when $|C| > 2$. Assume G is a n-overgraph in S colored by c using colors in $C \cup k$ ($k \notin C$). We now consider all colors in C as one sole color; using the induction hypothesis, if $|S| \geq f(n, N', 2)$ then there exists $S' \subseteq S$ such that $|S'| \geq N'$ and the complete n-overgraph of S' colored by c only uses colors in C, or is monochromatic with color k. Any way,

if one chooses N' as being greater than $f(n, N, |C|)$, there exists a subset S'' of S' such that $|S''| \geq N$ and the complete n-overgraph of S'' colored by c is monochromatic. We thus define $f(n, N, |C| + 1) = f(n, f(n, N, |C|), 2)$. □

ALICE
An Advanced Logic for Interactive Component Engineering

Borislav Gajanovic and Bernhard Rumpe

Software Systems Engineering Institute
Carl-Friedrich-Gauß Faculty for Mathematics and Computer Science
Braunschweig University of Technology, Braunschweig, Germany
http://www.sse-tubs.de

Abstract. This paper presents an overview of the verification framework ALICE in its current version 0.7. It is based on the generic theorem prover Isabelle [Pau03a]. Within ALICE a software or hardware component is specified as a state-full black-box with directed communication channels. Components send and receive asynchronous messages via these channels. The behavior of a component is generally described as a relation on the observations in form of streams of messages flowing over its input and output channels. Untimed and timed as well as state-based, recursive, relational, equational, assumption/guarantee, and functional styles of specification are supported. Hence, ALICE is well suited for the formalization and verification of distributed systems modeled with this stream-processing paradigm.

1 Introduction

1.1 Motivation

As software-based systems take ever more and more responsibility in this world, correctness and validity of a software-based system is increasingly important. As the complexity of such systems is also steadily increasing, it becomes ever more complicated to ensure correctness. This especially concerns the area of distributed systems like bus systems in transportation vehicles, operating systems, telecommunication networks or business systems on the Internet. Expenses for verification are an order of magnitude higher than the expenses of the software testing up to now. This, on the one hand, will not change easily in the short run but it will also become evident that crucial parts of software need a different handling than less critical ones. So verification will go along with testing in the future. Full verification, however, will at least be used for critical protocols and components. To reduce verification expenses, a lot has been achieved in the area of theorem provers, like Isabelle [Pau03a, Pau03b, NPW02], in the last years. Based on these foundational works and on the increasing demand for powerful domain specific theories for such theorem provers, we have decided to realize ALICE as a stream-processing-oriented, formal framework for distributed, asynchronously communicating systems.

ALICE is a still growing framework within Isabelle for the verification of logically or physically distributed, interactive systems, where the concept of communication or message exchange plays a central role. An interactive system (see also [BS01] for a characterization) consists of a number of components with precisely defined interfaces. An interactive component interacts with its environment via asynchronous message sending and receiving over directed and typed communication channels. Each channel incorporates an implicit, unbounded buffer that decouples the sending and arrival of messages, and thus describing asynchronous communication. In timed channels, we can control how long these messages remain in this implicit buffer. Fig. 1 illustrates the graphical notation for the syntactical interface of a simple interactive component with one input and one output channel.

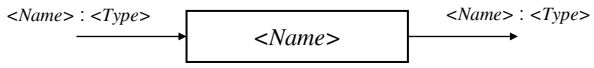


Fig. 1. Illustration of an interactive component as a black-box

In ALICE message flow over channels is modeled by possibly infinite sequences of messages called streams. Such a stream represents the observation of what happens on a channel over time. Since infinite sequences are also included, the liveness and fairness properties of systems can also be dealt with. ALICE provides type constructors *astream* for building (untimed) streams and *tastream* for timed streams over arbitrary types.

As an advanced verification framework, ALICE will offer precisely formalized and comfortably usable concepts based on an underlying logic language called HOL [NPW02] as available in Isabelle. Using a well explored and rather expressive logic language allows us to build on an elaborated set of verification techniques for ALICE.

ALICE will provide support for a number of techniques to specify a component. A specification can be a relation between input and output streams, a stream-processing function, a mapping of input to output, or a set of stream-processing functions allowing to describe non-determinism and underspecification. All variants can be timed or untimed. Further support will be given to map between these styles, allowing to choose appropriate specification techniques for each problem and integrating those later.

Although ALICE does already provide some of these features in its current version, this workshop paper also reports on work still to be done (for the previous version see [GR06]). In a future version ALICE will provide the following:

- A verification framework based on Isabelle supporting development methods for real time, distributed, asynchronously communicating and object oriented systems, respectively. This supports e.g. the development methodologies of [Rum96] and FOCUS [BS01].

- A formal semantics framework for various languages based on stream-processing, e.g. UML's composite structure diagrams that will be formalized based on streams [BCR06, BCR07a, BCR07b].
- A sophisticated verification tool for distributed, interactive systems or, at least, their communication protocols based on stream-processing (see [Ste97] for a survey of stream-processing).

In the following we give a compact overview of Isabelle's HOL and HOLCF that acts as a reminder for experts of the field. An introduction can be found in [NPW02, Reg94] before we start describing features of ALICE in Section 2 and demonstrating the use of ALICE in Section 3 on the Alternating Bit Protocol. Section 4 concludes the paper with a discussion.

1.2 HOL

Isabelle is a generic theorem prover, hence, it can be instantiated with object logics and appropriate proof tactics. Isabelle/HOL [NPW02], in short HOL, is such an elaborated higher order logic, dealing amongst others with sets, relations, total functions, natural numbers, and induction.

HOL provides a term syntax close to mathematical syntax and constructs from functional languages. It also provides basic types like *bool* or *nat*. For building sets over arbitrary types, HOL provides the type constructor *set*. Function types are built with the infix type constructor $\Rightarrow$ for total functions. To build more complex types, the mentioned, and a number of additional basic types and type constructors are provided.

HOL inherits the type system of Isabelle's metalogic including its automatic type inference for variables. There are two kinds of implication: the implication on the level of object logic, in this case HOL, symbolized by $\longrightarrow$, and the symbol $\Longrightarrow$ for Isabelle's inference. Analogously, there is an object logics symbol for the equality, in this case $=$, and the metalogics symbol $\equiv$ for the definitional equality.

In Isabelle assumptions of an inference rule are enclosed in $[\![\ ]\!]$ and separated by ;. The metalogics universal quantifier is symbolized by $\bigwedge$.

1.3 HOLCF

Isabelle/HOLCF [Reg94, MNvOS99], shortly HOLCF, is a conservative extension of HOL with domain theoretical concepts, such as chains, continuity, admissibility, fixpoint recursion and induction, as well as some basic types and datatypes e.g. for lazy lists.

HOLCF extends HOL with the central type class *pcpo* for "pointed complete partial orders". Any type that is a member of this type class features a special relation symbol $\sqsubseteq$ for a partial order on its elements, the least element symbolized by $\bot$, and the existence of the least upper bound for any chain of its elements with respect to $\sqsubseteq$.

This extension is carried out in layers of theories, beginning with the definition of type class *po* for partial orders. *po* is extended to type class *cpo*, where the existence of the least upper bound for any chain, symbolized by $\bigsqcup i.\ Y\ i$, is introduced. Here, *Y* is a chain of growing elements and *i* the index over natural numbers. Based on these theories, monotonicity and continuity for HOL functions on *cpo* types is formalized.

Type class *pcpo* finally introduces the existence of the least element in its members. We call the members of this class HOLCF types. Subsequently, HOLCF provides the new infix type constructor $\rightarrow$ for the construction of continuous functions on HOLCF types. Analogously to the HOLCF types, we call these functions HOLCF functions or operations. These functions, by definition, exhibit the advantages of continuous functions, such as composability, implementability etc. A lambda-abstraction, denoted by Λ (not to confuse with HOL's λ) and a corresponding function application, using the symbol $\cdot$ (opposite to HOL's white space) is provided accordingly.

Subsequently, the fixpoint theory *Fix* mainly implements a continuous fixpoint operator, symbolized by *fix*, and the fixpoint induction principle. Hence, with $\rightarrow$, *fix*, and HOLCF datatypes a complete HOLCF syntax for defining and reasoning about HOLCF functions and types is provided, which is separate from HOL's function space. As an advantage, by construction, HOLCF function abstraction and application remains in the HOLCF world.

1.4 Related Work

A good outline on different approaches to formalize possibly infinite sequences in theorem provers like Isabelle or PVS, as well as a detailed comparison can be found in [DGM97, Mül98]. In contrast to a HOLCF formalization given in [Mül98], where finite, partial, and infinite sequences are defined to model traces of I/O-Automata, our streams have been developed using only partial sequences and their infinite completions, which are more appropriate for modeling interactive systems as these are generally non-terminating. A pure HOL approach based on coinduction and corecursion is described in [Pau97].

Another approach is the formal specification language ANDL introduced in [SS95]. ANDL is a formalization of a subset of FOCUS with an untimed syntax and a fixed and an untimed semantics. Currently, ANDL does not provide an appropriate verification infrastructure or extended sophisticated definition principles, but it is HOLCF oriented. In [SM97] ANDL is used as interface for an A/C refinement calculus for FOCUS in HOLCF. In [Hin98] ANDL is extended to deal with time.

A recent work in this area is [Spi06], where a pure HOL approach to formalize timed FOCUS streams is used. By this approach (see also [DGM97, Mül98]), an infinite stream is represented by a higher-order function from natural numbers

to a set of messages. Furthermore a time-driven approach, as it will briefly be mentioned in Section 2.4, has been chosen there.

Apart from our idea of building such a logical framework, the realization of ALICE is based on a rudimentary formalization of FOCUS streams in HOLCF, developed by D. von Oheimb, F. Regensburger, and B. Gajanovic (the session HOLCF/FOCUS in Isabelle's release Isabelle2005), a concise depiction of HOLCF in [MNvOS99], as well as on the conclusions from [DGM97, SM97]. It is elaborately explained in [GR06]. Additionally, it is worth mentioning that, in the current version HOL's construct *typedef* has been used to define *astream*.

2 ALICE

The newly defined logic ALICE includes the following parts:

- HOL - the full HOL definitions.
- HOL/HOLCF - all theories from HOLCF, like *Pcpo*, *Cont*, etc. that are used on the "interface" between HOL and HOLCF (as discussed in Section 1.3).
- HOLCF - using HOLCF application/abstraction (LCF sublanguage) only.
- ALICE - basic type constructors *astream* and *tastream*, as well as recursion, pattern-matching, automata, etc.
- ALICE - lemmas provided by ALICE theories (they are generally partitioned in timed and untimed properties).

Please note that, for the development of ALICE, we use a combination of HOL and HOLCF syntax, but the user of ALICE does not need to. This is due to the fact that we internally use HOLCF to build up necessary types, operators, and proving techniques, but will encapsulate these as much as possible.

2.1 Basic Features of ALICE

To understand ALICE in more detail, we first summarize its basic features. ALICE provides:

- polymorphic type constructors *astream* and *tastream* for timed and untimed streams over arbitrary HOL types,
- sophisticated definition principles for streams and functions over streams, such as pattern-matching, recursion, and state-based definition techniques,
- incorporated domain theory (concepts of approximation and recursion),
- various proof principles for streams,
- incorporated automata constructs for state-based modeling, also supporting underspecification or non-determinism,
- extensive theories for handling timed streams, functions and properties,
- a powerful simplifier (while developing ALICE, a proper set of simplification rules has been defined carefully in such a way as to be used by ALICE automatically), and

– an extensive library of functions on streams and theorems, as well as commonly needed types (just like in any other programming language, a good infrastructure makes a language user friendly).

The following sections provide brief insights in the above listed features. For a deeper understanding we refer to [GR06].

2.2 Specifying Streams

ALICE provides a basic type constructor called *astream* for specifying untimed streams. For any Isabelle type *t*, the type *t astream* is member of the HOLCF type class *pcpo* as described in Section 1.3. The following exhaustion rule describes the basic structure of untimed streams as well as the fundamental operators for their construction:

```
⋀s. s = ε ∨ (∃h rs. s = <h>⌢rs)
```

A stream *s* is either empty, symbolized by ε, or there is a first message *h* and a remaining stream *rs* so that pre-pending *h* to *rs* yields the stream *s*. The operator <.> builds single element streams and .⌢. defines the concatenation on streams. It is associative and continuous in its second argument and has the empty stream (ε) as a neutral element. If the first argument of concatenation is infinite, the second is irrelevant and the first is also the result of the concatenation. This effectively means that the messages of the second stream then never appear in the observation at all.

According to the above rules, ALICE also offers selection functions, named *aft* for the head and *art* for the rest of a stream, respectively. Function *atake* allows us to select the first *n* symbols from a stream. Function *adrop* acts as a counterpart of *atake* as it drops the first *n* messages from the beginning of a stream *s*. The operator *anth* yields for a number *n* and a stream *s*, the *n*-th message. Beyond that, ALICE provides many other auxiliary functions, e.g. # for the length of a stream, yielding ∞ for infinite streams, *aflatten* for the flattening of streams of streams, *aipower* for the infinite repetition, *afilter* for message filtering. In Section 2.5 we give a tabular review of operators that are available in the current version of ALICE.

Since streams are HOLCF datatypes, they carry a partial order (see also Section 1.3), which is described by the following lemma

```
s1 ⊑ s2 ⟹ ∃t. s1⌢t = s2
```

The above rule characterizes the prefix ordering on streams. It is induced by a flat order on the messages, disregarding any internal structure of the messages themselves. Based on these operators, a larger number of lemmas is provided to deal with stream specifications, like case analysis, unfolding rules, composition rules, associativity, injectivity, and idempotency. Some foundational lemmas are given in Tab. 1.

Table 1. Some foundational lemmas on stream concatenation

$\varepsilon \frown s = s \frown \varepsilon = s$
$(s \frown t) \frown u = s \frown (t \frown u)$
$\#\varepsilon = 0$
$\#\langle m \rangle = 1$
$\#(s \frown t) = \#s + \#t$
$\#s = \infty \Longrightarrow s \frown t = s$

2.3 Timed Streams

Built on the untimed case, ALICE provides another type constructor called *tastream* for specifying timed streams. Structurally, both are rather similar. Again, for any Isabelle type *t*, the type *t tastream* is a member of *pcpo*. The following exhaustion rule describes the basic structure of timed streams. It shows that timed streams may still be empty, contain a message or a tick as their first element:

$\bigwedge ts.\ ts = \varepsilon \vee (\exists z.\ ts = \langle\surd\rangle \frown z) \vee (\exists m\ z.\ ts = \langle Msg\ m\rangle \frown z)$

In addition to ordinary messages, we use a special message $\surd$, called the tick, to model time progress. Each $\surd$ stands for the end of a time frame. To differentiate between the tick and ordinary messages, we use the constructor *Msg* as shown above. This operator is introduced by type constructor *addTick* that extends any type with the tick.

Please note that any timed stream of type *t tastream* is also an ordinary stream of type *(t addTick) astream*. Therefore, all machinery for *astream* types is available.

In addition, ALICE provides a timed take function. *ttake n·s* yields at most *n* time frames from the beginning of a timed stream *s*.

To allow inductive definitions, *tastream* streams may be empty. However, for specifications we restrict ourselves to observations over infinite time, which means that we will only use the subset of timed streams with infinitely many ticks. Therefore, additional machinery is necessary to deal with those. For example, the predicate *timeComplete* is provided to check whether a stream contains infinitely many time frames.

For an integration of both stream classes, operator *timeAbs* maps a timed stream into an untimed one, just keeping the messages, but removing any time information.

2.4 Stream Based Proof Principles

Having the necessary types and type classes as well as auxiliary functions and lemmas at hand, we can introduce proof principles for streams now. At first, we handle the untimed case, as the timed case can be built on that.

Proof Principles for Untimed Streams. A rather fundamental proof principle for untimed streams is the so called take-lemma for streams that gives us an inductive technique for proving equality

```
(∀n. atake n·x = atake n·y) ⟹ x = y
```

Two streams are equal if all finite prefixes of the same length of the streams are equal. More sophisticated proof principles, like pointwise comparison of two streams using the operator *anth* or the below given induction principles are built on the take-lemma. The following is an induction principle for proving a property *P* over finite (indicated by the constructor *Fin*) streams

```
⟦#x = Fin n; P ε; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x
```

As said, when necessary, we base our proof principles directly on HOLCF but try to avoid their extensive exposure. Here is a principle that uses admissibility from HOLCF (*adm*) for predicates to span validity to infinite streams (see [Reg94])

```
⟦adm P; P ε; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x
```

The above induction principles have also been extended to the general use of concatenation, where not only single element streams, but arbitrary streams can be concatenated.

The concept of approximation (provided by HOLCF) and induction on natural numbers can also be used to prove properties involving continuous functions over streams as discussed in Section 2.5.

Proof Principles for Timed Streams. Since timed streams can also be seen as normal untimed streams, the above given proof principles can also be used to prove properties of timed streams.

Please note that we have taken a *message driven* approach to inductively define timed streams. Messages are added individually to extend a stream. This also leads to event driven specification techniques. In the contrary, it would have been possible to model timed streams inductively as a stream *(t list) astream*, where each list denotes the finite list of messages of type *t* occurring in one time frame. This definition would lead to time-driven specification principles. It is up to further investigation to understand and integrate both approaches. As a first step in this direction, ALICE provides a timed-take-lemma for timed streams arguing that streams are equal if they are within first n time frames for each n, as given in the following.

```
(∀n. ttake n·x = ttake n·y) ⟹ x = y
```

Analogously, the following proof principle is based on time frame comparison

```
(∀n. tframe n·x = tframe n·y) ⟹ x = y
```

ALICE provides more sophisticated proof principles for timed streams, but also for special cases of timed streams, such as time-synchronous streams, containing exactly one message per time unit, and the already mentioned time complete streams, containing infinitely many time frames.

2.5 Recursive Functions on Streams

Specifying streams allows us to define observations on communication channels. However, ALICE focusses on specification of components communicating over those channels. The behavior of a component is generally modeled as function over streams and is often defined recursively or even state-based.

A recursively defined function *f* processes a prefix of its input stream *s* by producing a piece of the output stream and continues to process the remaining part of *s* recursively. All functions defined in this specification style are per construction correct behaviors for distributed components. This makes such a specification style rather helpful. Functions of this kind are defined in their simplest form as illustrated in the following (using the function *out* to process the message *x* appropriately)

```
f (<x>⌒s) = (out x)⌒(f s)
```

By construction, these functions are monotonic and continuous (lub-preserving, see below) wrt. their inputs, which allows us to define a number of proof principles on functions.

Table 2. Basic operators in ALICE

Operator	Signature
<.>	'a ⇒ 'a astream
aft	'a astream ⇒ 'a
art	'a astream → 'a astream
atake	nat ⇒ 'a astream → 'a astream
adrop	nat ⇒ 'a astream → 'a astream
anth	nat ⇒ 'a astream ⇒ 'a
#.	'a astream → inat
.⌒.	'a astream ⇒ 'a astream → 'a astream
aipower	'a astream ⇒ 'a astream
apro1	('a * 'b) astream → 'a astream
apro2	('a * 'b) astream → 'b astream
amap	('a ⇒ 'b) ⇒ 'a astream → 'b astream
azip	'a astream → 'b astream → ('a * 'b) astream
afilter	'a set ⇒ 'a astream → 'a astream
atakew	('a ⇒ bool) ⇒ 'a astream → 'a astream
adropw	('a ⇒ bool) ⇒ 'a astream → 'a astream
aremstutter	'a astream → 'a astream
aflatten	'a astream astream → 'a astream
ascanl	nat ⇒ ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b astream → 'a astream
aiterate	('a ⇒ 'a) ⇒ 'a ⇒ 'a astream

A number of predefined auxiliary operators assist in specifying components. Due to expressiveness, we also allow to use operators that are not monotonic or continuous in some arguments, such as ⌒ in its first argument or *aipower*. In ALICE, it is also possible to define more such functions using pattern-matching and recursion. The above notions can also be found in standard literature on semantics like [Win93]. In the following we concentrate on continuous functions.

Continuous Functions - The Approximation Principle. As briefly discussed, continuous functions capture the notion of computability in interactive systems and therefore play a prominent role in stream-processing specification techniques. The behavior of a continuous function for an infinite input can be predicted by the behavior for the finite parts of the input. Thus, its behavior can be approximated. As it has been shown amongst others in [Win93], composition of continuous functions results in continuous functions. Therefore, based on a number of basic functions and equipped with appropriate definition techniques, it becomes easy to specify further functions. ALICE provides amongst others

- pattern-matching and recursion (like in functional languages),
- state-based definitions (using I/O*-automata [Rum96], see Section 2.6),
- fixpoint recursion (using HOLCF), and
- continuous function-chain construction (using HOL's *primrec* and approximation, see [GR06])

Currently, we do have at least the operators on streams depicted in Tab. 2 and Tab. 3 available. For the sake of brevity, we do not explain those further, but refer to [GR06] as well as Section 2.2 and 2.3 and furthermore assume that readers will recognize the functionality through name and signature.

Table 3. Basic operators for timed specifications

Operator	Signature
timeComplete	*'a tastream ⇒ bool*
timeSync	*'a tastream ⇒ bool*
injectTicks	*nat astream → 'a astream → 'a tastream*
timeAbs	*'a tastream → 'a astream*
ttake	*nat ⇒ 'a tastream → 'a tastream*
tframe	*nat ⇒ 'a tastream → 'a astream*
stretchTimeFrame	*nat ⇒ 'a tastream → 'a tastream*
getTime	*nat ⇒ 'a tastream ⇒ nat*

2.6 State-Based Definition Techniques

There is quite a number of variants of state machines available that allow for a state-based description. We use I/O*-automata that do have transitions with one occurring message (event) as input and a sequence of messages (events) as output (hence I/O*). They have been defined in [Rum96] together with a formal semantics based on streams and a number of refinement techniques. In contrast to I/O automata [LT89], they couple incoming event and reaction and need no intermediate states.

As they are perfectly suited for a state-based description of component behavior, we provide assistance for the definition of an I/O*-automaton *A* in ALICE by modeling the abstract syntax as a 5-tuple in form of

```
A = (stateSet A, inCharSet A, outCharSet A, delta A, initSet A)
```

Automata of this structure can be defined using the type constructor *ioa*. I/O*-automata consist of types for its states, input and output messages. *delta* denotes the transition relation of an automaton. It consists of tuples of source state, input message, destination state and a sequence of output messages. The 5th element *initSet* describes start states and possible initial output (that is not a reaction to any incoming message).

As an illustration, we define[1] an I/O*-automaton representing a component dealing with auctions in the American style, where bidders spontaneously and repeatedly spend money and after a certain (previously unknown) timeout the last spender gets the auctioned artifact. The auction component is initialized with an arbitrary but a non-zero timeout. It counts down using the ticks and stores the last bidder as he will be the winner.

```
amiauction :: "((nat * Bid * IAP), Bid addTick, BidUclosed addTick) ioa"
amiauction_def:
  "amiauction ≡
    (UNIV, UNIV, UNIV,
     {t.∃k b m x.

         (* handle time and accept the last bid
              as soon as the time limit is reached *)
         t = ((k+1,b,x), √, (k,b,x), <√>) ∧ k > 0 ∨
         t = ((0,b,x), √, (0,b,x), <√>⌢<Msg closed>) ∨
         t = ((1,b,I), √, (0,b,I), <√>⌢<Msg closed>) ∨
         t = ((1,b,A), √, (0,b,A), <√>⌢<Msg (accept b)>⌢<Msg closed>) ∨

         (* store the new bid m if necessary *)
         t = ((k+1,b,x), Msg m, (k+1,m,A), ε) ∨ t = ((0,b,x), Msg m, (0,b,x), ε)},

     {((ϵ s. fst s > 0 ∧ snd (snd s) = I), ε)})"
```

The above automaton is well-defined, deterministic and complete. By applying the operator *ioafp*, we map this automaton into a function that is continuous by construction. The recursive definition of a stream-processing function is now embedded in the *ioafp* operator, leaving a non-recursive but explicit definition of the actual behavior in an event based style.

In fact, a number of proof principles are established on these state machines that do not need inductive proof anymore, but just need to compare transitions and states. More precisely, the behaviors can then be compared by establishing a (bi-)simulation relation between the automata.

A non-deterministic I/O*-automaton is defined in an analogous form and not mapped to a single but a set of stream-processing functions. This is especially suitable to deal with underspecification.

As said, ALICE is still in development. Although we have initial results on this kind of specification style, we will further elaborate ALICE to comfortably deal with I/O*-automata of this kind in the future.

[1] Due to lack of space, we skip HOL's keyword *constdefs* in front of a definition but symbolize it by indentation. We also do not introduce the necessary type declarations, which is actually straightforward for the specifications used here.

3 Alternating Bit Protocol - An Example

Based on the theory introduced so far, we show the usefulness of ALICE by developing a small, yet not trivial and well known example.

The Alternating Bit Protocol (ABP) is a raw transmission protocol for data over an unreliable medium. Goal of the ABP is to transmit data over a medium that looses some messages, but does not create, modify, rearrange or replicate them. The key idea is that the sender adds an identifier to each message that is being sent back as acknowledgement by the receiver. If the acknowledgement does not arrive, the sender sends the same message again. When only one single message is in transmission, the identifier can boil down to a single bit with alternating value – hence the name of the protocol.

The ABP specification involves a number of typical issues, such as underspecification, unbounded non-determinism and fairness. Fig. 2 illustrates the overall structure of the ABP. A detailed explanation of a similar specification can be found in [BS01].

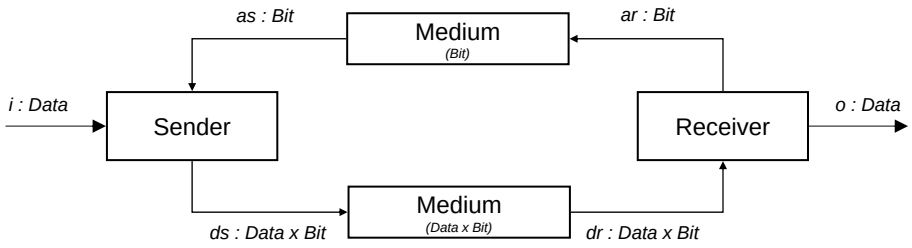


Fig. 2. The architecture of the Alternating Bit Protocol (ABP)

3.1 The ABP Medium

Please note that the medium is modeled after the existing, real world, while sender and receiver need to be specified and later implemented in such a way that they can safely deal with the given medium. So, we first specify the behavior of the medium as described above.

```
Med :: "'t astream ⇒ 't astream ⇒ bool"
Med_def:
  "Med x y ≡
     ∃p. #(afilter {True}·p) = ∞ ∧
         y = apro1·(afilter {a. ∃b. a = (b, True)}·(azip·x·p))"
```

Through the use of an internal oracle stream p, we can describe that a medium does eventually transmit a message if we retry long enough. The fairness, as described below, is deduced from the above specification as follows.

$$[\![\#x = \infty;\ Med\ x\ y]\!] \Longrightarrow \#y = \infty$$

The lemma is proven easily using the following auxiliary lemma, since the lengths of the first and the second pointwise projection (*apro1* and *apro2* respectively) of a stream consisting of ordered pairs are equal.

```
∀x. #x = ∞ ⟶ apro2·(afilter {a. ∃b. a = (b, z)}·(azip·x·p)) = afilter {z}·p
```

The above auxiliary lemma is again proven by induction on the free stream variable *p* using an appropriate proof principle from Section 2.4.

3.2 The Sender

Now, relative to a given medium, we have to define a sender and a receiver that establish the desired behavior: safe transmission of messages. The sender receives data from outside and transmits them together with the alternating bit. We give a specification in a functional style:

```
Snd :: "Data astream ⇒ Bit astream ⇒ (Data * Bit) astream ⇒ bool"
Snd_def:
  "Snd i as ds ≡
     let
        fas = aremstutter·as;
        fb = apro2·(aremstutter·ds);
        fds = apro1·(aremstutter·ds)
     in
        fds ⊑ i ∧
        fas ⊑ fb ∧
        aremstutter·fb = fb ∧
        #fds = imin #i (iSuc (#fas)) ∧
        (#fas < #i ⟶ #ds = ∞)"
```

We explicitly define the channel observations for the sender in Fig. 2. The conjuncts in the *in* part of the definition constrain the sender in the order of their appearance, using the abbreviations from the *let* part, as follows

1. Abstracting from consecutive repetitions of a message via *aremstutter*, we see that the sender is sending the input messages in the order they arrive.
2. The sender also knows which acknowledgement bit it is waiting for, nevertheless, it is underspecified which acknowledgment bit is sent initially.
3. Each new element from the data input channel is assigned a bit different from the bit previously assigned.
4. When an acknowledgment is received, the next data element will eventually be transmitted, given that there are more data elements to transmit.
5. If a data element is never acknowledged then the sender never stops transmitting this data element.

3.3 The Receiver

The receiver sends each acknowledgment bit back to the sender via the acknowledgment medium and the received data messages to the data output channel removing consecutive repetitions, respectively.

```
Rcv :: "(Data * Bit) astream ⇒ Bit astream ⇒ Data astream ⇒ bool"
Rcv_def: "Rcv dr ar o ≡ ar = apro2·dr ∧ o = apro1·(aremstutter·dr)"
```

3.4 The Composed System

The overall system is composed as defined by the architecture in Fig. 2. This composition is straightforwardly to formulate in ALICE:

```
ABP :: "Data astream ⇒ Data astream ⇒ bool"
ABP_def:
  "ABP i o ≡ ∃as ds dr ar. Snd i as ds ∧ Med ds dr ∧ Rcv dr ar o ∧ Med ar as"
```

This formalization of the ABP uses a relational approach similar to the specification in [BS01]. However, formalizations as sets of functions or in a state-based manner are possible as well. Using a more elaborate version of ALICE, we will be able to define a state-based version of sender and receiver (similar to [GGR06]), which is on the one hand more oriented towards implementation and on the other hand might be more useful for inductive proof on the behaviors. Most important however, we will be able to prove that this relational and the state-based specifications will coincide.

For this case study, we remain in the relational style and specify the expected property of the overall system (without actually presenting the proof):

```
ABP i o ⟹ o = i
```

Please note that, at this stage of the development of ABP, there are neither realizability nor sophisticated timing constraints considered in the above formalization. Due to relational semantics, additional refinement steps are then needed to reduce the underspecification towards an implementation oriented or timing-aware style, since there are infinite streams fulfilling the specification that are not valid protocol histories. These, however, would not occur, when using sets of stream-processing functions as they impose continuity on the overall behavior.

4 Discussion

In this paper we have introduced ALICE, an advanced logic for formal specification and verification of communication in distributed systems. ALICE is embedded in the higher order logic HOL, which itself is formalized using the Isabelle generic theorem prover.

Our approach is based on using HOLCF to deal with partiality, infinity, recursion, and continuity. We provide techniques to use ALICE directly from HOL, thus preventing the user to actually deal with HOLCF specialities.

ALICE is currently under development. So not all concepts and theories presented here are already completely mature. Further investigations will also deal with the question of expressiveness, applicability and interoperability. Beyond the ABP, we already have some experience with other formalizations that show that the overhead of formalizing a specification in ALICE as apposed to a mere paper definition is not too bad. However, it also shows where to improve comfort.

References

[BCR06] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML. Towards a System Model for UML. The Structural Data Model. Technical Report TUM-I0612, Munich University of Technology, 2006.

[BCR07a] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML, Towards a System Model for UML, Part 2: The Control Model. Technical Report TUM-I0710, Munich University of Technology, 2007.

[BCR07b] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML, Towards a System Model for UML, Part 3: The State Machine Model. Technical Report TUM-I0711, Munich University of Technology, 2007.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer Verlag Heidelberg, 2001.

[DGM97] M. Devillers, D. Griffioen, and O. Müller. Possibly Infinite Sequences in Theorem Provers: A Comparative Study. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 89–104, Murray Hill, New Jersey, 1997. LNCS 1275, Springer Verlag.

[GGR06] B. Gajanovic, H. Grönniger, and B. Rumpe. Model Driven Testing of Time Sensitive Distributed Systems. In J-P. Babau, J. Champeau, and S. Gerard, editors, *Model Driven Engineering for Distributed Real-Time Embedded Systems: From MDD Concepts to Experiments and Illustrations*, pages 131–148. ISTE Ltd, 2006.

[GR06] B. Gajanovic and B. Rumpe. Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Technical Report Informatik-Bericht 2006-03, Braunschweig University of Technology, 2006.

[Hin98] U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* Dissertation, Munich University of Technology, 1998.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MNvOS99] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[Mül98] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* Dissertation, Munich University of Technology, 1998.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer Verlag, 2002.

[Pau97] L. C. Paulson. Mechanizing Coinduction and Corecursion in Higher-Order Logic. *Journal of Logic and Computation*, 7(2):175–204, 1997.

[Pau03a] L. C. Paulson. *Introduction to Isabelle.* Computer Laboratory, University of Cambridge, 2003.

[Pau03b] L. C. Paulson. *The Isabelle Reference Manual. With Contributions by Tobias Nipkow and Markus Wenzel.* Computer Laboratory, University of Cambridge, 2003.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Dissertation, Munich University of Technology, 1994.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, 1996.

[SM97] R. Sandner and O. Müller. Theorem Prover Support for the Refinement of Stream Processing Functions. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97).* Springer, 1997.

[Spi06] M. Spichkova. FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study. Technical Report TUM-I0602, Munich University of Technology, 2006.

[SS95] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der FOCUS-Entwicklungsmethodik. Technical Report TUM-I9529, Munich University of Technology, 1995.

[Ste97] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.

A History-based Verification of Distributed Applications

Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan

German Research Center for Artificial Intelligence (DFKI GmbH)
Saarbrücken, Germany
{langenstein,nonnengart,rock,stephan}@dfki.de

Abstract. Safety and security guarantees for individual applications in general depend on assumptions on the given context provided by distributed instances of operating systems, hardware platforms, and other application level programs that are executed on these platforms. The problem for formal approaches is to formalize these assumptions without having to look at the details of the (formal) model of the operating system (including the machines that execute applications).
The work described in this paper presents a modular approach which uses histories of observable events to specify runs of distributed instances of the system. The overall verification approach decomposes the given verification problem into local tasks along the lines of assume-guarantee reasoning. In this paper we focus on this methodology and on its realization in the Verification Support Environment (VSE). We also illustrate the proposed approach with the help of a suitable example, namely the specification and verification of an SMTP server whose implementation makes extensive use of various system calls as e.g. fork and socket commands.

1 Introduction

The theory developed in the following aims at a modular approach for the specification and verification of concurrent systems with heterogeneous components. *Concurrency* typically results from the actual parallel execution of independent systems and the abstraction from a concrete scheduler within the context of a given platform. Like the systems themselves their formal models will consist of various types of components specified by different types of state transition systems. In the composed (global) system the components interact with each other by certain communication mechanisms.

In this paper we consider an instantiation of the general approach which is taken from the context of the Verisoft project where a pervasive formal model of a distributed collection of hardware-software platforms with application level programs running on each of these was established, [1].

Instead of verifying application level programs directly on the Verisoft model, we propose to use traces of *observable events* that according to a given view are attached to steps of computations (or runs) of the distributed system as they have been formally defined in Verisoft. Since for a state in a run we collect all events that have happened so far we call these (finite) lists of events *histories*. The behavior of the global system as well as that of single components is then

specified by sets of histories thereby abstracting from the local state spaces of the components. Like an input-output specification for a sequential piece of software sets of histories describe the concurrent, typically non-terminating computation of a global system or component thereof. Event traces defined by a certain view on a given model provide an appropriate interface for an inductive analysis of cryptographic protocols, [2, 3] or an information flow analysis [4].

Our approach is modular in the sense that the task of verifying a history specification against runs of the global system can be decomposed into local verification tasks for its components. Following the general assume-guarantee approach, see [5] for comprehensive discussion of these approaches, for each event specified in a history there is exactly one component which may *violate* the specification at this point. Therefore, our events allow to determine the component considered to be *responsible* for the event. Events in a history a particular component is not responsible for are considered as assumptions of that component w.r.t. its environment.

Each history specification, possibly obtained by a combination of sub-specifications, has to be verified against all components of the overall model. Here we focus on the verification of C0[1] machines that execute (and give meaning to) C0 programs extended by *external calls* that allow to exchange information with the corresponding operating system and, via a network component, with C0 machines that run on different (remote) instances of the operating system. As an example we consider the implementation of an e-mail system consisting of an e-mail client, a (so-called) mail bag, a SMTP-server, and a SMTP-client.

In the next section we summarize the instantiation of the general approach by the *Verisoft model* of distributed systems. As a concrete example in section 3 we provide the *specification of the SMTP server* by a set of histories. In Section 4 we describe the verification of application-level C0 programs, like the implementation of the SMTP server, by means of a transformation to abstract sequential programs that *replay and extend* histories. Section 5 summarizes the described approach and outlines possible future work.

2 Events and Histories

2.1 The Verisoft Model

In the Verisoft project a formal model of an operating system was developed and verified with respect to its implementation, [6]. Application level programs run on (abstract) machines that are part of the model. They interact by a kind of RPC like mechanism and access resources (of the OS) by external calls.

[1] C0 is an imperative C-like programming language with some restrictions compared to standard C. In C0 there are besides others no side effects in expressions and pointers are strictly typed (see [1] for a complete description of C0).

A model of a distributed system consists of instances of (the same) system and an abstract network component connecting them. Each system is identified by a unique *network address* $na \in Na$. It basically consists of two parts. The *simple operating system* provides resources like input-output devices, a file system, and sockets for the communication over the network component. A *process* is identified by a *process identifier* $pid \in Pid$ and the network address of the system instance it is running on. For each process given by $p = mk\_proc(pid, na)$ a machine (interpreter) is part of the system na. In this paper we are interested in applications implemented in C0, the subset of C considered in Verisoft. Hence our processes represent abstract execution mechanisms where the program part of a configuration is a C0 program π.

From point of view of an application programmer the system context consists of all operating systems and the network connecting them. Hence we consider this complete context as a single component in our decomposition. It will be denoted by the constant SOS.

The view we have chosen is depicted in Figure 1.

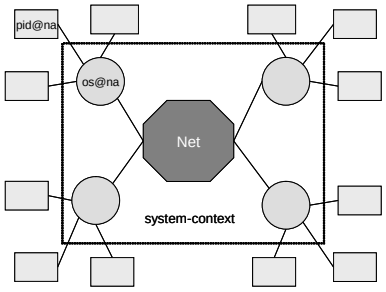


Fig. 1. Verisoft Model

The communication between user programs and the surrounding operating system (instance) is by so called *external calls*. External calls $c_{ext}(\bar{\tau} : \bar{z}, res)$ take the same syntax as ordinary function calls. We use $\bar{\tau}$ as a sequence of value parameters and $\bar{z}, res$ as a sequence of return parameters. Typically the value of res will indicate success or failure. Whenever a system call is reached during the computation of a process p the normal execution as given by the (small step) semantics $R_{C0} \subseteq Conf \times Conf$ is interrupted (stopped) and a request is sent to the corresponding operating system. With these steps of a global computation we associate *events* of the form $mk\_ev(p, SOS, m)$ where the message m encodes the particular call given by c_{ext} and the values of the parameters $(\bar{\tau} : \bar{z})$ in mem'. For a call of socket read `socket_read`(`sid` : `length`, `buffer`, `ec`) the corresponding message will by $Sread(sid, length)$ where $Sread$ is a constructor symbol for an abstract data type and $sid, length$ are the values of the programming variables `sid`, `length` in mem'. They indicate the socket and the length of the string to be read.

To model the return of external calls the standard C0 machines have to be extended by steps where the resulting configuration is determined by an answer (message) from the corresponding operating system. The (answer) information intended for process p will be written to the return parameters (of the pending call). The event we associate with these steps is of the form $mk\_ev(SOS, p, m)$ where the message m represents the return information. For example a successful call of socket read the message will be $Succ\_sread(length, buffer)$ where $length$

indicates the elements in the fixed length array $buffer$ that have actually been read. These values uniquely determine the values of the result parameters after return of that external call.

2.2 History Specifications

Having defined events for all external calls (and also RPC calls) we may specify global system runs by a set (or unary predicate) H of finite sequences of events. A global System $SOS(\pi_0, , \ldots, \pi_{n-1})$ consisting of arbitrary many instances of the operating system with arbitrary many C0-processes executing $\pi_0, , \ldots, \pi_{n-1}$.

By $SOS(\pi_0, , \ldots, \pi_{n-1}) \models H$ we denote the fact that for each state in a global run of $SOS(\pi_0, , \ldots, \pi_{n-1})$ the sequence of events that have happened so far satisfies H. Given an event $e = mk\_ev(s, r, m)$ we say that s (the sender) is *responsible* for that event. In addition we define that e is *relevant* for s as well as the receiver r. By $SOS(\pi_0, , \ldots, \pi_{n-1}) \downarrow i \models H$ we denote the fact that no process p executing $\pi_i \in \{\pi_0, , \ldots, \pi_{n-1}\}$ violates H first, i.e. in a step p is responsible for. Similarly we use $SOS(\pi_0, , \ldots, \pi_{n-1}) \downarrow SOS \models H$ for a projection to the operating systems themselves.

To establish $SOS(\pi_0, , \ldots, \pi_{n-1}) \downarrow i \models H$ locally outside the context of a global system we transform π_i into a program $\tilde{\pi}_i$ where the external calls manipulate histories. Altogether $\tilde{\pi}_i$ replays and possibly extends histories.

Using $\tilde{\pi}_i \models H$ for the fact that $\tilde{\pi}_i$ preserves H, soundness of this method (w.r.t. the verification of application level programs) is demonstrated by a kind of *simulation theorem* that allows to conclude that $\tilde{\pi}_i \models H \Rightarrow SOS(\pi_0, , \ldots, \pi_{n-1}) \downarrow i \models H$ holds. The simulation theorem is application independent and established by looking at the C0 execution mechanism. As opposed to that $\tilde{\pi}_i \models H$ is concerned with the verification of *individual* programs.

Before we present the specification of the SMTP-server and the verification technique given by the ˜-transformation we have to discuss special events that provide the binding between programs π_i and processes in a given history.

2.3 Life Cycle Events

Histories cover the whole life cycle of processes. This includes the association of process identifiers with the programs that are executed. This binding takes place upon *creation* of a new process. The lifetime of a process ends by an explicit *termination* event.

In the verification process we are interested in a particular program (text) π. To associate programs with process identifiers we assume a fixed enumeration of programs. In histories $\pi = \pi_i$ is then represented by the constant i.

Create events $\langle SOS, p, Create(i)\rangle$ are *caused* by the corresponding instance of the SOS while the identifier for the new process $p = mk\_proc(na, pid)$ is

considered as the recipient of the message indicating the program text π_i to be executed starting with a fixed initial state.

Clone create events $\langle SOS, p, Clone\_Create(p', b)\rangle$ are caused by the corresponding SOS as possible positive reaction to a call of fork. $p = mk\_proc(na, pid)$ indicates the *child process* which *continues* to execute the program of the *parent process* given by p' in the message. $b \in \{0, 1\}$ is a flag indicating access to the terminal. The initial state of p is the state of p' reached before execution of fork.

A process p (executing some π_i) is terminated by *exit* or *kill events*. Exit events are caused by p while kill events are caused by the corresponding instance of the SOS.

A sub-history h_0 of h is called a *thread* of p in h if there is exactly one create event for p in h_0 which is the first event (in h_0) relevant for p. The thread is called open if h_0 does not contain a terminating event for p.

Now in defining $\tilde{\pi}_i \models H$ by a program $\tilde{\pi}(i)$ that replays and (possibly) extends histories h we have to consider all threads in h that execute π_i. Since a given specification H can only be violated if some $h \in H$ is actually extended by $\tilde{\pi}$ we restrict ourselves to open threads executing π_i. It is a simple observation that for each p there is at most one open thread in a given h. Therefore we provide a (guess of) p as an argument to the replay and extend procedure. If there is no open thread of p that executes π_i, then the given history h is delivered unchanged as the result.

Open threads chosen in this way include those where a child p of p' is created by an event $\langle SOS, p, Clone\_Create(p', b)\rangle$ following a call of fork during the execution of π_i. Since we have to know the correct internal state the replay and extend procedure has to start with the first ancestor p'' of p. The computation of this process is initiated by a create event $\langle SOS, p'', Create(i)\rangle$.

In the start of the replay and extend procedure for π_i as well as in the implementation of calls of fork we use the function $anc(i, p, h)$ which computes the sequence of ancestor threads for a given p and h. In case there is no open thread of p in h executing π_i $anc(p, h) = []$. For $anc(p, h) = [h_0, \ldots, h_{n_1}]$, h_{n_1} is the open thread of p in h with $\langle SOS, p, Create(i)\rangle$ or $\langle SOS, p, Clone\_Create(p', b)\rangle$ as first event and h_0 is the first ancestor thread with first element $\langle SOS, p'', Create(i)\rangle$.

3 SMTP-Server

As already mentioned earlier we considered a non-trivial example in the Verisoft context, namely the full implementation (in a C-like language) and the full specification (in terms of histories) of an SMTP-Server as part of an Simple Mail Transfer Scenario. All in all this implementation required about 7.500 lines of code.

The SMTP server listens for connections from SMTP clients. If a connection has been established, it spawns a child process, which inherits the socket grant-

ing access to that new connection. The child communicates with the remote SMTP client while obeying the so-called SMTP Protocol. In the meantime, the main SMTP server process listens again for new connections and spawns child processes to handle the session. This behaviour can be formalised by a step by step description of the main process and its child processes. For the formalisation we fix the constant SOS (Simple Operating System) representing the operating system. Any process – and thus the SOS as well – is determined by a network address (the host) and a process id on this host. We assume that – for a given process p – we can access the network address by $get\_na(p)$.

For simplicity we make use of the following definitions: For every history h and process p we define $h \downarrow p$ as the *projection* of the history h on process p. I. e.,

$$\begin{aligned}
() \downarrow p &= () \\
(\langle s, r, m\rangle \circ h) \downarrow p &= \langle s, r, m\rangle \circ h \downarrow p \text{ if s=p or r=p} \\
(\langle s, r, m\rangle \circ h) \downarrow p &= h \downarrow p \text{ otherwise}
\end{aligned}$$

With $h^+ = \{h' \in Hist \mid h' \downarrow p = h\}$ we can describe (for a given history h and an implicitly given process p) the set of histories whose projection on p is just h. Recall that we defined the binary operator $\circ$ on histories as the concatenation of its two arguments. In what follows we also use this operator for the "concatenation" of two history sets: $H_1 \circ H_2 = \{h_1 \circ h_2 \mid h_1 \in H_1, h_2 \in H_2\}$.

The top-level specification (in terms of histories) of the SMTP-Server then looks as follows: we consider the prefix-closure of the set $H_{SMTP\_Server}(p)$ where $H_{\text{SMTP\_Server}}(p) = H_{\text{INIT}}(p) \circ H_{\text{LOOP}}(p, sid)$ for some process p representing the SMTP-Server process and some socket id sid.
The history set $H_{\text{INIT}}(p)$ describes the initialization phase of the SMTP-Server. I. e., the SOS first creates the SMTP-process (represented by the message $Create(SMTP\_Server, SOS, 1)$). Then the newly created SMTP-process sends a message to the SOS that it wants a socket to be opened on port 25 (the standard SMTP-port). After the SOS successfully responds with a new socket id the SMTP-Server requests to listen to this new socket. The history set $H_{\text{INIT}}(p)$ is thus easily defined as

$$\begin{aligned}
H_{\text{INIT}}(p) = (&\langle SOS, p, Create(SMTP_Server, SOS, 1)\rangle \circ \\
&\langle p, SOS, Sopen(25)\rangle \circ \langle SOS, p, Succ_sopen(sid)\rangle \circ \\
&\langle p, SOS, Slisten(sid)\rangle \circ \langle SOS, p, Succ\rangle)^+
\end{aligned}$$

for a process p representing the SMTP-Server process and a socket id sid. The history set $H_{\text{LOOP}}(p, sid)$ is supposed to cover the parent process of the SMTP-server together with all the children processes that might be initiated. It is

defined as the smallest set of histories that satisfies the equation

$$\begin{aligned} H_{\text{LOOP}}(p, sid) = \quad & (\langle p, SOS, Saccept(sid)\rangle)^+ \cup \\ & (H_{\text{ACC}}(p, sid, sid') \circ H_{\text{FORK\_CALL}}(p) \circ \\ & \quad ((H_{\text{FORK\_ANS\_C}}(p') \circ H_{\text{CHILD}}(p', sid)) \cap \\ & \quad\ (H_{\text{FORK\_ANS\_P}}(p) \circ H_{\text{CLOSE}}(p, sid') \circ H_{\text{LOOP}}(p, sid)))) \end{aligned}$$

for some socket id $sid' \neq sid$ and some process $p' \neq p$.
This history set $H_{\text{LOOP}}(p, sid)$ might require some more explanation. First the SMTP-Server issues a socket-accept command. This command might never be answered and thus the SMTP-Server might wait forever (first line in the definition of $H_{\text{LOOP}}(p, sid)$). If, however, there is an answer to the accept-request (another process issued a corresponding connect-request) then the SMTP-Server calls a fork-command, thus producing a child of its own process. Now, both the SMTP-Server and its child run concurrently as indicated by the intersection of the two history sets in the last two lines of the $H_{\text{LOOP}}(p, sid)$ definition.

With this explanation the definition of the history sets $H_{\text{ACC}}(p, sid, sid')$, $H_{\text{FORK}}(p)$, and $H_{\text{CLOSE}}(p, sid')$ should be fairly obvious, namely

$$\begin{aligned} H_{\text{ACC}}(p, sid, sid') &= (\langle p, SOS, Saccept(sid)\rangle\langle SOS, p, Succ\_saccept(sid', rna, rpn)\rangle)^+ \\ &\quad \text{for some remote network address } rna \text{ and port number } rpn \\ H_{\text{FORK\_CALL}}(p) &= (\langle p, SOS, Afork(1)\rangle)^+ \\ H_{\text{FORK\_ANS\_P}}(p) &= (\langle SOS, p, Succ\_afork(hdl)\rangle)^+ \text{ for some handle } hdl \neq none \\ H_{\text{FORK\_ANS\_C}}(p) &= (\langle SOS, p, Create\_Clone(i_C, p, 1)\rangle\langle SOS, p, Succ\_afork(none)\rangle)^+ \\ H_{\text{CLOSE}}(p, sid) &= (\langle p, SOS, Sclose(sid)\rangle\langle SOS, p, Succ\rangle)^+ \end{aligned}$$

Note that the first argument of the *Create_Clone* message indicates the (index of the) program that is supposed to run as the child process.
Remains the most complicated case, namely the specification of the child process which is responsible for carrying out the SMTP protocol. As above, we consider only the successful case here.

$$\begin{aligned} H_{\text{CHILD}}(p, sid) = H_{\text{GREETING}}(p, sid) \circ H_{\text{ReadEmails}}(p, sid) \circ \\ H_{\text{QUIT}}(p, sid) \circ H_{\text{CLOSE}}(p, sid) \end{aligned}$$

The history set for $H_{\text{CLOSE(p,sid)}}$ is already defined above. $H_{\text{GREETING}}(p, sid)$ and $H_{\text{QUIT}}(p, sid)$ look as follows:

$$\begin{aligned} H_{\text{GREETING}}(p, sid) &= H_{\text{READY}}(p, sid) \circ H_{\text{ReadLine}}(p, sid, \text{"EHLO "} + ip_r) \circ \\ &\quad H_{\text{GREETS}}(p, sid, ip_r) \quad \text{for some remote ip address } ip_r \\ H_{\text{READY}}(p, sid) &= (\langle p, SOS, Swrite(sid, \text{"220 "} + get\_na(p) + \\ &\quad \text{" SMT Service Ready"})\rangle\langle SOS, p, Succ\rangle)^+ \\ H_{\text{GREETS}}(p, sid, ip_r) &= (\langle p, SOS, Swrite(sid, \text{"250 "} + get\_na(p) + \text{" greets "} + ip_r)\rangle \\ &\quad \langle SOS, p, Succ\rangle)^+ \\ H_{\text{QUIT}}(p, sid) &= H_{\text{ReadLine}}(p, sid, \text{"QUIT"}) \circ \\ &\quad (\langle p, SOS, Swrite(sid, \text{"221 "} + \text{p} + \text{" closing"})\rangle \circ \\ &\quad \langle SOS, p, Succ\rangle \circ \langle p, SOS, Exit\rangle)^+ \end{aligned}$$

ReadLine consists essentially of successively reading one character after the other. A slight complication arises as it may be possible that the attempt to

read a single character may be successful, yet results in an empty string (i. e., we assume the socket-read command to be non-blocking).

$$H_{\text{ReadLine}}(p, sid, string) = \begin{cases} H_{\text{ReadString}}(p, sid, string) & \text{if } \exists s: \ string = s \hat{} CR \hat{} LF \\ & \text{and } s \text{ does not contain } CR \hat{} LF \\ \emptyset & \text{otherwise} \end{cases}$$

i. e., reading a line means to read a string that (uniquely) ends with a carriage return (CR) followed by a line feed (LF).
$H_{\text{ReadString}}$ is defined as the smallest set satisfying the equations

$$\begin{aligned} H_{\text{ReadString}}(p, sid, "") &= ()^+ \\ H_{\text{ReadString}}(p, sid, c \hat{} s) &= H_{\text{ReadChar}}(p, sid, c) \circ H_{\text{ReadString}}(p, sid, s) \end{aligned}$$

where

$$\begin{aligned} H_{\text{ReadChar}}(p, sid, c) &= H_{\text{ReadEmpty}}(p, sid) \circ H_{\text{ReadChar1}}(p, sid, c) \\ H_{\text{ReadChar1}}(p, sid, c) &= (\langle p, SOS, Sread(sid, 1)\rangle\langle SOS, p, Succ\_sread(1, "c")\rangle)^+ \end{aligned}$$

and $H_{\text{ReadEmpty}}(p, sid) = \mu H.(H = ()^+ \cup H_{\text{ReadEmpty1}}(p, sid) \circ H)$ where

$$H_{\text{ReadEmpty1}}(p, sid) = (\langle p, SOS, Sread(sid, 1)\rangle\langle SOS, p, Succ\_sread(0, "")\rangle)^+$$

Remains to specify the history set $H_{\text{ReadEmails}}$ (which in addition covers writing the email to the Inbox file). $H_{\text{ReadEmails}}$ is the smallest set satisfying the equation $H_{\text{ReadEmails}}(p, sid) = ()^+ \cup \big(H_{\text{ReadEmail}}(p, sid) \circ H_{\text{ReadEmails(p,sid)}}\big)$, i. e.,

$$H_{\text{ReadEmails}}(p, sid) = \mu H.\ \big(H = ()^+ \cup H_{\text{ReadEmail}}(p, sid) \circ H\big)$$

where $H_{\text{ReadEmail}}(p, sid)$ splits into several parts, namely in reading the sender's address, the recipient's address, the email data and the writing of the email to the file system.

$$\begin{aligned} H_{\text{ReadEmail}}(p, sid) &= H_{\text{ReadS}}(p, sid, s) \circ H_{\text{ReadR}}(p, sid, r) \circ H_{\text{ReadD}}(p, sid, d) \circ \\ &\quad H_{\text{WriteEmail}}(p, s \hat{} r \hat{} d) \quad \text{for some } s, r, d \\ H_{\text{ReadS}}(p, sid, s) &= H_{\text{ReadLine}}(p, sid, "\text{MAIL FROM: } " + s) \circ \\ &\quad (\langle p, SOS, Swrite(sid, "OK")\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadR}}(p, sid, r) &= H_{\text{ReadLine}}(p, sid, "\text{RCPT TO: } " + r) \circ \\ &\quad (\langle p, SOS, Swrite(sid, "OK")\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadD}}(p, sid, d) &= H_{\text{ReadLine}}(p, sid, "\text{DATA:}") \circ \\ &\quad (\langle p, SOS, Swrite(sid, "\text{354 Start mail input;} \\ &\qquad \text{end with CRLF . CRLF}")\rangle \circ \langle SOS, p, Succ\rangle)^+ \circ \\ &\quad H_{\text{ReadD}'}(p, sid, d) \circ \\ &\quad (\langle p, SOS, Swrite(sid, "OK")\rangle \circ \langle SOS, p, Succ\rangle)^+ \\ H_{\text{ReadD}'}(p, sid, ".") &= H_{\text{ReadLine}}(p, sid, ".") \\ H_{\text{ReadD}'}(p, sid, l \hat{} d) &= H_{\text{ReadLine}}(p, sid, l) \circ H_{\text{ReadD}'}(p, sid, d) \quad \text{provided } l \neq "." \end{aligned}$$

The final step is to specify $H_{\text{WriteEmail}}$.

$$\begin{aligned} H_{\text{WriteEmail}}(p, e) = (&\langle p, SOS, Flock(Inbox)\rangle\langle SOS, p, Succ\rangle \circ \\ &\langle p, SOS, Fseek(Inbox, 1, 0)\rangle\langle SOS, p, Succ\_fseek(pos_1)\rangle \circ \\ &\langle p, SOS, Fwrite(Inbox, e)\rangle\langle SOS, p, Succ\_fwrite(pos_2, n)\rangle \circ \\ &\langle p, SOS, Funlock(Inbox)\rangle\langle SOS, p, Succ\rangle)^+ \end{aligned}$$

for some file positions pos_1 and pos_2.

It is certainly out of the scope of this paper to show all the verification details for the whole SMTP-Server. Instead, we emphasise on a small portion of it, namely the readLine procedure as specified above.
In a VSE-like fashion the procedure is listed below:

```
PROCEDURE readLine(sid:length,buffer,res)
int length, ec;
buffer buffer_array;
char c, cprevious;
bool res;
 BEGIN length := 1; res := true; c := null; cprevious := null;
  cl := nil;
  WHILE ((cprevious /= CR OR c /= LF) AND res = true) DO
   length := 1; socket_read(sid:length,buffer,ec);
   if (ec = SUCC) then res := true else res := false fi;
   if (length = 1 and res = t) then
    cprevious := c; c := buffer[0]; cl := write(cl,c) fi
  OD;
 END
```

The readline procedure is supposed to read characters from the given TCP/IP socket until it finds a CR followed by a LF. This behaviour is described by the history set $H_{\text{ReadLine}}(p, sid, cl)$ for a procedure identifier p, socket id sid and a list of characters (string) cl. The segments of the histories that are members of this set are the results of calling the *readLine* procedure from above. Therefore, for the verification of the SMTP server, we need to make sure that this procedure (implementation) meets its intended semantics (the corresponding history sets from above).

According to the technique described above we have to prove the following property:

$$
\begin{array}{l}
h_c^0 = h_c \wedge h_{out}^0 = h_{out} \wedge mode = fin \\
\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\quad mode \neq stop \rightarrow \\
\qquad \exists h : h_{out} = h_{out}^0 \circ h \wedge ((mode = fin \wedge res = t) \leftrightarrow h \in H_{\text{ReadLine}}(p, sid, cl))
\end{array}
$$

The proof of this property is split into three main lemmas (and several small lemmas about the data structures used): The first lemma is formulated close to an invariant used to deal with the (single) while loop occurring in the body of *readLine*.

$$
\begin{array}{l}
h_{out}^0 = h_{out} \wedge mode = fin \\
\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\quad mode \neq stop \\
\qquad \rightarrow \exists h : h_{out} = h_{out}^0 = h \wedge \\
\qquad\quad ((mode = fin \wedge res = t) \\
\qquad\qquad \rightarrow h \in H_{\text{ReadString}}(p, sid, cl) \circ H_{\text{ReadEmpty}}(p, sid)) \wedge \\
\qquad\quad (h \in H_{\text{ReadString}}(p, sid, cl) \circ H_{\text{ReadEmpty}}(p, sid) \wedge cl \neq \langle\rangle \\
\qquad\qquad \rightarrow (mode = fin \wedge res = t))
\end{array}
$$

The following lemma states that we can drop the history sets $H_{\text{ReadEmpty}}$, because $H_{\text{ReadEmpty}}(p, sid) \setminus H_{\text{ReadEmpty1}}(p, sid) = \{[]\}$.

$$
\begin{array}{l}
h^0_{out} = h_{out} \wedge mode = fin \\
\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\quad mode \neq stop \rightarrow \exists h : \; h_{out} = h^0_{out} \circ h \wedge \\
\qquad ((mode = fin \wedge res = t) \rightarrow h \notin H_{\text{ReadEmpty1}}(p, sid) \circ H)
\end{array}
$$

Finally, we need a lemma that deals with the fact that end of lines are marked with $\langle CR, LF\rangle$. Notably, the proof for this lemma does not require any knowledge about the external call simulation *socket_read_sim*. Thus this example shows how a proof can be separated into parts dealing with concurrent communication and those dealing with properties independent of the communication, even if the properties are not separated by the program structure.

$$
\begin{array}{l}
\langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle \\
\qquad\qquad mode = fin \wedge res = t \rightarrow \exists cl_0 : cl = cl_0 \circ \langle CR, LF\rangle
\end{array}
$$

4 Application Level Programs

In this section we describe the construction of $\tilde{\pi}_i$ out of π_i. The C0 program π with external calls is transformed into a program $\tilde{\pi}$ that takes histories as input and produces histories as output but uses only standard function calls. Since histories describe initial segments of nonterminating behaviors the new program is intended always to terminate. We consider its result as an *approximation* of the computation of π following the general replay and extend strategy outline above.

We suggest a uniform transformation of the program into an approximation exhibiting the same behavior as the original program with respect to prefixes of event histories. The transformation preserves the structure of the program. Thus it is possible to use a verification approach that follows the structure of the implementation. Moreover this approach enables us to employ well known verification techniques for sequential programs as described in, for instance, [7] and [8]. The latter system has been used for the verification of SMTP.

4.1 Computing Approximations

In this section the uniform procedure to convert programs π_i into their approximations $\tilde{\pi}_i$ is described. Let the program π_i be given as $\pi_i = (\delta_i|\alpha_i(\bar{x}))$, where $\bar{x}$ are the (program) variables occurring free in α_i and δ_i is the list of procedure

declarations used in α_i. The function $approx_{\pi_i}(p_0, h_0)$ will be computed by the program $\tilde{\pi_i}$ given by

$$
\begin{array}{l}
(\ approx\_i(p, h_{in} : h_{out}, mode) \Leftarrow \textbf{declare}\ h_c := h_{in}; \bar{x} := \bar{\sigma} \\
\quad \textbf{begin}\ mode := fin; h_{out} := []; start\_i(: p, h_c, h_{out}); \widetilde{\alpha_i}(\bar{x}); \\
\qquad stop(: h_{out}, h_c, mode)\ \textbf{end}, \\
\quad ext\_call(\pi_i), \tilde{\delta_i} \mid approx\_i(p_0, h_0 : h_1, m_0)\ )
\end{array}
$$

where the initial values of h_1 and m_0 (used to return the results) are not relevant.

The sequence $ext\_call(\pi_i)$ contains declarations for the procedures that simulate the external calls occurring in π_i together with additional start and stop procedure, *start_i* and *stop*, respectively.

In the computation of the approximation a local variable h_c is used that contains the *currently remaining history* during the execution of $\widetilde{\alpha_i}$. It is set to h_{in} initially. The output history is collected in h_{out} as the computation proceeds while the mode is kept in *mode*.

The construction is guided by the following general idea. An initial segment of the computation of α_i executed by p is replayed using (consuming) h and extending h_{out}.

External calls $c(\bar{\tau} : \bar{z}, res)$ are replaced (or simulated) by procedures with declarations $c\_sim(p, \bar{x} : \bar{y}, res, h_c, h_{out}, mode) \Leftarrow body_c$. The simulating procedures analyze and shorten (consume) the current history h and extend the current output h_{out}. The first argument indicates the process that is executing α_i. Let $\bar{v}$ and $\bar{w}$ be the values of $\bar{\tau}$ and $\bar{z}$, respectively.

If in h_c there is no event generated by p, then the computation (of $\tilde{\alpha_i}$) stops with $h_{out} \circ h \circ [ev_c(p, \bar{v} : \bar{w})]$ as final output, where $ev_c(p, \bar{v} : \bar{w})$ is the event generated by this call of c.

If in h_c there is a further event generated by p, then it has to be $ev_c(p, \bar{v} : \bar{w})$. Otherwise the computation stops signalling a failure. In that case the particular h_c is not realized by π_i (and $\tilde{\pi_i}$) which might happen due to over specification.

For $h_c = h_0 \circ h_1'$, where in h_0 there is no event generated by p and $fst(h_1') = ev_c(p, \bar{v} : \bar{w})$, h_1' is scanned for a matching answer event. If there is no such answer, then the computation stops with $h_{out} \circ h$ as the final output history and *mode* being set to *stop*.

In all these cases the procedure simulating the external call leaves the result parameters untouched since they are not needed anymore.

In the following paragraph we make use of the predicate $Match\_ev(e_1, e_2)$ which checks whether the event e_2 represents a matching answer for the event e_1. $\langle p, SOS, Sread(sid, 1)\rangle$ and $\langle SOS, p, Succ\_sread(1, "c")\rangle$ represents an example for a pair of matching messages. These two messages represent the call of a socket read on the socket identified by *sid* and the corresponding answer message containing the read string c (see also chapter 3).

For $h_1' = h_1 \circ h_2$, where $rst(h_1)$ contains no answer matching $fst(h_1') = fst(h_1) = ev_c(p, \bar{v} : \bar{w})$ and $fst(h_2) = e$ such that $Match\_ev(ev_c(p, \bar{v} : \bar{w}), e)$ the procedure returns values for the result parameters according to the message contained in e and the computation of $\tilde{\alpha_i}$ continues with $rst(h_2)$ as the new remaining history and $h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]$ as the new current output.

The above mentioned analysis of the current history h with respect to an external call $c(\bar{\tau} : \bar{z}, res)$ of p is given by $parse_c(p, h, \bar{v}, \bar{w}) \in His \times His \times His$, where again $\bar{v}$ and $\bar{w}$ are the values of $\bar{\tau}$ and $\bar{z}$, respectively.

$$
\begin{aligned}
&parse_c(p, h, \bar{v}, \bar{w}) = (h_0, h_1, h_2) \leftrightarrow (h = h_0 \circ h_1 \circ h_2 \wedge \\
&\quad ev_c(p, \bar{v} : \bar{w}) \notin h_0 \wedge \\
&\quad (h_1 \neq [] \rightarrow (fst(h_1) = ev_c(p, \bar{v} : \bar{w}) \wedge \\
&\qquad\qquad \forall e \in rst(h_1).\neg Match\_ev(ev_c(p, \bar{v} : \bar{w}), e))) \wedge \\
&\quad (h_2 \neq [] \rightarrow Match\_ev(ev_c(p, \bar{v} : \bar{w}), fst(h_2))))
\end{aligned}
$$

The body of the procedure is given below.

$$
\begin{aligned}
body_c :\equiv\ & \textbf{declare } h_0 := parse_c(p, h_c, \bar{x}, \bar{y})_{\cdot 0}; \\
& \qquad h_1 := parse_c(p, h_c, \bar{x}, \bar{y})_{\cdot 1}; \\
& \qquad h_2 := parse_c(p, h_c, \bar{x}, \bar{y})_{\cdot 2} \\
& \textbf{begin} \\
& \textbf{if } mode \neq fin \textbf{ then skip else} \\
& \ \textbf{if } \exists e \in h_0.Gen(p, e) \textbf{ then } mode := fail \textbf{ else} \\
& \ \ \textbf{if } h_2 = [] \textbf{ then } mode := stop; \\
& \ \ \ \textbf{if } h_1 = [] \textbf{ then} \\
& \ \ \ \ h_{out} := h_{out} \circ h \circ [ev_c(p, \bar{x}, \bar{y})]; \\
& \ \ \ \ h_c := [] \textbf{ fi} \\
& \ \ \ \textbf{else} \\
& \ \ \ \ \ h_{out} := h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]; \\
& \ \ \ \ \ h_c := rst(h_2); y_0 := ret\_val_c^1(fst(h_2)) \\
& \ \ \ \ \ \ldots \\
& \ \ \ \ \ y_{n-1} := ret\_val_c^{n-1}(fst(h_2)) \\
& \ \ \ \ \ res := ret\_res_c(fst(h_2)) \\
& \ \ \ \textbf{fi} \quad \textbf{fi} \quad \textbf{fi}
\end{aligned}
$$

where $Gen(p, e)$ is true if the event e is generated by the process represented by p. The function $ret\_val_c^i(e)$ extracts the result parameters from the event e and $ret\_res_c(e)$ returns the result value of the corresponding external call c.

Before the execution of $\tilde{\alpha_i}$ is started the begin of an active thread of p has to be determined by the start procedure, or if p has been started by a fork call, the start of the ancestor's thread who was running the very beginning of the program has to be found. If there is no p-thread executing π_i (or no suitable ancestor), then the given input history is returned as output and *mode* is set to *term*.

The procedure that simulates the start of a process π_i is given by the declaration $start\_i(: p, h_c, h\_out, mode) \Leftarrow body_{start\_i}$. It *parses* the given history h according to the definition of $Proc$.

$$\begin{aligned} parse_{start\_i}(p, h_c) = (h_0, h_1) \leftrightarrow h = h_0 \circ h_1 \wedge \\ (h_1 \neq [] \rightarrow (Create(i, p, fst(h_1)) \wedge \\ \forall e \in rst(h_1).\neg Term(i, p, e))) \end{aligned}$$

The procedure body then is given below.

$$\begin{aligned} body_{start\_i} :\equiv & \textbf{ declare } ah := anc(i, p, h); h_0 := []; h_1 := []; \\ & \textbf{ begin} \\ & \textbf{ if } ah = [] \textbf{ then } mode := term; h_{out} := h_c; \textbf{ else} \\ & \quad h_1 := fst(ah); h_0 := \Delta(h_c, h_1); \\ & \quad h_c := rst(h_1); h_{out} := h_0 \circ [fst(h_1)]; \\ & \quad p := get\_rec(fst(h_1)) \\ & \textbf{ fi} \end{aligned}$$

Finally we need a stop procedure $stop(: h_{out}, h, mode) \Leftarrow body_{stop}$ that finalizes the simulation. It restores the original history by appending the remaining h to h_{out}. Note that in those cases where a new (final) event was generated h_c will be []. If we have reached the end of $\tilde{\alpha_i}$, indicated by $mode = fin$, we check whether according to the remaining history something needs to be done a signal the result by setting *mode* to fin or *term*, respectively. This information is needed for decomposing verification problems. The body of the stop procedure is then given as

$$\begin{aligned} body_{stop} :\equiv & \; h_{out} := h_{out} \circ h; \\ & \textbf{ if } mode = fin \wedge \forall e \in h.\neg Gen(p, e) \textbf{ then } mode := term \textbf{ fi} \end{aligned}$$

Whenever *mode* is changed (to $m \in \{stop, fail\}$) by a procedure simulating an external call the rest of $\tilde{\alpha_i}$ has to be skipped. This is achieved by adding a kind of guards to while loops and (possibly) recursive procedures. In addition h, h_{out}, and *mode* have to be passed as arguments to the procedures declared in δ_i.

For declarations we have

$$\emptyset \mapsto_\sim \emptyset$$

$$q(\bar{x} : \bar{y}) \Leftarrow \beta \ , \ \delta \mapsto_\sim \tilde{q}(\bar{x} : \bar{y}, h_c, h_{out}, mode) \Leftarrow \textbf{if } mode \neq fin \textbf{ then skip else } \tilde{\beta} \textbf{ fi} \ , \ \widetilde{\delta}$$

Commands are modified as follows.

$$\textbf{skip} \mapsto_\sim \textbf{skip}$$

$$x := \tau \mapsto_\sim x := \tau$$

$$\alpha_0; \alpha_1 \mapsto_\sim \widetilde{\alpha_0}; \widetilde{\alpha_1}$$

$$\textbf{if } \epsilon \textbf{ then } \alpha_0 \textbf{ else } \alpha_1 \textbf{ fi} \mapsto_\sim \textbf{if } \epsilon \textbf{ then } \widetilde{\alpha_0} \textbf{ else } \widetilde{\alpha_1} \textbf{ fi}$$

$$\textbf{while } \epsilon \textbf{ do } \alpha \textbf{ od} \mapsto_\sim \textbf{while } \epsilon \wedge mode \neq fin \textbf{ do } \widetilde{\alpha} \textbf{ od}$$

$$q(\bar{\tau} : \bar{z}) \mapsto_\sim \tilde{q}(\bar{\tau} : \bar{z}, h_c, h_{out}, mode)$$

$$c(\bar{\tau} : \bar{z}, res) \mapsto_\sim c\_sim(p, \bar{\tau} : \bar{z}, res, h_c, h_{out}, mode)$$

5 Conclusion and Related Work

Our work was motivated by the problem of verifying application level programs with certain communication primitives given a complex formal model for distributed instances of an operating system that are connected by a network and include machines for the interpretation of C0 programs. Instead of working directly on the model we have introduced sets of (finite) sequences H of communication events to specify open distributed systems. This is a particular kind of stream specification as discussed in [9]. However, we restrict ourselves to prefix closed sets expressing safety properties.

Apart from abstracting from the local state spaces these histories were used for a reduction to local verification problems (compositionality). In case of application level programs this reduction is provided by a uniform transformation $\pi \mapsto \tilde{\pi}$. Once and for all we had to establish a relation to the original model by a simulation theorem. This proof is based on the Verisoft C0 interpreter and is the only semantic consideration necessary in our approach. As opposed to the Hoare-style proof system presented in [10] we do not need a new semantic interpretation for $\tilde{\pi}$.

An earlier attempt to map the Verisoft model to the temporal framework implemented in VSE failed. Temporal verification techniques, like those mentioned in [11], turned out not to be appropriate for large programs (more than 7.500 lines) and complex internal data structures. The results of the verification of $\tilde{\pi}$ can be viewed as properties of a (total!) function $approx_\pi : Hist \to Hist$ that (possibly) extends a given history h by a further event (step). Turning this function into an action of TLA, [12], (manipulating variables for histories) allows for a temporal treatment of liveness and reactivity. The underlying safety assertion, $\Box h \in H$ has already been established *outside* temporal logic.

Despite many technical differences the basic idea for the reduction to $\tilde{\pi}$ is similar to the use of (prefix closed) time diagrams in [10]. In particular this holds for the distinction between events caused by $\tilde{\pi}$ and those caused by the environment. However, neither do we need a special semantics for the transformed program $\tilde{\pi}$ nor an explicit composition theorem for the concurrent execution of programs. Composition as well as the inference of additional properties is done entirely at the level of history specifications H. For the latter we might use functions that extract (as a first-order data structure) for example "the last e-mail that was sent" from a given history h.

References

1. The Verisoft Consortium: The verisoft project http://www.verisoft.de.
2. Cheikhrouhou, L., Rock, G., Stephan, W., Schwan, M., Lassmann, G.: Verifying a chip-card-based biometric identification protocol in vse. In: The 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP 2006). (2006)
3. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security **6** (1998) 85–128
4. Mantel, H.: Information flow control and applications — bridging a gap. Lecture Notes in Computer Science **2021** (2001)
5. de Roever, W.P.: Concurrency Verification – Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)
6. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In Hurd, J., Melham, T.F., eds.: Proceedings of the TPHOLs 05, Springer (2005) 1–16
7. Schirmer, N.: A verification environment for sequential imperative programs in isabelle/hol. In Baader, F., Voronkov, A., eds.: Proceedings of the LPAR 04, Springer (2005) 398–414
8. Hutter, D., Langenstein, B., Sengler, C., Siekmann, J.H., Stephan, W., Wolpers, A.: Deduction in the Verification Support Environment (VSE). In: Proceedings FME96. Volume 1051., Springer (1996)
9. Broy, M., Stolen, K.: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces and Refinement. Springer (2001)
10. de Boer, F.S., Hannemann, U., de Roever, W.P.: Hoare-style compositional proof systems for reactive shared variable concurency. In: Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science, London, UK, Springer-Verlag (1997) 267–283
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer (1991)
12. Lamport, L.: Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2003)

Symbolic Fault Injection

Daniel Larsson and Reiner Hähnle

Chalmers University of Technology, Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden, {reiner,danla}@chalmers.se

Abstract. Fault tolerance mechanisms are a key ingredient of dependable systems. In particular, software-implemented hardware fault tolerance (SIHFT) is gaining in popularity, because of its cost efficiency and flexibility. Fault tolerance mechanisms are often validated using fault injection, comprising a variety of techniques for introducing faults into a system. Traditional fault injection techniques, however, lack coverage guarantees and may fail to activate enough injected faults. In this paper we present a new approach called *symbolic fault injection* which is targeted at validation of SIHFT mechanisms and is based on the concept of symbolic execution of programs. It can be seen as the extension of a formal technique for formal program verification that makes it possible to evaluate the consequences of *all* possible faults (of a certain kind) in given memory locations for *all* possible system inputs. This makes it possible to formally prove properties of fault tolerance mechanisms.

1 Introduction

One of the most common and important ways to ensure the dependability of computer systems and to analyse their fault tolerance mechanisms is *fault injection*. This includes a variety of techniques for deliberately introducing faults into a computer system and monitoring the system's behavior in the presence of these faults.

From a methodological point of view, fault injection is an experimental technique similar to *testing*: individual runs of a system are executed with input test data which in the case of fault injection is additionally instrumented with specific locations for fault injection.

During the last decade *formal methods* were increasingly used to ensure the absence of (or to detect the presence of) permanent software faults. Formal techniques such as model checking [11], extended static checking [10], and deductive verification [5] are able to find bugs or verify safety properties of industrial software. The common advantage of these methods is that they are *symbolic* and work on a logic-based representation of software properties. In consequence, one single correctness proof of a system property represents system runs for *all* admissible inputs.

Formal methods are not a replacement, but a complement of conventional software testing, because they typically work on source or bytecode and do not cover faults in machine code, compilers, or runtime environments. In order to verify the latter, testing is indispensable. Formal methods are also too expensive (or unsuitable) to cover all aspects of a system such as the user interface or I/O. For safety-critical segments of source code, on the other hand, formal verification is an increasingly cost efficient and extremely reliable alternative to testing [24, 2].

In the existing approaches to formal software verification, a program is proven to have certain properties under the assumption that no hardware faults occur (that are not detected and handled by the hardware or the operating system) during execution of the program. In other words, nothing is proven about the fault tolerance of the program. This is clearly a limitation of formal methods in the area of safety-critical systems.

The main contribution of this paper is to show that symbolic techniques such as formal software verification can be extended to symbolic analysis of fault injection and to software fault tolerance mechanisms. In contrast to conventional fault injection, this establishes the possibility to *prove* that a given fault tolerance mechanism achieves the desired behaviour for all inputs and *all modeled faults*. In particular, it is possible to guarantee that all injected faults are actually activated. Even when a fault tolerance mechanism fails to contain the injected faults and, therefore, a proof is not possible, the verification system allows to investigate the effects of the introduced faults. The method presented in this paper is applicable to *node-level* fault tolerance mechanisms, i.e., mechanisms for achieving fault tolerance withing a single node (or in an non-distributed environment).

To the best of our knowledge, this is the first presentation of a formal verification framework for software-implemented hardware fault tolerance (SIHFT). Related work is discussed in Sect. 7. We call our approach *symbolic fault injection*. It is based on *symbolic execution* of source code [8], a technique where program execution is simulated using symbolic representations rather than actual values for input data, and the effect of program execution is expressed as logical expressions over these symbols.

The central idea is to inject symbolic faults (representing whole classes of concrete faults) during symbolic execution which then reflects the consequences of the injected faults. This has been prototypically implemented and evaluated in a tool for formal verification of (JAVA) software, the KeY [1, 5] tool.

The paper is organized as follows: in the following section we review SIHFT, our main target application. We discuss our fault model in Sect. 3. Readers unfamiliar with formal verification find the necessary background in Sect. 4. The core of the paper is Sect. 5, where we explain how symbolic fault injection is modeled and implemented in the logic of the verification system. In Sect. 6 we present a case study showing the potential of symbolic fault injection. We close with related work, a discussion of the achieved results and future work.

2 SIHFT

It is impossible to guarantee that a given computer system is free of faults. Even using the best available techniques for manufacturing hardware components, the best available processes for the design of the hardware and the software, and the best available techniques for testing a system, it may still contain defects. Moreover, it is

impossible to guarantee that no transient faults occur during operation of the computer system. Therefore, in order to construct dependable computer systems we need to equip them with mechanisms for detecting and recovering from faults. Faults can be classified into hardware and software faults. An orthogonal classification divides faults into transient, intermittent, and permanent faults. In this paper the focus is on *transient hardware faults*, specifically, bit-flips in data memory locations.

Fault tolerance mechanisms can be based on hardware (for example, redundant components) or on software. From a cost perspective it is often beneficial to use software-implemented fault tolerance whenever possible, because (i) commercial, standardized components can be used; (ii) hardware redundancy can be avoided; and (iii) high flexibility can be obtained.

The scenario, where mechanisms for handling hardware faults are implemented in software, is called SIHFT (Software-Implemented Hardware Fault Tolerance) (for example, [7]). Common SIHFT techniques include assertions, algorithm-based fault tolerance (ABFT), control-flow checking, and data duplication and comparison. Other examples are checksum algorithms like CRC (Cyclic Redundancy Check). We apply our symbolic technique to the latter in Sect. 6 below.

Our method for formal verification in the presence of faults operates on the source code level of high-level programming languages and hence is restricted to software mechanisms. The type of faults we tried to emulate so far are transient hardware faults, specifically, bit-flips in the data area of memory (this is not an inherent limitation: other kinds of faults could be modelled, see Sect. 8). SIHFT is a natural target application for our method. The fact that hardware-implemented fault detection mechanisms rarely detect faults in the data area of the memory [4] further motivates the choice of our fault model which is described in detail in the following section.

3 Fault Injection and Fault Model

The purpose of using fault injection is to provoke the occurrence of errors in a system in order to validate the system's dependability. Errors occur too infrequently during normal operation of a computer system to be able to perform such a validation within reasonable time.

Existing fault injection approaches can be classified into hardware-implemented and software-implemented fault injection (SWIFI). Examples of the first are techniques, where integrated circuits are exposed to heavy-ion radiation [16] or electromagnetic interference [15], and the injection of faults directly on the pins of an integrated circuit [3]. Software-implemented fault injection can be further classified into prototype-based [9] and simulation-based [14] fault injection. In the first case the actual computer system to be validated (or a prototype thereof) is running while faults are introduced into the system through software. In the second case a simulation of the system is used when the faults are introduced.

A fault injection approach is based on a *fault model* which specifies the exact kind of faults to be injected or emulated. In many fault injection approaches/tools only single bit-flips are used since they are considered to be efficient in revealing dependability weaknesses.

The next question to consider is *where* and *when* the faults are to be injected. For the purpose of evaluating the relative effectiveness of fault tolerance mechanisms on different levels (hardware level, operating system level, and application level), it is useful to be able to inject faults, with high precision, in specific parts of the hardware. For example, faults might be injected into the MMU (Memory Management Unit) to evaluate a system's robustness against this kind of faults. When one is mainly interested in evaluating the mechanisms on the application level, it is often sufficient to inject faults in memory. It is also useful to be able to control when faults are injected, i.e., how the faults are *triggered*. Fault injections can be related to a certain instruction being executed or a memory location being manipulated, or a fault can be injected after a specified time.

The major weakness of conventional fault injection techniques is their lack of coverage. For example, to evaluate the effect of a fault in a given memory location, typically one bit or a few bits are flipped. But there is no guarantee that these particular faults will actually exhibit any defects present in the fault handling hardware or software. In other words, using fault injection nothing is *proved* about the fault tolerance property of a system. Similar to ordinary testing, fault injection can only show the presence of defects, not their absence.

Conventional fault injection techniques also suffer from other problems. Hardware-implemented techniques require special hardware which is very difficult—sometimes even impossible—to design for modern processors [9]. These techniques are also not easily ported to other platforms or expanded to new classes of faults. In the case of techniques using heavy-ion radiation or electromagnetic interference it is difficult to exactly trigger the time and location of a fault injection [13]. One source of problems with existing SWIFI tools is the target system monitoring for detecting the activation of faults and for investigating the exact effects of the faults [9]. Software solutions for monitoring have an undesired impact on the target system behavior. Moreover, the analysis of the huge amounts of monitor data is both difficult and time-consuming. Another problem with existing SWIFI techniques is that a large proportion of the injected faults are not activated, for example, faults injected into unused memory locations or faults placed in registers before the registers are written to [4].

The approach presented in this paper can be characterized as software-implemented, simulation-based fault injection. In the experiments performed, the simulation consists of the machinery available in the KeY tool for performing symbolic program execution. The fault model so far consists of bit-flips. There are no inherent restrictions on the types of faults we can emulate; if a certain hardware part is explicitly simulated as part of the verification, it is possible to emulate the effects of faults in that part. It would also be quite easy to emulate software faults. Our approach works

on the source code level. We emulate transient bit-level faults in the data segment of memory by manipulating the variables in the program, and we relate fault injection to pseudo-statements instrumented into the source code and triggered during symbolic execution.

4 Formal Methods

Formal methods comprise a wide range of techniques including black box approaches such as specification-only or specification-based testcase generation. Here we concentrate on *formal verification* of software. Among the various approaches to formal software verification [1, 10, 11] we single out verification by symbolic program execution [8], because of its compatibility with the analysis of propagation of injected faults through a program.

Our implementation platform is the formal software verification tool KeY [1, 5]. In its current version it can handle most of sequential JAVA and there is ongoing work to deal with concurrency [18] and for support of the C language. KeY takes as input a JAVA program (source code) and a formal specification of that program. The combination of the program and the specification is combined into a *proof obligation* expressed in JAVA Dynamic Logic (JAVADL). JAVADL is a typed first-order logic (FOL) extended with a dynamic part that can handle JAVA programs.

The idea of verification by symbolic program execution is to use logic in order to represent all possible values of locations in a program and to track their value updates during execution. We illustrate the main ideas by an example.

```
public class C {
  static int a,b;

  public static void swap()
  {
    b = a - b;
    a = a - b;
    b = a + b;
  };
}
```

Fig. 1. The `swap()` method.

The `swap()` method in Fig. 1 exchanges the values of the fields `a` and `b` of class `C` without the need for a temporary variable. Symbolic execution of the method would start by assigning symbolic integer values i and j to fields `C.a` and `C.b`, respectively. Since we want to analyse `swap()` for arbitrary values i and j we quantify universally

over them. A total correctness assertion in the program logic used in the KeY system [1] looks then as follows:

$$\forall \textbf{int}\ i;\ \forall \textbf{int}\ j;\ (\{\texttt{C.a} := i\}\{\texttt{C.b} := j\} \\ \langle \texttt{C.swap();}\rangle(\texttt{C.a} \doteq j\ \&\ \texttt{C.b} \doteq i)) \tag{1}$$

The universal quantifiers range over integer variables i and j that are assigned to the fields `C.a` and `C.b` as symbolic initial values. Variables i and j are so-called *rigid* variables whose value cannot be changed during the execution of a program (roughly corresponding to **final** locations in JAVA). For a compilable JAVA program `p`, a formula of the form "$\langle$`p`$\rangle$post" expresses that every run of `p` with the current initial values terminates normally and afterwards the postcondition "post" is true. In other words, `p` is *totally correct* with respect to the given postcondition. If one is merely interested in *partial correctness*, the $[\,]$-operator can be used instead: "[`p`]post" expresses that *if* `p` terminates normally *then* "post" will be true in the end state. In formula (1) the postcondition expresses that the initial values of the fields `C.a` and `C.b` have been swapped by stating that the value of `C.a` now is equal to initial value j and `C.b` is equal to i (we use the symbol $\doteq$ to distinguish between equality in formulas and assignment statements). In this way it is possible to formally specify the functionality of a given method.

The translation into a logical framework makes it possible to reason formally about a program. A universally quantified formula such as (1) is valid if and only if the formula

$$\{\texttt{C.a} := i\}\{\texttt{C.b} := j\}\langle \texttt{C.swap();}\rangle(\texttt{C.a} \doteq j\ \&\ \texttt{C.b} \doteq i) \tag{2}$$

is true for any possible interpretation of i and j. The expressions in curly brackets are called state *update*. Let $\mathcal{U} = \{\mathrm{loc} := \mathrm{val}\}$ be such an update, where loc is a location (program variable, field or array access) and val is a side-effect free expression. The semantics of an updated formula $\mathcal{U}\phi$ is to change the environment relative to which ϕ is evaluated in such a way that the value of loc becomes val and everything else is unchanged. Hence, the meaning of formula (2) is: whenever `swap()` is started in an initial state where `C.a` has value i and `C.b` has value j, then `swap()` terminates normally and afterwards the contents of the fields `C.a` and `C.b` is swapped.

The logic JAVADL used in the KeY system provides symbolic execution rules for any formula of the form "$\mathcal{U}\langle\xi;\omega\rangle\mathrm{post}$", where ξ is a single JAVA statement and ω the remaining program. ξ is called the first *active statement* of the program, i.e., the statement the rule operates on. JAVADL rules such as (3) can be seen as an operational semantics of the JAVA language. Application of rules can then be thought of as symbolic code execution. A program is verified by executing its code symbolically and then checking that the FOL conditions after execution is finished are valid. During proof search rules are applied from bottom to top. From the old proof obligation (conclusion), new proof obligations are derived (premisses).

We give some examples of JAVADL symbolic execution rules. Updates are used to record the effect of assignment statements during symbolic execution:

$$\frac{\vdash \{\mathtt{v} := \mathtt{e}\}\langle\omega\rangle\phi}{\vdash \langle \mathtt{v = e;}\ \omega\rangle\phi} \tag{3}$$

The symbol $\vdash$ stands for derivability. The rule says that in order to derive the formula in the *conclusion* (on bottom) it is sufficient to derive the formula in the single *premiss* (on top). The idea is to simply replace an assignment with a state update. This rule can only be used if `e` is a side-effect free JAVA expression. Otherwise, other rules have to be applied first to evaluate `e` and the resulting state changes must be added to the update.

The effect of an update is not computed until a program has been completely (symbolically) executed. For example, after expanding the method body of `C.swap()` and symbolic execution of the first two statements we obtain the following intermediate result:

$$\vdash \{\mathtt{C.a} := j\}\{\mathtt{C.b} := i - j\}\langle\mathtt{method\text{-}frame(C()): b = a + b;}\rangle (\mathtt{C.a} \doteq j \,\&\, \mathtt{C.b} \doteq i)$$

During method expansion a *method frame*, which records the receiver of the invocation result and marks the boundaries of the inlined implementation, was created. The updates of `C.a` and `C.b` reflect the assignment statements that have been executed already. After executing the last statement and returning from the method call the code has been fully executed. The subgoal reached at this point is similar to $\vdash \{\mathtt{C.a} := j\}\{\mathtt{C.b} := i\}\langle\rangle(\mathtt{C.a} \doteq j \,\&\, \mathtt{C.b} \doteq i)$, where the updates are followed by the empty program. Only now updates are applied to the postcondition which results in the trivial subgoal $\vdash j \doteq j \,\&\, i \doteq i$.

Below is another rule example, namely the rule for the **if - else** statement which has two premisses. The rule is slightly simplified.

$$\frac{\mathtt{b} \doteq TRUE \vdash \langle \mathtt{p}\,\omega\rangle\phi \qquad\qquad \mathtt{b} \doteq FALSE \vdash \langle \mathtt{q}\,\omega\rangle\phi}{\vdash \langle \textbf{if}\ \ \mathtt{(b)\ p}\ \ \textbf{else}\ \ \mathtt{q;}\ \omega\rangle\phi}$$

This rule shows that in contrast to normal program execution, in symbolic execution even of sequential programs it is sometimes necessary to branch the execution path. This happens whenever it is impossible to determine the value of an expression that has an influence on the control flow. This is the case for conditionals, switch statements, and polymorphic method calls, among others. The rule above is applicable if `b` is an expression without side effects, otherwise other rules need to be applied first.

A problem occurs with loops and recursive method calls. If the loop bound is finite and known, then one can simply unwind the loop a suitable number of times. But in general one needs to apply an induction argument or an invariant rule to prove properties about programs that contain unbounded loops. Both approaches tend to be expensive, because they require human interaction. The automation of induction proofs for imperative programs is an area of active research [26].

5 Symbolic Fault Analysis

5.1 General Idea

Our plan is to extend the approach to formal verification of software sketched in the previous section with the concept of symbolic fault injection. This makes it possible to prove that a program with software-based fault tolerance mechanisms ensures certain properties even in the presence of faults. Alternatively, one may calculate the consequences of the introduced faults in terms of strongest postconditions. The realization is based on the following two ideas:

- The source code is instrumented with pseudo-instructions of the form "`inject(location);`" that are placed where the faults are to be injected. The argument `location` is the name of a memory location (local variable, field access, formal parameter, etc.) visible at this point in the program. This makes it possible to handle (symbolic) fault injection *uniformly* by symbolic code execution.
- Symbolic fault injection is realized by extending the symbolic execution mechanism with suitable rules for the `inject` pseudo-instructions.

The examples given below are in JAVA since the current version of KeY handles JAVA, but the principles given hold for any imperative language.

An injection of a symbolic fault causes a change in the JavaDL representation of the symbolic program state, and this state change corresponds to the consequences of *all* the concrete faults that can appear during program execution and that are instances of the symbolic fault.

Assume that we want to emulate the effect of *all possible* bit-flips in the memory location that corresponds to a given variable. First we need to clarify what is meant by "all possible" bit-flips. Is it the effect of all possible *single* bit-flips or all possible combinations of an arbitrary number of bit-flips (in the same memory location)? Considering a JAVA **int** (represented by 32 bits): there are 32 different possible outcomes in the first case, but 2^{32} in the second. Obviously, when trying to prove properties about algorithms that can detect bit-flips, it is essential to distinguish between single bit-flips (or, perhaps, a fixed, small number) and an arbitrary number of bit-flips. For example, the CRC algorithm discussed in Sect. 6 can detect situations where one or a few bits are flipped. Trying to prove the fault detection capability of such an algorithm using the "arbitrary number of bit-flips" semantics of the `inject` statement will not succeed. However, in other situations it might be desirable and possible to prove properties for an arbitrary number of bit-flips. Our solution is to use two different inject statements: `inject(location)` means that an arbitrary number of bits in the memory location will be flipped, while `inject1(location)` means that a single bit is flipped. To model a situation where a fixed number n of bits in a location is flipped, `inject1` is simply applied at that location n times.

Another important question is whether the "no change" case is included in the meaning of the `inject`/`inject1` statements, i.e. whether the property we are trying to prove should also hold for the case where no bits are flipped. As will become apparent in Sect. 6, sometimes a semantics *not* including the "no change" case is needed. Below we introduce different flavours of rules for handling the `inject`/`inject1` statements covering both cases: one including the "no change" case and the other one excluding it.

5.2 Rules

We need to add new rules to the JAVADL calculus that handle the `inject` pseudo-instructions. The rules for the cases when the `location` argument of `inject` has type **boolean** or **byte** are below. In the case of **boolean** typed variables there is no need to distinguish between single bit-flips and an arbitrary number of bit-flips as they hold only one bit, however, the distinction between inclusion and exclusion of the "no change" case is relevant, and the two rules are presented below (inclusion of the "no change" case is indicated by appending *NC* to the rule name).

$$\text{booleanNC}\ \frac{\vdash \{\mathsf{b} := \textbf{true}\}\langle\omega\rangle\phi \quad \vdash \{\mathsf{b} := \textbf{false}\}\langle\omega\rangle\phi}{\vdash \langle \texttt{inject(b);}\omega\rangle\phi}$$

$$\text{boolean}\ \frac{\vdash \{\mathsf{b} := \texttt{!b}\}\langle\omega\rangle\phi}{\vdash \langle \texttt{inject(b);}\omega\rangle\phi} \tag{4}$$

The first rule splits symbolic execution into two paths, where in exactly one of them b is unchanged and in the other it is complemented. The second rule continues symbolic execution with the value of b complemented. Next we show the rule for an arbitrary number of bit-flips in a **byte** variable. Only the "no change" version is shown.

$$\text{byteNC}\ \frac{\vdash \forall\, \textbf{byte}\ i; \{\mathsf{b} := i\}\langle\omega\rangle\phi}{\vdash \langle \texttt{inject(b);}\omega\rangle\phi} \tag{5}$$

In this case the memory location can contain any **byte** value after the injection. This means that whatever program property that should be proved has to be proved for all values of this variable. In logical terms it means that a universal quantification has to be introduced. We do this by quantifying over a new logical variable i followed by an update that assigns the value of i to the location b.

Finally, the rules for the `inject1` statement on **byte**s and arrays of **byte**s are presented. Only the rules excluding the "no change" case is shown.

$$\text{byte1}\ \frac{\vdash \forall\, \textbf{int}\ j;\ 0 \leq j \leq 7 \rightarrow \{\mathsf{b} := \mathsf{b} \hat{}\, (1 \ll j)\}\langle\omega\rangle\phi}{\vdash \langle \texttt{inject1(b);}\ \omega\rangle\phi} \tag{6}$$

After injection, the memory location can contain any value resulting from flipping exactly one bit in b. The intuition behind the rule is that the variable is xored with the masks, 00000001, 00000010,..., 10000000 respectively. The rule for arrays of **byte**s,

$$\text{byteArr1}\ \frac{\vdash \forall \mathsf{int}\, i;\ \forall \mathsf{int}\, j;\ 0 \leq i < \mathsf{a.length}\ \&\ 0 \leq j \leq 7 \rightarrow \{\mathsf{a[i]} := \mathsf{a[i]\hat{}}(1 \ll j)\}\langle\omega\rangle\phi}{\vdash \langle \mathsf{inject1(a)};\ \omega\rangle\phi}$$

Fig. 2. The inject1 rule for byte arrays.

pictured in Fig. 2, is similar but includes universal quantification over the array elements. The rule shown is a bit simplified since the real rule has to take the possibility of a **null** reference into account. We created analogous rules for the other primitive JAVA types, which are not presented here.

5.3 Example: Verification

We proceed to show by example how the rules for the pseudo-instruction `inject` are used in practice. The examples are based on a simple JAVA class shown below.

```
class MyBoolean {
    boolean v;
    boolean myOr(boolean b) {
        boolean t=b;
        inject(t);
        return t||v;
    }
}
```

`MyBoolean` can be viewed as a wrapper for **boolean** primitive valuess. It contains a **boolean** field `v` that holds the value of a `MyBoolean` instance. It also has a method, `myOr`, with obvious meaning. (The temporary variable `t` is unnecessary for the behavior of `myOr`. It is added for the presentations of the proofs below, as it makes it possible to refer to the original value of the parameter `b` in an easy way.) The interesting point is the `inject` statement that injects a fault into the **boolean** argument before the return value is computed. Attempts to prove that `myOr` has certain correctness properties even in the presence of faults are shown below.

Symbolic execution and first-order logic reasoning as implemented in KeY is used in the proof attempts. Note the use of rule (4) for `inject(t)` (marked with an asterisk on the right in Fig. 3 and Fig. 4). In the first example, shown in Fig. 3, an attempt is made to prove that the method still has the semantics expected from logical or (the variable `result` in the proof stands for the return value of `myOr`): the postcondition states that the return value of `myOr` is true if and only if one of the arguments is true. This is impossible to prove due to the injected fault. We get four different branches in the proof, one for each combination of values in field `v` and parameter `b`. All branches must be proven in order to show the property. Due to space restrictions we only show one of the branches that are impossible to prove, indicated by Γ in the antecedent which abbreviates "`v` $\doteq$ **false** $\&$ `b` $\doteq$ **true**". The proof tree is shown in Fig. 3. As expected, we end up with a sequent which is impossible to prove valid.

$$\begin{array}{c}
\Gamma \vdash false \\ \hline
\Gamma \vdash false \leftrightarrow true \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \leftrightarrow (\mathsf{false} \doteq \mathsf{true} \lor \mathsf{true} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{false}\}\langle\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{v}\}\langle\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \langle \mathsf{return\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle \mathsf{return\ t\ ?\ true : v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle \mathsf{return\ t} \parallel \mathsf{v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{true}\}\langle \mathsf{inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \quad * \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{b}\}\langle \mathsf{inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \langle \mathsf{boolean\ t{=}b;\ inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true}))
\end{array}$$

Fig. 3. Failed proof attempt: correctness property of MyBoolean::myOr(). One of four branches in the proof: Γ stands for "$\mathsf{v} \doteq \mathsf{false}\ \&\ \mathsf{b} \doteq \mathsf{true}$".

Now consider an attempt to prove something weaker, namely that `myOr()` returns **true** whenever the field v was **true**. Again, only one of the four proof branches is shown, but all are provable. We use Γ in the same way as above. The proof tree is in Fig. 4. We end up with a sequent that is valid indicating provability. We showed the

$$\begin{array}{c}
[\Gamma \vdash true] \\ \hline
\Gamma \vdash false \rightarrow false \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \rightarrow \mathsf{false} \doteq \mathsf{true} \\ \hline
\Gamma \vdash \mathsf{v} \doteq \mathsf{true} \rightarrow \mathsf{false} \doteq \mathsf{true} \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{false}\}\langle\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{v}\}\langle\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \langle \mathsf{return\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle \mathsf{return\ t\ ?\ true : v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle \mathsf{return\ t} \parallel \mathsf{v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{true}\}\langle \mathsf{inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \quad * \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{b}\}\langle \mathsf{inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \langle \mathsf{boolean\ t{=}b;\ inject(t);\ return\ t} \parallel \mathsf{v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true})
\end{array}$$

Fig. 4. Successful proof: weakened correctness property of MyBoolean::myOr(). One of four branches in the proof: Γ stands for "$\mathsf{v} \doteq \mathsf{false}\ \&\ \mathsf{b} \doteq \mathsf{true}$".

formal proofs in some detail in order to give an impression how symbolic execution of code and injected faults works. All proofs, respectively, proof attempts are created by the KeY prover within fractions of a second and fully automatically.

5.4 Example: Calculating Strongest Postcondition

Besides proving that a program has certain properties in the presence of faults it is possible to calculate the consequences of a fault in terms of strongest postconditions. Below is a simple program containing an `inject` statement for which we calculate the strongest postcondition.

```
int aMethod() {
   int i = 0;
   inject(i);
   return i;
}
```

The calculation of the strongest postcondition of the program is shown below. Note that an `inject` rule for **int** type variables similar to (5) is used.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \exists\, \textbf{int}\ k; result \doteq k}{\vdash \forall\, \textbf{int}\ j; \{result{:=}j\}\langle\rangle ?}}{\vdash \forall\, \textbf{int}\ j; \{\mathrm{i}{:=}j\}\{result{:=}\mathrm{i}\}\langle\rangle ?}}{\vdash \forall\, \textbf{int}\ j; \{\mathrm{i}{:=}j\}\langle \textbf{return i;}\rangle ?}}{\vdash \forall\, \textbf{int}\ j; \{\mathrm{i}{:=}0\}\{\mathrm{i}{:=}j\}\langle \textbf{return i;}\rangle ?}}{\vdash \{\mathrm{i}{:=}0\}\langle \texttt{inject(i);}\ \textbf{return i;}\rangle ?}}{\vdash \langle \textbf{int i=0;}\ \texttt{inject(i);}\ \textbf{return i;}\rangle ?}$$

The symbolic execution tells us that after a fault injection in variable `i` the return value can be any **int** value. The example is trivial and not very interesting in itself but illustrates the idea: the symbolic execution makes it possible to analyse the consequences of faults for all admissible inputs.

6 Case Study

We illustrate the application of symbolic fault injection to a realistic fault handling mechanism: an implementation of the widely used CRC (Cyclic Redundancy Check) algorithm. CRC is a fault detection algorithm: it calculates a checksum on a block of data. This checksum is typically appended to the data block before it is transmitted and the receiver is then able to determine whether the data has been corrupted. The basic idea behind CRC is to treat the block of data as a binary representation of an integer and then to divide this integer with a predetermined divisor. The remainder of the division becomes the checksum. The kind of division used is not the one found in standard arithmetic but in so-called polynomial arithmetic. The property that makes CRC so useful is that it minimizes the possibility that several bit-flips "even out" with respect to the checksum and therefore go undetected. The algorithm fully utilizes the number of bits used to represent the checksum. By choosing the divisor (also called *poly*) carefully, the algorithm can detect all single bit-flips, all two-bit errors (up to a

certain size of the block of data), all errors where an odd number of bits are flipped, and so-called burst errors (where a number of adjacent bits are flipped) up to a certain number of bits depending on the size of the divisor.[1]

We describe briefly the implementation of the algorithm. We cannot use JAVA's built-in division operation, because the block of data, viewed as an integer, in general is far too big to store in a register; also, we need to use polynomial arithmetic. Therefore, the data is fed step by step to a division register while the required operations are applied to its content. In its simplest and least efficient implementation of the algorithm the data is shifted bit by bit, while the most commonly used implementation shifts the data one "register length" at a time and uses a lookup table. Below is an example of a table-driven implementation in JAVA generated by the "CRC generator".[2] The block of data is here represented by an array of **byte**s, which is given as parameter `buf` to the program. The method returns the computed CRC value.

```
static byte compute(byte[] buf) {
    int count = buf.length;
    byte reg = (byte)0x0;
    while (count > 0) {
        byte elem = buf[buf.length-count];
        int t = ((int)(reg^elem)&0xff);
        reg <<= 8;
        reg ^= table[t];
        count--;
    }
    return reg;
}
```

The array `table` in the program above refers to an array of 256 precomputed **byte**s that allows to perform the division, shifting the block of data one **byte** at the time (instead of one *bit* at the time). It would be useful to prove formally that this method has certain properties. Even though the theory behind the CRC algorithm is well known, there is no guarantee that this particular *implementation* of the algorithm is free from errors, in particular, since concrete algorithms are synthesized by a program generator based on several parameters.

In the following we document an attempt to formally prove that the implementation above detects all single bit-flips. Detecting single bit-flips is something we expect even the most simple checksum algorithms to manage, but nevertheless it is valuable to formally prove that a given CRC implementation actually does this. More precisely, the following should be proved. Assume one arbitrary **byte** array of arbitrary length. This array is duplicated, an arbitrary single bit-flip in *one* of the arrays is performed, and then CRC checksums for both arrays are computed. The two checksums should

[1] For the algorithm and possible implementations see http://www.repairfaq.org/filipg/LINK/F_crc_v3.html.
[2] http://members.cox.net/tonedef71/body_jcrcgen.htm#output

differ and the first step to prove this property is to create the test harness below. It is a modified version of the CRC implementation with the following changes.

- All variables (except `count`), including the **byte** array constituting the input to the algorithm, are duplicated. All statements acting on these variables are also duplicated.
- An `inject1` statement (described in Sect. 5.2) is added that injects a bit-flip fault in one of the input arrays.
- Instead of returning the CRC checksum, this modified version returns the comparison of the two computed CRC checksums in form of a **boolean** value. That is, the method returns **true** if the two checksums are equal (the fault is *not* detected) and **false** otherwise.

```
static byte crcTest(byte[] buf1, byte[] buf2) {
    inject1(buf2);
    int count = buf1.length;
    byte reg1 = (byte)0x0;
    byte reg2 = (byte)0x0;
    while (count > 0) {
        byte e1 = buf1[buf1.length-count];
        byte e2 = buf2[buf2.length-count];
        int t1 = ((int)(reg1^e1)&0xff);
        int t2 = ((int)(reg2^e2)&0xff);
        reg1 <<= 8;
        reg2 <<= 8;
        reg1 ^= table[t1];
        reg2 ^= table[t2];
        count--;
    }
    return (reg1 == reg2);;
}
```

The reason for modifying the original program is to facilitate the proving process. It could be argued that this modification might change the behavior of the original program in an unintended way, e.g., that the checksum calculated for the non-faulty array is not equal to the checksum calculated for the same array using the original program. For this program, however, it is fairly easy to see that this is not the case. In case of doubt, it is possible to formally prove this.

The next step is to express formally the property this program should have. The proof obligation expressed in JAVADL is presented below (7). The variable *result* stands for the return value of the method. For sake of clarity some parts dealing with potential `NullPointerException`s and similar are omitted. The variables $msg1lv$ and $msg2lv$ are used to quantify over all possible values of the input message

blocks `msg1` and `msg2`. The precondition states that (the reference variables) `msg1` and `msg2` do not refer to the same array, but that the arrays are identical. Note that the $[\,]$-operator is used, i.e., proving termination is not part of the proof obligation (see Sect. 4). The reason is that this makes it possible to apply a loop invariant rule; see discussion below. Termination has been proven separately.

$$\begin{aligned}
&\forall \textbf{byte[]}\ msg1lv;\ \forall \textbf{byte[]}\ msg2lv;\\
&\quad (msg1lv \neq msg2lv\ \&\ msg1lv.length \doteq msg2lv.length\\
&\qquad \&\ \forall\ \textbf{int}\ j; (j \geq 0\ \&\ j < msg1lv.length\ \rightarrow msg1lv[j] \doteq msg2lv[j])\\
&\rightarrow \{\texttt{msg1} := msg1lv\}\ \{\texttt{msg2} := msg2lv\}\\
&\qquad [\texttt{Crc.crcTest(msg1,msg2);}](result \doteq \textbf{false}))
\end{aligned} \tag{7}$$

When trying to prove properties about programs containing unbounded loops (like the `crcTest()` method), then either a loop invariant or induction must be used. We chose to use an invariant of which a simplified version is shown below (8). The part of the program preceding the while statement was symbolically executed. Then a loop invariant rule was applied, which includes providing the actual invariant.

$$\begin{aligned}
&(inject_ar_elem < \texttt{msg1.length} - \texttt{count} \rightarrow \texttt{reg1} \neq \texttt{reg2})\\
\&\ &(inject_ar_elem \geq \texttt{msg1.length} - \texttt{count} \rightarrow \texttt{reg1} \doteq \texttt{reg2})
\end{aligned} \tag{8}$$

The integer $inject\_ar\_elem$ results from the execution of the `inject1` statement and refers to the element in the `msg2` array where the fault is injected. It is a skolem constant originating from the universal quantification over the array elements used in the rule for `inject1` (see Fig. 2). In other words, the invariant has to hold for all possible values of $inject\_ar\_elem$.

After application of the loop invariant rule, the proof splits into three branches: one where it must be proven that the invariant holds before the while statement starts to execute, one where one needs to prove that (8) is indeed an invariant of the loop body provided that the guard holds, and one where it must be shown that the proof obligation (7) follows from the invariant and the negated loop condition. We proved all three cases using KeY. Here is the summary of the overall proof that (7) holds after execution of `crcTest()`.

1. The part of the program preceding the while statement was symbolically executed. This is automatic.
2. The loop invariant rule was applied and the loop invariant (8) manually provided.
3. KeY's automatic application of rules was restarted which resulted in about 2500 rule applications in less than 8 minutes. The result was 13 open goals, i.e., branches of the proof that could not be proved automatically. In all open goals, the program part of the proof obligation was completely (symbolically) executed, i.e., only program-free FOL formulas remained.

4. The 13 open goals were proved by manual rule application. This is tedious, but straightforward.

In summary, we proved formally that a certain implementation of the CRC fault detection algorithm discovers *all possible* single bit-flips in an arbitrary **byte** array. The proving activity was to a large extent automatic. It is straightforward to apply the same methodology to related algorithms, now that a valid pattern of loop invariants has been established.

7 Related Work

In [19] an approach for evaluating the system reliability with respect to bit-flip errors using model-checking principles is presented. This is applied to a software-implemented mechanism that detect errors corrupting the control flow, a signature analysis technique. A control flow graph of the considered generic target program, which is a representative model over a general class of all possible applications (i.e., it covers all possible fault scenarios with respect to the fault model) is created. The model checker SPIN is applied to the model and the fault detection mechanism in order to investigate whether the detection mechanism detects *all* faults. Since the description of the approach in the paper is highly dependent on the signature analysis technique, it is hard to see to which degree it is possible to generalize it to other kinds of fault tolerance mechanisms. Clearly, a necessary requirement is the ability to construct an abstract model of an imagined target program that covers all possible fault scenarios with respect to a considered fault model.

In several papers one specific fault tolerance mechanism is formally verified. In most cases these are system-level (in contrast to node-level) mechanisms for distributed systems, e.g., the TTP Group Membership Algorithm. Some examples of this line of work follow: in a paper by Bernardeschi et al. [6], a fault tolerance mechanism called “inter-consistency mechanism”, a component of an architecture for embedded safety-critical systems, was formally specified and verified using the model checker JACK. The properties the mechanism should satisfy were expressed as temporal logic formulas and the model of the mechanism was given as a Labelled Transition System (LTS) which included faults that could affect the behavior of the mechanism itself. In [25], a model of a startup algorithm for the Time-Triggered Architecture was proven to have certain safety, liveness, and timeliness properties using model checking (the SAL toolset from SRI). It is claimed that all possible failure modes were examined, an approach the authors call “exhaustive fault simulation”. A fault-tolerant group membership algorithm of TTP was formally specified and verified using a diagrammatic representation of the algorithm. The work is described in [21]. The PVS theorem prover was used for the verification. Clock synchronization algorithms are an important part of distributed dependable real-time systems. The paper [23] describes a formal generic theory of clock synchronization algorithms (that extracts the commonalities of specific algorithms) in the form of parameterized PVS theories. Several

concrete algorithms are formally verified with PVS using this framework. In [22], different aspects of formal verification of algorithms for critical systems are discussed. As an example, the Interactive Convergence Algorithm (ICA) is proved to have certain properties using the EHDM system.

What distinguishes our approach from the mentioned papers is that we present *a general framework* for analysis and formal verification of *executable implementations* (in contrast to abstract models) of fault tolerance mechanisms.

Finally, it should be mentioned that symbolic fault injection has been used in a method for calculating the *coverage factor*, i.e., the proportion of faults that are actually handled by a system [17].

8 Discussion and Future Work

Traditional fault injection techniques suffer from a number of drawbacks, notably lack of coverage and failure to activate injected faults. In this paper we presented a new approach called symbolic fault injection which is targeted at validation of SIHFT mechanisms and is based on the concept of symbolic execution of programs.

It is an analytic approach in contrast to experimental evaluation done in conventional fault injection. With symbolic fault injection it becomes possible to emulate the consequences of *all possible* faults in a certain memory location. All injected faults are also activated, which is in general not the case with conventional fault injection. Symbolic fault injection based on formal verification can be expensive and requires some expertise, but this is also the case with conventional fault injection. In particular, to investigate the consequences of an injected fault is difficult and time consuming when using conventional methods.

Our fault model so far consists of single bit-flips in memory locations. This is achieved through pseudo-instructions added to the source code together with rules for handling these pseudo-instructions during symbolic execution. We implemented a prototype of our method based on the formal software verification tool KeY. We showed the viability of the approach by proving that a CRC implementation detects all possible single bit-flips. Clearly, this is only a proof of concept and a proper evaluation with realistic industrial software needs to be done.

An argument that is often raised against the usage of formal methods is that formal specifications of systems are normally not available and are very time consuming to create. Note that our approach is useful even without the availability of a formal specification, because it can be used to compute the symbolic effect of faults in the form of strongest postconditions (Sect. 5.4).

Limitations Our current implementation suffers from a number of limitations: since our fault injection technique is simulation-based, no real-time properties can be evaluated with it. Formal verification of real-time systems is still an area of research. So far we have not considered the injection of faults in pointer- or reference-variables,

and we have only looked at faults in the data area of the memory, not the code area. We also inherit a number of limitations from the underlying verification system. The most important are that the program logic of the KeY system at the moment cannot handle multi-threaded programs or floating point data types. Research that overcomes the first of these is under way [18]. A practical limitation is that full automation can only be achieved when bounds on loops and recursion are finite and concrete. Otherwise, induction or invariant rules with expensive user interaction is required. Again, research to improve this situation is under way [20].

Future Work It would be interesting to generalize our approach to different fault models and fault trigger mechanisms. This is principally possible by parameterizing the total correctness modality $\langle \mathtt{p} \rangle \phi$ with additional parameters for a trigger condition t, a symbolic fault `inject`, and a reset expression $\mathcal{R}$, where t is a FOL formula, `inject` is an inject pseudo-statement, and $\mathcal{R}$ is a state update (Sect. 4). The semantics of the formula $\langle \mathtt{p} \mid t \mid \mathtt{inject} \mid \mathcal{R} \rangle \phi$ is the same as $\langle \mathtt{p} \rangle \phi$, but before symbolic execution of the next active statement it is checked whether t holds and `inject` is inserted whenever it does. In addition, after symbolic execution of each statement the update $\mathcal{R}$ is added to the current environment. If $\mathcal{R}$ is something like $\{\mathtt{b} := \textbf{false}\}$, then it easy to emulate a stuck-at-zero fault.

We think that it is attractive to integrate our technique into a framework for design and assessment of dependable software such as Hiller et. al.'s [12]. Part of this framework uses fault injection for error propagation analysis to find the locations in the software where it is most effective to place fault handling mechanisms. We think that our technique could be very useful in the error propagation analysis.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.
2. P. Amey. Correctness by construction: Better can also be cheaper. *CrossTalk Magazine, The Journal of Defense Software Engineering*, pages 24–28, March 2002.
3. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, 1990.
4. J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.*, 52(9):1115–1133, 2003.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
6. C. Bernardeschi, A. Fantechi, and S. Gnesi. Formal validation of the GUARDS inter-consistency mechanism. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Intl. Conf. on Computer Safety, Security and Reliability (SAFECOMP)*, pages 420–430, 1999.
7. P. Bernardi, L. Bolzani, M. S. Rebaudengo, M. S. Reorda, and M. Violante. An integrated approach for increasing the soft-error detection capabilities in SoCs processors. In *Intl. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 445–453, 2005.
8. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
9. J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *Software Engineering*, 24(2):125–136, 1998.

10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN Conf. on Progr. Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
11. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
12. M. Hiller, A. Jhumka, and N. Suri. PROPANE: an environment for examining the propagation of errors in software. In *Proc. ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, pages 81–85. ACM Press, 2002.
13. M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
14. E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proc. 24th Intl. Symp. on Fault Tolerant Computing, (FTCS-24), IEEE, Austin/TX, USA*, pages 66–75, 1994.
15. J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, pages 267–287, Urbana-Champaign, USA, September 1995. IEEE Computer Society.
16. J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, 1994.
17. L. T. Klauwer. Application of Formal Methods to Fault Injection and Coverage Factor Calculation. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Göteborg, Sweden, 2006.
18. V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. Verification of JCSP programs. *Concurrent Systems Engineering*, 63:203–218, 2005.
19. B. Nicolescu, Y. Savaria, E. Aboulhamid, and R. Velazco. On the use of model checking for the verification of a dynamic signature monitoring approach. *IEEE Transactions on Nuclear Science*, 52:1555–1561, Oct. 2005.
20. O. Olsson and A. Wallenburg. Customised induction rules for proving correctness of imperative programs. In B. Beckert and B. Aichernig, editors, *Proc. Software Engineering and Formal Methods, Koblenz, Germany*, pages 180–189. IEEE Press, 2005.
21. H. Pfeifer. Formal verification of the TTP Group Membership algorithm. In *Proc. FIP TC6 WG6.1 Joint Intl. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, pages 3–18. Kluwer, 2000.
22. J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Trans. Softw. Eng.*, 19(1):13–23, 1993.
23. D. Schwier and F. W. von Henke. Mechanical verification of clock synchronization algorithms. In *Proc. 5th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS, pages 262–271. Springer-Verlag, 1998.
24. A. E. K. Sobel and M. R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320, 2002.
25. W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The Intl. Conf. on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.
26. A. Wallenburg. Proving by induction. In B. Beckert, R. Hähnle, and P. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*, pages 453–480. Springer-Verlag, 2006.

A Termination Checker for Isabelle Hoare Logic

Jia Meng[1], Lawrence C. Paulson[2], and Gerwin Klein[3]

[1] National ICT Australia jia.meng@nicta.com.au
[2] Computer Laboratory, University of Cambridge lp15@cam.ac.uk
[3] National ICT Australia gerwin.klein@nicta.com

Abstract. Hoare logic is widely used for software specification and verification. Frequently we need to prove the total correctness of a program: to prove that the program not only satisfies its pre- and post-conditions but also terminates. We have implemented a termination checker for Isabelle's Hoare logic. The tool can be used as an oracle, where Isabelle accepts its claim of termination. The tool can also be used as an Isabelle method for proving the entire total correctness specification. For many loop structures, verifying the tool's termination claim within Isabelle is essentially automatic.

1 Introduction

For many critical systems, such as operating systems kernels, testing is not adequate to ensure correctness. Formal methods have become popular in research as well as industry. There are several approaches to verifying program correctness. For example, Hoare logic has been used to specify a program's pre- and post-conditions; using logical deduction, one can prove the program meets its specification. A program that fails to terminate satisfies its post-condition by default, so *total correctness* requires a proof that the program terminates.

Total correctness proofs are complicated and require a lot of human effort. A different approach is to focus on the termination property of a program, which can usually be automated with some human assistance. An example is the `Terminator` program developed at Microsoft Research [3, 8], which checks whether a C program terminates.

A proof of termination is useful, but it does not guarantee that a program does what it is supposed to do. Full specification and verification is still what we are aiming at, and the most important formalism for that purpose is Hoare logic. Our current work is part of a larger project to verify the functional correctness of the L4 operating system micro kernel. An automatic termination checker can reduce manual work. Since `Terminator` is not publicly available, we have implemented a termination checker in the spirit of `Terminator` and have integrated this tool into Isabelle [6]. The tool can be used by Isabelle's Hoare logic to prove total correctness specifications. In this paper, we concentrate on termination of `WHILE` constructs.

Although our termination checker and `Terminator` are based on the same technology, we have a different emphasis. In addition to implementing the termination tool, we also investigate how we can have the tool's results used by

Isabelle's Hoare logic. The termination tool is based on model checking and it returns a set of well-founded (WF) relations for each cyclic path. However, a total correctness specification in Hoare logic requires one single WF relation (the variant) for each looping construct. An explicit variant represents evidence of termination, but this variant is not given as a result by either `Terminator` or our tool, which are both based on the model checking technology. Much of our investigations concern how to make the results from the termination tool usable to Isabelle. As far as we know, our tool is the first integration of a model-checking-based termination tool with an interactive prover, and our work is the first integration of the terminator technology to Hoare logic.

In addition, we invented an optimization for our termination checker (§4.3) to meet our particular requirements on the generated WF relations. Finally, Podelski and Rybalchenko [8] have proved the mathematical theory behind `Terminator`. In order that we can use the tool in Isabelle, we have had to formalize their proofs in Isabelle, and this is the first formalization of the proofs in any interactive prover.

Another method for termination analysis is to translate imperative programs into functional programs so that one can prove the termination property of an imperative program by proving its functional counterpart terminates [1]. However, we decided to adopt another approach, which is based on disjunctively well-founded relations. This approach is comparatively novel and is a promising method that has been applied to full C programs.

Paper outline. We first present background information on Isabelle's Hoare logic and termination requirements of programs (§2). We then describe how we have implemented our termination checker (§3). Subsequently, we illustrate the two approaches we have used to integrate the termination checker into Isabelle (§4 and §5). In order to show how we can use our termination checker with Isabelle, we give some examples (§6). Finally, we conclude the paper (§7).

2 Isabelle and Termination Properties

2.1 Hoare logic in Isabelle

We base our work on Norbert Schirmer's implementation of Hoare logic in Isabelle/HOL [9]. Schirmer has designed a small but expressive imperative language, called SIMPL, with recursive procedures. He defines an operational semantics for SIMPL and derives a sound and complete Hoare logic. The Hoare logic implementation includes an automated verification condition generator (`vcg`). In other work [10], we have provided an Isabelle front-end for mapping C into SIMPL.

Since SIMPL treats procedures, the Hoare triple format that we show in examples later also mentions procedure environments, Γ. Partial correctness

is written $\Gamma \vdash \{\!|P|\!\} C \{\!|Q|\!\}$; total correctness is written with subscript t, as $\Gamma \vdash_t \{\!|P|\!\} C \{\!|Q|\!\}$. Partial correctness means if C is executed in a state where P is true, and *if* C terminates, then it will end in a state where Q is true. Total correctness includes a termination requirement: if C is executed in a state satisfying P, then it will terminate in a state satisfying Q.

For proving total correctness, there are two approaches. One approach is to separate partial correctness from termination, as with the following rule

$$\frac{\Gamma \vdash \{\!|P|\!\}\ C\ \{\!|Q|\!\} \quad \Gamma \vdash_t \{\!|P|\!\}\ C\ \{\!|\top|\!\}}{\Gamma \vdash_t \{\!|P|\!\}\ C\ \{\!|Q|\!\}}$$

where $\top$ is logical truth. The second premise says that the program C started in a state satisfying P will terminate. If we assert it without giving any evidence, we have implemented an oracle that accepts an external claim of termination. Here we follow the convention that unmentioned variables do not change their values.

The second approach is to use Hoare logic rules for total correctness. These rely on the concept of a well-founded (WF) relation: one that has no infinite descending chains. In this paper we do not consider the recursive procedure call of SIMPL, which makes `WHILE` the only looping construct. Schirmer uses sets of states to formalize all predicates, such as pre- and post-conditions and the loop's boolean expression. For the verification condition generator, a typical `WHILE` statement is annotated with an invariant, which is again a set of states, and a variant, which is some WF relation. This means the `WHILE`-rule for total correctness in Schirmer's Hoare logic is the following:

$$\frac{\begin{array}{c}\forall \sigma.\ \Gamma \vdash_t \{\!|\ \{\sigma\} \cap I \cap b\ |\!\}\ C\ \{\!|\ \{t \mid (t,\sigma) \in V\} \cap I\ |\!\} \\ P \subseteq I \qquad I \cap -b \subseteq Q \qquad \texttt{wf}\ V\end{array}}{\Gamma \vdash_t \{\!|P|\!\}\ \texttt{WHILE}\ b\ \texttt{INV}\ I\ \texttt{VAR}\ V\ \texttt{DO}\ C\ \{\!|Q|\!\}}$$

The first premise fixes the pre-state σ and requires that the loop body C decreases the variant while maintaining the invariant I. The set $\{t \mid (t,\sigma) \in V\}$ consists of all post-states t such that (t,σ) are in the variant V. If V is a WF relation, the loop must terminate. This approach is good for generating full termination proofs, because the variant gives explicit evidence for termination.

The rest of this paper shows how we can integrate externals tools into Isabelle/HOL to produce the variants mentioned above and how to prove them well-founded. This reduces the manual proof effort for total correctness goals, either by oracle or by proof fully verified in Isabelle/HOL.

2.2 Termination Properties

Before we explain the termination properties, it is helpful to have a brief review of the transition system that is usually used as an abstraction of programming languages. For more information, we refer readers to the book by Manna [5].

Each program has an associated *transition system*. A transition system consists of four parts: Π, Σ, R and Θ.

- Π is a finite set of *state variables*, which consists of program variables that appear in the program statements and also control variables, such as the program counter (PC).
- Σ is a set of *states*. Each state s is an interpretation of Π, which assigns values to each variable in Π. For example, $x\ s$ is the value of variable x in the state s.
- R is a finite set of *transitions*. For a deterministic program, a transition τ is a function $\Sigma \to \Sigma$. If a transition τ leads a state s to another state s' (written as $s \xrightarrow{\tau} s'$), then s' is reachable from s. We say that s and s' are pre- and post-states of τ.
- Finally Θ is the *initial condition*, which is a set of initial states

Each transition τ is characterized by its *transition relation* ρ_τ, where $(s', s) \in \rho_\tau$ if and only if s' is reachable from s by τ. In addition, we can extract one or more transition relations from each program statement. For example, if we have a `WHILE` statement at program location L as `WHILE(x > 0){x = x -1;}`, then its transition relations are $\{(s', s) \mid\ PC\ s = L \wedge x\ s > 0 \wedge x\ s' = x\ s - 1 \wedge PC\ s' = L\}$ and $\{(s', s) \mid\ PC\ s = L \wedge x\ s \leq 0 \wedge x\ s' = x\ s \wedge PC\ s' = M\}$, where M is the exit location of the `WHILE` construct.

This set notation for transition relations is sometimes abbreviated by a logical formula: the value of a variable in the pre-state is represented directly by the variable name and the value in the post-state is represented by the primed version of the variable name. When the PC value is understood from the context, its pre- and post-values are also ignored. For example, the first transition relation above can be abbreviated to $x > 0 \wedge x' = x - 1$.

A *computation* is a possibly infinite sequence of states $s_0, s_1 \ldots$, such that s_0 is in an initial state and each s_{i+1} is reachable from s_i via some transition.

A *path* in a transition system is a sequence of transitions $\pi = \tau_1 \ldots \tau_n$, such that the PC's value in the post-state of τ_i is the same as the value in the pre-state of τ_{i+1}. There is a path transition relation for each path, which is just the relational composition of each consecutive transition relation involved in this path. The path above is cyclic if the PC value in the pre-state of τ_1 is the same as the value in the post-state of τ_n.

A program is terminating if there is no infinite computation. Theoretically, this can be proved by showing that there is a WF relation T, such that each consecutive pair of states s_i and s_{i+1} has $(s_{i+1}, s_i) \in T$. However, it is often

too difficult to find one single WF relation T for this purpose. Podelski and Rybalchenko [8] have proved that it is sufficient to find a finite set of WF relations $T = \{T_1, \ldots, T_n\}$ to show the program is terminating, provided we can prove $R_I^+ \subseteq T_1 \cup \ldots \cup T_n$, where R is the program's transition relation, R^+ is the transitive closure of R and R_I^+ is a reachable subset of R^+. This means for each reachable path, its path transition relation must be a member of some T_i of T.

Since each non-cyclic path induces a WF relation, to show the program is terminating, we only need to show there is a WF relation T_i for each reachable cyclic-path. Cook et al. have shown [3] that program termination checking can be translated into program reachability analysis. For this to work, auxiliary program variables are introduced and the relations between them and the original variables are established to mimic the transition relations of the program. A designated program location `ERROR` is used: it is only reachable if there is no WF relation T_k such that the post-state s' and pre-state s of a cyclic path's transition relation has $(s', s) \in T_k$. If this happens, one can try to find a WF relation for that cyclic path. If no WF relation can be found, then the program is reported as possibily non-terminating. If a WF relation can be generated for the path, then the process continues with other cyclic paths, until `ERROR` is really not reachable, which means all cyclic paths are covered by some WF relations.

2.3 Using a Termination Tool for Generating Variants

Although we can use the termination tool to generate WF relations for all cyclic paths, these relations cannot be used as variants for Hoare logic.

First, there may be nested `WHILE` constructs. The variant V for the outer `WHILE` has to be one single WF relation so that the pre-state s and the post-state s' of the body transition relation satisfy $(s', s) \in V$, regardless how many times the inner `WHILE` are entered and whether they are entered at all. This means V has to satisfy multiple cyclic paths. We can use the termination tool to generate WF relations to cover all these cyclic paths, but the variant must be one single WF relation, and we believe that automatically combining multiple WF relations into one is impossible in general.

Second, even if a program has no nested `WHILE`s, there may be more than one (cyclic) path and hence more than one path transition relation between the start and the end of the `WHILE` loop. Each transition relation has to be a conjunction of positive atomic formulae, and each atomic formula is a mathematical assertion over state variables. This is because we use an external ranking function generator to synthesis WF relations and the tool only accepts a transition relation expressed as a conjunctive formula without negations. Consequently, if the guard of the `WHILE` has disjunctions or negations or the body has `IF` constructs, then we will have multiple path transition relations. We again need to combine multiple WF relations into one.

Instead of generating a single WF relation, we could investigate another approach that tries to construct a termination argument from these WF relations. However, our work aims to use the termination checker to support proofs performed in Isabelle Hoare logic, and the total correctness rule requires the variant as the termination argument.

3 Implementing a Termination Checker

We have implemented a termination checker that checks C programs involving integer variables, in the spirit of `Terminator` [3], in the sense that we check the termination of the entire program by generating one or more WF relations for its cyclic paths. Currently, we only use the tool to check cyclic paths produced by `WHILE` constructs and the tool does not handle pointers. Moreover, we use two external tools: `Blast` and `Rankfinder`.

3.1 Using `Blast` and `Rankfinder`

`Blast` [4] is a model checker for C programs. It checks that software behavior meets its specification using lazy abstraction. For our purpose, we use `Blast` to check if a designated location called `ERROR` is reachable. If `ERROR` is not reachable, then `Blast` reports the program is safe. If `ERROR` is reachable, then `Blast` returns a trace: a sequence of locations from the start of the program to `ERROR`.

Here, we are using the location `ERROR` to signal a possibly non-terminating cyclic path. If `ERROR` is not reachable, then there is no non-terminating cyclic path. However, if `Blast` reports a trace that leads to `ERROR`, then we can extract a cyclic path from it, and then we can examine if we can generate a WF relation to show the path is terminating, using `Rankfinder`.

`Rankfinder` [7] is a ranking function (a.k.a. measure function) generator. A ranking function is a decreasing function, with a lower bound. Given a transition relation τ, `Rankfinder` tries to synthesize a decreasing ranking function, with two parameters: an integer bound b and a positive integer d that is the minimum decrease of the ranking function during the transition relation. For example, if the transition relation is $x \geq 0 \wedge x' = x - 1$, then the ranking function from `Rankfinder` is x, the bound is 0 and minimum decrease is 1. Each ranking function F induces the well-founded relation

$$\{(s', s) \mid b\ s \leq F\ s \wedge F\ s' \leq F\ s - d\}$$

where s' and s are the post- and pre-states of the transition.

3.2 The Termination Checker

Our tool is closely integrated with Isabelle and it is called via an Isabelle invocation. It works as follows.

1. The C program embedded in SIMPL is extracted to generate a control flow graph. For better performance, we "compact" the flow graph so that only `WHILE` locations and the program entrance point are kept in the graph. All the remaining program locations are removed from the graph by joining the path transition relations. If the pre-condition P of the Hoare specification is non-empty (i.e. there are initial conditions on program variables), then we modify the flow graph to include the initial conditions. Subsequently, we examine each `WHILE` construct in turn, and the order in which we examine the `WHILE` constructs does not matter.
2. For each `WHILE` construct, writing its program location as L, check the termination of all cyclic paths that start from and finish at L:
   (a) Insert the already-generated WF relations for L into the C program and generate a text file, then call `Blast`. Initially no WF relation is generated, so nothing is inserted.
   (b) If `Blast` reports the program is safe, i.e. `ERROR` is not reachable, then move to the next `WHILE` construct. If `Blast` reports an error trace, then we extract the cyclic paths from the trace and calculate the reachable transition relations. We then call `Rankfinder` to generate a ranking function for each transition relation. If `Rankfinder` succeeds, then use the newly generated WF relations to modify the C program and re-run `Blast`. If `Rankfinder` cannot generate a well-founded relation for a transition relation, then the program is reported as possibly non-terminating.
3. If `ERROR` is no longer reachable, `Blast` will report the program is safe. We can then move on to the next `WHILE` construct if available.
4. If for each `WHILE` construct, its cyclic paths are reported to be terminating, then the entire program is terminating; the generated WF relations are also reported. Otherwise, the program is reported as possibly non-terminating.

4 Integrating the Termination Checker into Isabelle

Our termination checker has been used as a tool for Isabelle, both as an oracle and as a proof method. Isabelle's oracle mechanism accepts an external tool's result without verifying it. When used as a proof method, the result is used to create an Isabelle proof that is verified through Isabelle's kernel.

4.1 Integration as an Oracle

Recall that a total correctness goal can be proved separately as a partial correctness goal and a termination goal (§2.1). When used as an oracle, we only use the tool to prove the second subgoal, namely the program is terminating, if started from a state satisfying P. We do not need to generate variants in this case. Therefore, if the tool reports the program is terminating, the second subgoal is removed from the proof state.

4.2 Integration as an Isabelle Proof Method

Using the termination checker as an Isabelle oracle gives us a quick answer to whether the program is terminating. Of course, using the tool as an Isabelle proof method would yield greater confidence. This requires us to use the tool to generate a variant for each `WHILE` construct. Moreover, we would like the variant to be as simple as possible. The form of the variant generated depends on the complexity of the program, as well as the WF relations generated for `WHILE` constructs.

For this purpose, we divide the WF relations for each `WHILE` construct (W) into two sets: T_{in} is generated for cyclic paths that do not leave W and T_{out} is generated for cyclic paths that leave W and re-enter. This is because a variant for W is essentially a set of transition relations of paths that do *not* leave W. Therefore, if we define a variant using a relation that does not include any path that leaves W, we only need T_{in} for WF proofs. T_{out} is needed when we try to prove the entire program R is WF, since we need to prove R^+ is WF and R^+ contains paths that leave W.

In this section, we describe the form of the variants generated and will informally explain why they are variants and why they are WF. We will show some formal proofs in the next section.

Programs with No Nested Loops If a program has a single loop, then the variant generated only depends on the number of WF relations in T_{in} because we are not concerned about the paths leaving and re-entering the `WHILE` (W). There are two cases to consider.

First, if there is only one ranking function F generated, then we generate an Isabelle measure function M from it. The difference between F and M is that F involves integers whereas M uses natural numbers only, but this is easily dealt with.

Hoare logic requires us to prove that the loop body decreases the variant. More formally, V must be a set containing all the post- and pre-states pairs of the loop. We can indeed prove that the transition relations of each cyclic path starting and finishing at location W form a subset of V.

Second, if there is more than one ranking function in T_{in}, then we define the variant to be the intersection $R_L \cap I$ of the transition relation of the loop and the invariant of the loop. Frequently we can use R_L as the variant, which is weaker. The use of the invariant[1] is important when reachability becomes a concern (§4.4). As there is only one `WHILE` construct, R_L does not need to mention its PC value. As an example, consider the following C program.

```
WHILE (x > 0 || p > 2){x = x - 1; p = p - 2;}
```

[1] Currently, invariants are generated manually, but we plan to incorporate automatic invariant generation in the future.

Its R_L is

$$\{(s', s) \mid x\ s > 0 \wedge x\ s' = x\ s - 1 \wedge p\ s' = p\ s - 2 \vee \\ p\ s > 2 \wedge x\ s' = x\ s - 1 \wedge p\ s' = p\ s - 2\}$$

Obviously, the transition relation is a variant, and we can prove (see §5) that R_L is well-founded.

Programs with Nested Loops This is the complicated case. We generate the variant for each `WHILE` in turn. The form of the generated variant also depends on the complexity of the `WHILE` construct.

If there is only one ranking function F in T_{in}, then we generate its corresponding Isabelle measure function M as above.

If there are multiple ranking functions, then the variants are defined in terms of transition relations. Since there are multiple `WHILE` loops, the transition relation must mention PC values. There are two cases to consider.

First, if the `WHILE` construct with location L has no nested inner loop, then we define the variant to be

$$V = \mathit{fix\_pc}\ L\ (R_L \cap\ I)$$

where R_L, I are transition relation and invariant of the `WHILE` construct and the definition of *fix_pc* is

```
definition
  fix_pc :: "int ⇒((α * int) * (α * int)) set ⇒(α * α) set"
  where "fix_pc pc R = {(s',s). ((s',pc),(s,pc)) ∈R}"
```

The function *fix_pc* removes the dependence on the PC by restricting a relation to the given PC value. Again, R_L can replace $R_L \cap\ I$ sometimes.

Second, if the loop has inner nested loops, then we define its variant V as

$$V = \mathit{fix\_pc}\ L\ R^+$$

where R is the transition relation of the entire program. The formula on the right hand side is indeed a variant, because any path that starts from and finishes at the `WHILE` with location L must have its corresponding transition relation ρ as a subset of R^+, i.e. $\rho \subseteq R^+$. In addition, ρ must be a set containing tuples of the form $((s', pc'), (s, pc))$, where $pc = L$ and $pc' = L$. We will discuss the well-foundedness property of V in section 5.

4.3 An Optimization

The complexity of the WF proofs for variants largely depends on the number of WF relations our tool generates for the `WHILE` constructs. If a `WHILE` construct

is shown terminating using a set of WF relations, then it may also be possible to find another smaller set of WF relations that does the job. Suppose we have two WF relations T_1 and T_2, which show the termination of two cyclic paths π_1 and π_2 using $\rho_1 \subseteq T_1 \wedge \rho_2 \subseteq T_2$, where ρ_1 and ρ_2 are the path transition relations for the two paths. Then if we can generate a weaker WF relation S such that $\rho_1 \subseteq S \wedge \rho_2 \subseteq S$, then we can replace T_1 and T_2 by S. Please note, in general we cannot derive S by simply making a union of the two WF relations, since the result of the union may not be well-founded.

For each `WHILE` construct, our tool attempts to generate a WF relation when a possibly non-terminating cyclic path is found. Therefore, the order in which the WF relations are generated depends on the order in which these cyclic paths are detected. Since we use `Blast` to detect these cyclic paths, we have no control over its searching strategy and so we cannot ask for any specific paths to be reported first.

For a `WHILE` construct W that has one or more inner loops, some of W's cyclic paths (i.e. those start from and finish at W) enter inner loops (call them P_1) while some do not (call them P_2). It may happen that there is something decreasing along the execution of W, regardless whether any of W's inner loops are entered. More precisely, there may be a set T with one or more WF relations that cover paths from both P_1 and P_2. This means, we may be able to generate T without entering any of W's inner loops. We call this set of WF relations the *global* WF relations for W, since it exists regardless of the inner loops' behaviour.

Suppose there is one inner `WHILE` U of W, and the path transition relation from W to U is ρ_1, the path transition of U loop is ρ_u and the path transition relation from U back to W is $\neg b \wedge \rho_2$, where b is the guard of U, i.e. the condition when U is entered. The path transition relation from W back to W is

$$\rho = (\neg b \wedge \rho_2) \; o \; \rho_u^+ \; o \; \rho_1.$$

To generate the required T, we generate WF relations for $\rho_A = \rho_2 \; o \; \rho_1$. This path corresponds to a program W', which is W with U completely commented out. We do not generate WF relations for $\rho_B = (\neg b \wedge \rho_2) \; o \; \rho_1$. since this path simulates the effect that the guard of U is not true. Clearly ρ_A is weaker than ρ_B and if a WF relation can be used for ρ_A, then it can be used for ρ_B. Nevertheless, our aim is to have the generated WF relations to work for ρ as well; if the WF relation for ρ_B is too strong, then it may not work for ρ.

Of course, a given `WHILE` construct W may not have this global set T of WF relations. As a result, we still need to generate WF relations for all cyclic paths that enter inner loops. The advantage of generating T is that some relations in T may make it unnecessary to generate new WF relations for some cyclic paths, thereby reducing the number of relations generated. We have implemented this optimization in our termination checker.

4.4 An Issue of Reachability

As we have mentioned, the technique of termination checking works by ensuring all *reachable* cyclic paths are terminating. When we use `Blast` to check the reachability of the `ERROR` location, the notion of reachable cyclic path is already present implicitly with `Blast`. However, sometimes we need to express reachability explicitly, for two reasons.

The first one is for `Rankfinder` to generate WF relations. For example, we may have a program

```
y = 2; WHILE (x > 0){x = x - y;}
```

Without knowing $y > 0$, the transition relation of `WHILE`'s cyclic path is not well-founded and so `Rankfinder` will fail to generate a WF relation for it. To *strengthen* the transition relation, we note that $y > 0$ is an invariant and by adding it to the transition relation of the path, the new relation is indeed WF.

The second reason for including the reachability condition is to have a strong enough variant. There may be non-terminating cyclic paths that do not concern `Blast` since they are deemed to be unreachable, and so `Rankfinder` will not have to generate WF relations for them. However, when defining the variant, which are effectively the transition relations of paths, we need to incorporate in it the reachability condition so that unreachable paths are removed from it.

We have tried several ways of expressing this reachability requirement of loop variants. A simple way is to include the loop's invariant in the variant. For single-looped program, we define the variant as $R_L \cap I$. For an innermost loop, we define its variant as $\mathit{fix\_pc}\ L\ (R_L \cap I)$. We have used this method to prove problems that were not provable otherwise.

If the `WHILE` construct has nested inner loops, its variant can also be strengthened by adding invariants. To discover an invariant can require much thought, and ideally we would use an automatic tool for generating invariants. At present we are using no such tool and have decided to use $\mathit{fix\_pc}\ L\ R^+$ as the variant. This heuristic choice does not affect the soundness of our tool and will not affect the way the tool is used as an oracle. When the tool is used as a proof method, if a non-reachable non-terminating cyclic path is included, then a user will fail to prove that a (non-WF!) path transition is well-founded. This is a signal that the path may be in fact not reachable. The user can then make another attempt: either trust the oracle or try to strengthen the variant by finding a strong invariant of the entire program.

5 Proving Variants being Well-Founded

Having generated the required variants, we need to show their well-foundedness. Showing that the relation defined for V is a variant is a separate task, which requires the users to have found the correct invariants.

When the generated V has the form *measure* M, we can apply Isabelle's existing methods to show it is WF automatically. Otherwise, there are several possibilities:

- A single-loop program: the variant has the form $R_L \cap I$ or R_L.
- A multi-loop program: the variant has the form *fix_pc* L R, where R is either $R_L \cap I$ or R_L.
- In the most complicated case, the variant has the form *fix_pc* L R^+.

The proofs are based on *disjunctively well-founded relations* of Podelski and Rybalchenko [8]. A relation is disjunctively well-founded, if it is the union of finitely many well-founded relations. We need to formalize them for Isabelle proofs to work.

5.1 Proving that R and *fix_pc* L R are Well-Founded

These are the simpler of the three cases. In order to show that R is WF, we need to prove R^+ is WF. We have proved the following two essential theorems in Isabelle:

```
theorem union_wf_disj_wf1:
  "⟦⋀s. s ∈ R ⟹ wf s; r ⊆ ⋃R; finite R⟧ ⟹ disj_wf r"

theorem trans_disj_wf_implies_wf:
  "⟦trans r; disj_wf r⟧ ⟹ wf r"
```

The first theorem characterizes what it means for a relation r to be disjunctively well-founded (*disj_wf*). The second theorem states the crucial result that if a relation is both transitive and disjunctively WF, then it is WF. The Isabelle proof follows the informal argument [8] in using Ramsey's theorem. Using these two theorems, we can prove that R^+ is WF by proving $R^+ \subseteq \bigcup T$, where T is the set of WF relations the tool generates. We can prove that R is well-founded using another Isabelle lemma:

```
"wf (r⁺) ⟹ wf r"
```

Finally, we have used another theorem to prove variants of the form *fix_pc* L R.

```
theorem fix_pc_wf:
  "wf R ⟹ wf (fix_pc pc R)"
```

5.2 Proving that *fix_pc* L R^+ is Well-Founded

This is the complicated case and we have tried two approaches.

In the first approach, we tried to prove R^+ is WF by restricting attention to cyclic paths. In order to restrict R^+ to the transitions of cyclic paths, we have defined the constant *same_pc*:

```
definition
  same_pc :: "((α * int) * (α * int)) set"
  where "same_pc = {((s',pc'),(s,pc)). pc' = pc}"
```

Now $R^+ \cap$ *same_pc* denotes the subset of R^+ concerning cyclic paths. We need to prove that the relation *fix_pc* L R^+ is well-founded. It suffices to show

$$R^+ \cap \textit{same\_pc} \subseteq \bigcup T,$$

where T is the set of generated WF relations. We cannot prove this by induction because the *same_pc* property is not preserved from one transition to the next. To make this approach work, we need to identify an invariant S of R^+, such that we can prove $R^+ \subseteq S$ by induction and then prove $S \cap \textit{same\_pc} \subseteq \bigcup T$.

In the second approach, we attempted to prove that $R^+ \subseteq \bigcup T$ directly. Since R^+ includes both cyclic and non-cyclic paths, we tried modifying the tool to generate WF relations for non-cyclic paths as well. However, we found that $R^+ \subseteq \bigcup T$ still could not be proved by induction, apparently because the induction hypothesis was too weak: the set $\bigcup T$ was too large. We suspect that it is not practical to generate sufficiently strong WF relations for all non-cyclic paths because there are simply too many such paths.

5.3 Automation in WF Proofs

We have implemented several Isabelle proof methods to invoke the termination checker. When the tool generates all the required WF relations and shows the program is terminating, the goal will be modified with variants inserted. The generated WF relations are also proved automatically to be WF. There is also an option to insert the WF relations as theorems to the assumption of the proof goal for users to inspect.

After this step, we can use *vcg* followed by *auto* to finish the proof, if the variants are measure functions. For the variants of the forms R, $R \cap I$, *fix_pc* L R or *fix_pc* L $(R \cap I)$, we have implemented proof methods *check_wf_sw* and *check_wf_mw* to prove their well-foundedness automatically.

6 Examples and Experiments

Our termination tool is still in the early stage of development. At the moment, it does not support pointers or data structures, such as arrays. However, based on our current development, we can easily add in support for arrays, though pointers require more effort.

Users invoke the termination tool via Isabelle methods: *check_termination_oracle* uses the tool as an oracle; *check_termination* and *check_terminationH* construct variants and the latter uses the optimization(§4.3). If variants are generated, users

will need to apply a a few more Isabelle methods to prove the variants being WF. For this step to work, users usually also need to construct invariants. For example, in order to prove the lemma

```
lemma "Γ⊢t {|True|}
  WHILE (x >= 0) INV {|True|}
  DO
  x :== x + 1;; y :== 1;;
  (WHILE (y  <= x ∨ p > 0)  FIX X. INV {|x = X|}
  DO y :== y + 1;; p :== p - 2 OD);;
  x :== x - 2
  OD
  {|True|}"
```

we first apply *check_terminationH* to generate variants and then finish the proof with *vcg*, *auto* and *check_wf_mw*. *check_wf_mw* is an Isabelle method that we have implemented to automatically prove relations WF.

We carried out several experiments on our tool, with the results shown in Table 1. The experiments we ran mainly involved nested `WHILE` loops. The last example is terminating, but because of the lack of invariants, our tool reported it as non-terminating.

Result	*Remark*
proved by oracle	Fibonacci series with two nested `WHILE`s, no invariants
proved by oracle	Factorial with two nested `WHILE`s, no invariants
proved by oracle	Arithmetic exponentiation with two nested `WHILE`s, no invariants
proved by oracle	Example from [2]. Two nested `WHILE`s, no invariants
proved by method	Example lemma shown above
proved by method	Artificial example with two nested `WHILE`s, with invariants
proved by oracle	Artificial example with three nested `WHILE`s, no invariants
`Blast` failed to terminate	Arithmetic exponentiation with three nested `WHILE`s
`Blast` failed to terminate	Artificial example with three nested `WHILE`s
reported as non-terminating	A non-terminating program
reported as non-terminating	A terminating Euclid algorithm, no invariants

Table 1. Termination Tool Experiments

7 Conclusions

Automatic termination checking is too valuable a tool to reserve for the field of automated program analysis. Interactive program verifiers would like to benefit from as much automation as possible. We have shown how techniques designed for automatic termination checking can, in many cases, be incorporated in a Hoare logic proof, with the termination argument made explicit as variant functions in `WHILE` loops. The resulting subgoals can often be proved automatically,

using dedicated proof methods that we have written. In the most complicated loop structures, the information returned by the automated analysis does not appear to be detailed enough to allow the proof to be reproduced in Isabelle/HOL. To handle those cases, we have also implemented an interface to the tool that accepts the termination as an oracle.

In order to meet our objectives, we have formalized Podelski and Rybalchenko's theory of disjunctive well-foundedness [8] in Isabelle/HOL[2], and we have optimized the termination tool to eliminate redundant outputs that would complicated the proofs.

Acknowledgements. The research was funded by the L4.verified project of National ICT Australia.[3] Norbert Schirmer answered our questions about his implementation of Hoare logic, and Ranjit Jhala answered questions about `Blast`. In formalizing the theory of disjunctive well-foundedness, we used an Isabelle proof of Ramsey's theorem contributed by Tom Ridge.

References

1. Jürgen Brauburger and Jürgen Giesl. Approximating the domains of functional and imperative programs. *Sci. Comput. Program.*, 35(2):113–136, 1999.
2. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.
3. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426, New York, NY, USA, 2006. ACM Press.
4. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, Lecture Notes in Computer Science 2648, pages 235–239. Springer-Verlag, 2003.
5. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.
6. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.
7. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.
8. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In Harald Ganzinger, editor, *Proceedings of the Nineteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2004*, pages 32–41. IEEE Computer Society Press, July 2004.
9. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.
10. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.

[2] The proofs will appear in a future technical report.

[3] National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

The Heterogeneous Tool Set

Till Mossakowski[1], Christian Maeder[1], and Klaus Lüttich[2]

[1] DFKI Lab Bremen and Department of Computer Science, University of Bremen, Germany
[2] SFB/TR 8 and Department of Computer Science, University of Bremen, Germany

Abstract. Heterogeneous specification becomes more and more important because complex systems are often specified using multiple viewpoints, involving multiple formalisms. Moreover, a formal software development process may lead to a change of formalism during the development. However, current research in integrated formal methods only deals with ad-hoc integrations of different formalisms.
The heterogeneous tool set (HETS) is a parsing, static analysis and proof management tool combining various such tools for individual specification languages, thus providing a tool for heterogeneous multi-logic specification. HETS is based on a graph of logics and languages (formalized as so-called institutions), their tools, and their translations. This provides a clean semantics of heterogeneous specifications, as well as a corresponding proof calculus. For proof management, the calculus of development graphs (known from other large-scale proof management systems) has been adapted to heterogeneous specification. Development graphs provide an overview of the (heterogeneous) specification module hierarchy and the current proof state, and thus may be used for monitoring the overall correctness of a heterogeneous development.
We illustrate the approach with a sample heterogeneous proof proving the correctness of the composition table of a qualitative spatial calculus. The proof involves two different provers and logics: an automated first-order prover solving the vast majority of the goals, and an interactive higher-order prover used to prove a few bridge lemmas.

1 Introduction

> "As can be seen, a plethora of formalisms for the verification of programs, and, in particular, for the verification of concurrent programs has been proposed. *. . . there are good reasons to consider all the mentioned formalisms, and to use whichever one best suits the problem.*" [43] (italics in the original)

In the area of formal specification and logics used in computer science, numerous logics are in use:

- logics for specification of datatypes,
- process calculi and logics for the description of concurrent and reactive behaviour,
- logics for specifying security requirements and policies,
- logics for reasoning about space and time,
- description logics for knowledge bases in artificial intelligence/the semantic web,
- logics capturing the control of name spaces and administrative domains (e.g. the ambient calculus), etc.

Indeed, at present, it is not imaginable that a combination of all these (and other) logics would be feasible or even desirable — even if it existed, the combined formalism would lack manageability, if not become inconsistent. Often, even if a combined logic exists, for efficiency reasons, it is desirable to single out sublogics and study

translations between these (cf. e.g. [43]). Moreover, the occasional use of a more complex formalism should not destroy the benefits of *mainly* using a simpler formalism.

This means that for the specification of large systems, heterogeneous multi-logic specifications are needed, since complex problems have different aspects that are best specified in different logics. Moreover, heterogeneous specifications additionally have the benefit that different approaches being developed at different sites can be related, i.e. there is a formal interoperability among languages and tools. In many cases, specialized languages and tools often have their strengths in particular aspects. Using heterogeneous specification, these strengths can be combined with comparably small effort.

Current heterogeneous languages and tools do not meet these requirements. The heterogeneous language UML [3] deliberately has no formal semantics, and hence is not a formal method or logic in the sense of the present work. (However, UML could be integrated in the Heterogeneous Tool Sets as a formalism without semantics, while the different formal semantics that have been developed for UML would be represented as logic translations.) Likewise, languages for mathematical knowledge management like OpenMath and OMDoc [18] are deliberately only semi-formal. Service integration approaches like MathWeb [48] are either informal, or based on a fixed formalism. Moreover, there are many bi- or trilateral combinations of different formalisms; consider e.g. the integrated formal methods conference series [41]. Integrations of multiple decision procedures and model checkers into theorem provers, like in the PROSPER toolkit [9], provide a more systematic approach. Still, these approaches are uni-lateral in the sense that there is one logic (and one theorem prover, like the HOL prover) which serves as the central integration device, such that the user is forced to use this central logic, even if this may not be needed for a particular application (or the user may prefer to work with a different main logic).

By contrast, the heterogeneous tool set (HETS) is a both flexible, multi-lateral *and* formal (i.e. based on a mathematical semantics) integration tool. Unlike other tools, it treats logic translations (e.g. codings between logics) as first-class citizens. This can be compared with the treatment of *theory morphisms* as first-class citizens, which is a distinctive feature of formalisms like OMDoc [18] and tools like Specware [17] and IMPS [12, 11]. A clear referencing of symbols to their theories can distinguish, for example, the naturals with zero from the naturals without zero, even if they are denoted with the same symbol *Nat*. Theory morphisms can relate the two different theories of naturals. In HETS, both theory morphisms and logic comorphisms are first-class citizens. This means that HETS can also distinguish conjunction in Isabelle/HOL from conjunction in PVS[3] (these actually have two different semantics!) and relate the underlying logics with a comorphism.

[3] At least once a logic for PVS has been added.

The architecture of the heterogeneous tool set is shown in Fig. 2 on page 123. In the sequel, we will explain the details of this figure.

2 Heterogeneous Specifications: the Model-Theoretic View

We take a model-theoretic view on specifications [42]. This means that the notion of logical theory (i.e. collection of axioms) is considered to be only an auxiliary concept, and the meaning of a formal specification (of a program module) is given by

- its signature; listing the names of entities that need to be implemented, typically together with their types, that is, the *syntactic interface* of the module, and
- its class of models, that is, the set of possible *realizations* or implementations of the interface.

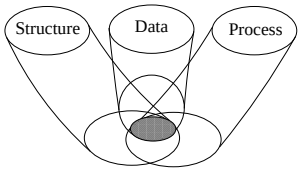


Fig. 1: Multiple viewpoints

This model-theoretic view is even more important when moving from homogeneous to heterogeneous specifications: in general, one cannot expect that different formalisms (say, a specification and a programming language, or a process algebra and a temporal logic) are related by translating theories — it is the *models* that are used to link different formalisms. This point of view is also expressed by the so-called *viewpoint specifications* (see Fig. 1), which use logical theories in different logical formalisms in order to restrict the model class of an overall system from different viewpoints (while a direct specification of the model class of the overall system would become unmanageably complex).

The correct mathematical underpinnings to this are given by the theory of *institutions* [14]. Institutions capture in a very abstract and flexible way the notion of a logical system, by leaving open the details of signatures, models, sentences (axioms) and satisfaction (of sentences in models). The only condition governing the behaviour of institutions is the *satisfaction condition*, stating that *truth is invariant under change of notation* (or enlargement of context):

$$M' \models_{\Sigma'} \sigma(\varphi) \Leftrightarrow M'|_\sigma \models_\Sigma \varphi$$

Here, $\sigma\colon \Sigma \longrightarrow \Sigma'$ is a *signature morphism*, relating different signatures (or module interfaces), $\sigma(\varphi)$ is the translation of the Σ-sentence φ along σ, and $M'|_\sigma$ is the reduction of the Σ'-model M' to a Σ-model.

The importance of the notion of institutions lies in the fact that a whole body of specification theory (concerning structuring of specifications, module concepts, parameterization, implementation, refinement, development, proof calculi) can be developed independently of the underlying institutions — all that is needed is captured by the satisfaction condition.

Different logical formalisms are related by *institution comorphisms* [13], which are again governed by the satisfaction condition, this time expressing that truth is invariant also under change of notation across different logical formalisms:

$$M' \models^{J}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \Leftrightarrow \beta_\Sigma(M') \models^{I}_{\Sigma} \varphi.$$

Here, $\Phi(\Sigma)$ is the translation of signature Σ from institution I to institution J, $\alpha_\Sigma(\varphi)$ is the translation of the Σ-sentence φ to a $\Phi(\Sigma)$-sentence, and $\beta_\Sigma(M')$ is the translation (or perhaps: reduction) of the $\Phi(\Sigma)$-model M' to a Σ-model.

Heterogeneous specification is based on some graph of logics and logic translations, formalized as institutions and comorphisms. The so-called *Grothendieck institution* [10, 24] is a technical device for giving a semantics to heterogeneous specifications. This institution is basically a flattening, or disjoint union, of the logic graph. A signature in the Grothendieck institution consists of a pair (L, Σ) where L is a logic (institution) and Σ is a signature in the logic L. Similarly, a Grothendieck signature morphism $(\rho, \sigma) : (L_1, \Sigma_1) \to (L_2, \Sigma_2)$ consists of a logic translation $\rho = (\Phi, \alpha, \beta) \colon L_1 \longrightarrow L_2$ plus an L_2-signature morphism $\sigma \colon \Phi(\Sigma_1) \longrightarrow \Sigma_2$. Sentences, models and satisfaction in the Grothendieck institution are defined in a component wise manner.

The Grothendieck institution can be understood as a flat combination of all of the involved logics. Here, "flat" means that there is no direct interaction of e.g. logical connectives from different logics that lead to new sentences; instead, just the disjoint union of sentences is taken. However, this does not mean that the logics just coexist without any interaction: they interact through the comorphisms. Comorphisms allow for translating a logical theory into some other logic, and via this translation to interact with theories in that logic (e.g. by expressing some refinement relation).

We refer the reader to the literature [14, 13, 23, 30] for full formal details of institutions and comorphisms. Subsequently, we use the terms "institution" and "logic" interchangeably, as well as the terms "institution comorphism" and "logic translation".

3 Implementation of a Logic

How is a single logic implemented in the Heterogeneous Tool Set? This is depicted in the left column of Fig. 2.

The syntactic entities of a logic are represented using types for *signatures* and signature *morphisms* forming a category with functions for identity morphisms and composition of morphisms as well as for extracting domains and codomains. There is also a type of *sentences* as well as a sentence translation function, allowing for translation of sentences along a signature morphisms.

In order to model a more verbose and user-friendly input syntax of the logic we further introduce types for the abstract syntax of *basic specifications* and *symbol maps*.

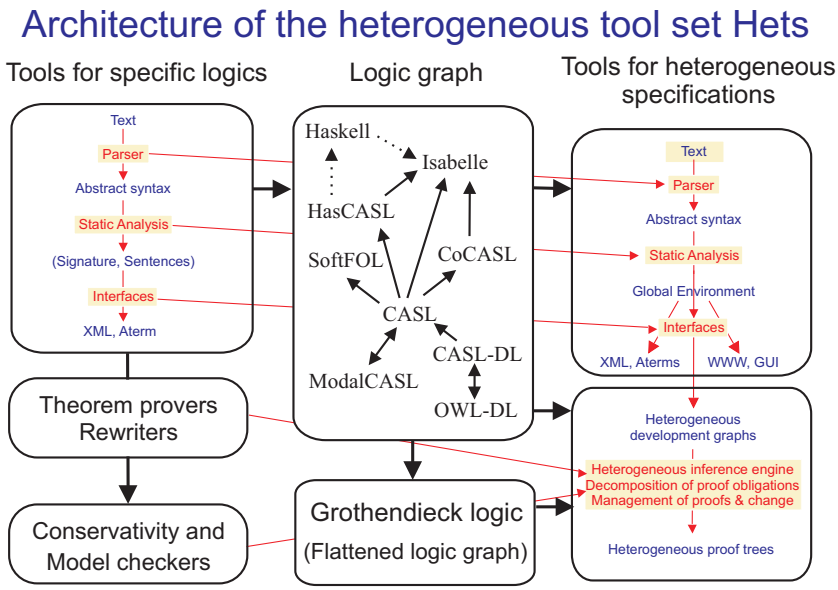


Fig. 2. Architecture of the heterogeneous tool set

```
class Logic lid sign morphism sentence basic_spec symbol_map
    | lid -> sign morphism sentence basic_spec symbol_map where
  identity :: lid -> sign -> morphism
  compose :: lid -> morphism -> morphism -> morphism
  dom, codom :: lid -> morphism -> sign
  parse_basic_spec :: lid -> String -> basic_spec
  parse_symbol_map :: lid -> String -> symbol_map
  parse_sentence   :: lid -> String -> sentence
  empty_signature :: lid -> sign
  basic_analysis :: lid -> sign -> basic_spec -> (sign, [sentence])
  stat_symbol_map :: lid -> sign -> symbol_map -> morphism
  map_sentence :: lid -> morphism -> sentence -> sentence
  provers ::
    lid -> [(sign, [sentence]) -> [sentence] -> Proof_status]
  cons_checkers :: lid -> [(sign, [sentence]) -> Proof_status]
```

Fig. 3. The basic ingredients of logics

Each logic has to provide *parsers* taking an input string and yielding an abstract syntax tree of either a basic specifications or a symbol map. *Static analysis* takes the abstract syntax of a basic specification to a *theory* being a signature with a set of sentences. Actually, an additional parameter of the analysis, a signature called “local environment”, corresponds to imported parts of a specification and will be initially the empty signature. The static analysis also takes symbol maps (written concise and user-friendly) to signature morphisms (corresponding to mathematical objects, as part of an institution).

Models are usually mathematical objects, often infinite, and hence usually not directly represented as syntactical objects. Still, usually it is possible to represent all finite models and some of the infinite models finitely. We assume that there is a syntactically recognizable subset of *constructive* specifications that are guaranteed to have a model, and use these as descriptions for models.[4] We do not require that a constructive specification has exactly one model; this covers cases where a uniqueness property would be achievable only with additional effort (such as recursive function definitions). A *model checker* evaluates whether a formula holds in a given model, or more precisely, in all models of a constructive specification.

Proof theory, more specifically, derivability of sentences from other sentences, is captured by the notion of *entailment system* [23]. In the HETS interface for logics, this is realized as follows. A theory, where some sentences are marked as axioms and others as proof goals, can be passed to a (logic-specific) *prover* which computes the entailment relation. A prover returns a proof-status answer (proved, disproved or open), together with a proof tree and further prover-specific information. The proof tree is expected to give at least the information about which axioms have been used in the proof. A *model finder* tries to construct models for a given theory, while a *conservativity checker* can check whether a theory extension is conservative (i.e. does not lead to new theorems).

Each logic is realized in the programming language Haskell [35] by a set of types and functions, see Fig. 3, where we present a simplified, stripped down version, where e.g. error handling is ignored. For technical reasons a logic is *tagged* with a unique identifier type (`lid`), which is a singleton type the only purpose of which is to determine all other type components of the given logic. In Haskell jargon, the interface is called a multiparameter type class with functional dependencies [36]. The Haskell interface for logic translations is realised similarly.

4 Logics Available in Hets

In this section we give a short overview of the logics available in HETS.

Propositional is classical propositional logic, with the zChaff SAT solver [15] connected to it.

CASL extends many sorted first-order logic with partial functions and subsorting. It also provides induction sentences, expressing the (free) generation of datatypes. For more details on CASL see [8, 6]. We have implemented the CASL logic in such a way that much of the implementation can be re-used for CASL extensions as well; this is achieved via "holes" (realized via polymorphic variables) in the types for signatures, morphisms, abstract syntax etc. This eases integration of CASL extensions and keeps the effort of integrating CASL extensions quite moderate.

[4] If necessary, one can always extend the logic with new sentences leading to constructive specifications.

CoCASL [33] is a coalgebraic extension of CASL, suited for the specification of process types and reactive systems. The central proof method is coinduction.

ModalCASL is an extension of CASL with multi-modalities and term modalities. It allows the specification of modal systems with Kripke's possible worlds semantics. It is also possible to express certain forms of dynamic logic.

HasCASL [44] is a higher order extension of CASL allowing polymorphic datatypes and functions. It is closely related to the programming language Haskell and allows program constructs to be embedded in the specification.

Haskell [35] is a modern, pure and strongly typed functional programming language. It simultaneously is the implementation language of HETS, such that in the future, HETS might be applied to itself.

OWL DL is the Web Ontology Language (OWL) recommended by the World Wide Web Consortium (W3C, `http://www.w3c.org`). It is used for knowledge representation and the Semantic Web [5].

CASL-DL [20] is an extension of a restriction of CASL, realizing a strongly typed variant of OWL DL in CASL syntax.

SoftFOL [21] offers three automated theorem proving (ATP) systems for first-order logic with equality: (1) SPASS [45]; (2) Vampire [39]; and (3) MathServ Broker[5] [47]. These together comprise some of the most advanced theorem provers for first-order logic.

Isabelle [34] is an interactive theorem prover for higher-order logic, and (jointly with others) marks the frontier of current research in interactive higher-order provers.

Propositional, SoftFOL and Isabelle are the only logics coming with a prover. Proof support for the other logics can be obtained by using logic translations to a prover-supported logic.

5 Heterogeneous Specification

Heterogeneous specification is based on some graph of logics and logic translations. The graph of currently supported logics is shown in Fig. 2. However, syntax and semantics of heterogeneous specifications as well as their implementation in HETS is parameterized over an arbitrary such logic graph. Indeed, the HETS modules implementing the logic graph can be compiled independently of the HETS modules implementing heterogeneous specification, and this separation of concerns is essential to keep the tool manageable from a software engineering point of view.

Heterogeneous CASL (HETCASL; see [26]) includes the structuring constructs of CASL, such as union and translation. A key feature of CASL is that syntax and semantics of these constructs are formulated over an arbitrary institution (i.e. also for institutions that are possibly completely different from first-order logic resp. the CASL institution). HETCASL extends this with constructs for the translation of specifications along logic translations.

[5] which chooses an appropriate ATP upon a classification of the FOL problem

```
SPEC ::= BASIC-SPEC
       | SPEC then SPEC
       | SPEC then %implies SPEC
       | SPEC with SYMBOL-MAP
       | SPEC with logic ID

DEFINITION ::= logic ID
             | spec ID = SPEC end
             | view ID : SPEC to SPEC = SYMBOL-MAP end
             | view ID : SPEC to SPEC = with logic ID end

LIBRARY = DEFINITION*
```

Fig. 4. Syntax of a simple subset of the heterogeneous specification language. `BASIC-SPEC` and `SYMBOL-MAP` have a logic specific syntax, while `ID` stands for some form of identifiers.

The syntax of heterogeneous specifications is given (in very simplified form) in Fig. 4. A specification either consists of some basic specification in some logic (which follows the specific syntax of this logic), or an extension of a specification by another one (written `SPEC then SPEC`, or, if the extension only adds theorems that are already implied by the original specification, written `SPEC then %implies SPEC`). A translation of a specification along a signature morphism is written `SPEC with SYMBOL-MAP`, where the symbol map is logic-specific (usually abbreviatory) syntax for a signature morphism. A translation along a logic comorphism is written `SPEC with logic ID`.

A specification library consists of a sequence of definitions. A definition may select the current logic (`logic ID`), which is then used for parsing and analysing the subsequent definitions. It may name a specification, and finally it may also declare a *view* between two specifications. A view is a kind of refinement relation between two specifications, expressing that the first specification (when translated along a signature morphism or a logic comorphism) is implied by the second specification. Indeed, using the heterogeneous language constructs (including the possibility to add new logic translations involving e.g. behavioural quotient constructions) it is possible to capture a large variety of different refinement notions just by heterogeneous views as explained above.

It should be stressed that the name "HETCASL" only refers to CASL's structuring constructs. The individual logics used in connection with HETCASL and HETS can be completely orthogonal to CASL. Actually, the capabilities of HETS go even beyond HETCASL, since HETS also supports other module systems. This enables HETS to directly read in e.g. OWL files, which use a structuring mechanism that is completely different from CASL's. Moreover, support of further structuring languages is planned.

The Grothendieck logic (see Sect. 2), which is the semantic basis of HETCASL, can be implemented as a bunch of *existential* datatypes over the type class `Logic`. Usually, existential datatypes are used to realize — in a strongly typed language —

heterogeneous lists, where each element may have a different type. We use lists of (components of) logics and translations instead. This leads to an implementation of the Grothendieck institution over a logic graph.

6 Parsing and Analysis of Heterogeneous Specifications

Based on the type class `Logic`, a number of logics and various comorphisms among these have been implemented for HETS. We now come to the logic-independent modules in HETS, which can be found in the right half of Fig. 2. These modules comprise roughly one third of HETS' 100.000 lines of Haskell code.

The heterogeneous parser transforms a string conforming to the syntax in Fig. 4 to an abstract syntax tree, using the `Parsec` combinator parser [19]. Logic and translation names are looked up in the logic graph — this is necessary to be able to choose the correct parser for basic specifications. Indeed, the parser has a state that carries the current logic, and which is updated if an explicit specification of the logic is given, or if a logic translation is encountered (in the latter case, the state is set to the target logic of the translation). With this, it is possible to parse basic specifications by just using the logic-specific parser of the current logic as obtained from the state.

The static analysis is based on the static analysis of basic specifications, and transforms an abstract syntax tree to a development graph (cf. Sect. 7 below). Starting with a node corresponding to the empty theory, it successively extends (using the static analysis of basic specifications) and/or translates (along the intra- and inter-logic translations) the theory, while simultaneously adding nodes and links to the development graph.

7 Proof Management with Development Graphs

The central device for structured theorem proving and proof management in HETS is the formalism of *development graphs*. Development graphs have been used for large industrial-scale applications with hundreds of specifications [16]. They also support management of change. The graph structure provides a direct visualization of the structure of specifications, and it also allows for managing large specifications with hundreds of sub-specifications.

A development graph (see Fig. 7 for an example) consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called *definition links*, indicating the dependency of each involved structured specification on its subparts. Each node is associated with a signature and some set of local axioms. The axioms of other nodes are inherited via definition links. Definition links are usually drawn as black solid arrows, denoting an import of another specification.

Complementary to definition links, which *define* the theories of related nodes, *theorem links* serve for *postulating* relations between different theories. Theorem links

are the central data structure to represent proof obligations arising in formal developments. Theorem links can be *global* (drawn as solid arrows) or *local* (drawn as dashed arrows): a global theorem link postulates that all axioms of the source node (including the inherited ones) hold in the target node, while a local theorem link only postulates that the local axioms of the source node hold in the target node.

Both definition and theorem links can be *homogeneous*, i.e. stay within the same logic, or *heterogeneous*, i.e. the logic changes along the arrow. Technically, this is the case for Grothendieck signature morphisms (ρ, σ) where $\rho \neq id$. This case is indicated with double arrows.

Theorem links are initially displayed in red in the tool. (In Fig. 7, they are displayed using thin lines and non-filled arrow heads.) The *proof calculus* for development graphs [28, 31, 27] is given by rules that allow for proving global theorem links by decomposing them into simpler (local and global) ones. Theorem links that have been proved with this calculus are drawn in green. Local theorem links can be proved by turning them into *local proof goals*. The latter can be discharged using a logic-specific calculus as given by an entailment system (see Sect. 3). Open local proof goals are indicated by marking the corresponding node in the development graph as red; if all local implications are proved, the node is turned into green. This implementation ultimately is based on a theorem [27] stating soundness and relative completeness of the proof calculus for heterogeneous development graphs.

While the semantics of theorem links is explained in entirely model-theoretic terms, theorem links can ultimately be reduced to local proof obligations (and conservativity checks) of a proof-theoretic nature, amenable to machine implementation. Note however, that this approach is quite different from that of logical frameworks. Suppose that we have a global theorem link $\sigma\colon N_1 \longrightarrow N_2$ between two nodes N_1 and N_2 in a development graph. Note that the logics of N_1 and N_2 may be different. The logical framework approach assumes that the theories of N_1 and N_2 are encoded into some logic that is fixed once and forall. By contrast, in HETS we can rather flexibly find a logic that is a "common upper bound" of the logics of both N_1 and N_2 and that moreover has best possible tool support. This freedom allows us to exploit specialized tools. This is also complemented by a sublogic analysis, which is required for each of the logics in HETS, and which allows for an even more fine-grained determination of available tools.

8 An Example

In the domain of qualitative constraint reasoning, a subfield of AI which has evolved in the past 25 years, a large number of calculi for efficient reasoning about spatial and temporal entities have been developed. A prominent example of that kind are the various region connection calculi [38]. In the region connection calculus RCC8, which also has become a GIS standard, it is possible to express relations between regions (= regular closed sets) in a metric space. The set of RCC8 base relations consists

of the relations DC ("DisConnected"), EC ("Externally Connected"), PO ("Partially Overlap"), TPP ("Tangential Proper Part"), NTPP ("Non-Tangential Proper Part"), the converses of the latter two relations (TPPi and NTPPi, resp.) and EQ ("EQals") (see Fig. 5 for a pictorial representation). The RCC5 calculus is similar, but does not distinguish between tangential and non-tangential parts; it hence has only 5 basic relations.

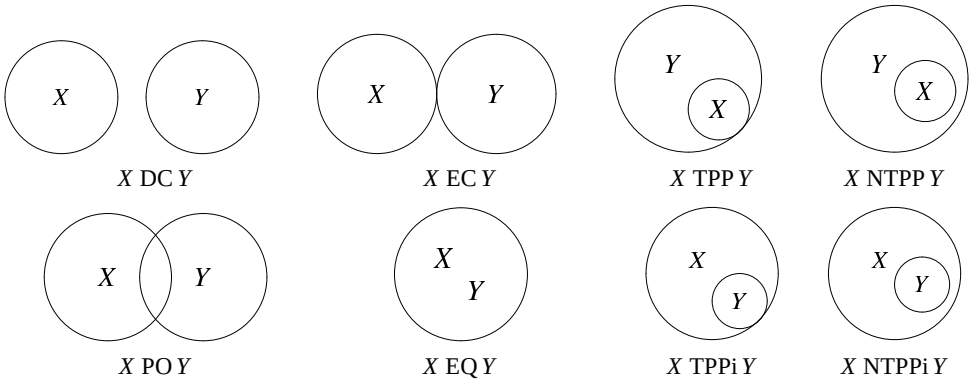


Fig. 5. The RCC-8 relations

For efficiency and feasibility reasons, qualitative spatial and temporal reasoning is not directly done in a (typically infinite) metric space, but rather at the abstract level of a (finite) relation algebra, for example, using the so-called path consistency algorithm. The heart of this approach is the composition table, which captures composition of relations at the abstract and finitary level of the relation algebra.

Composition tables need to be set up only once and for all. Still, this process is error-prone, and we already have found errors in published composition tables. Hence, formal verification of composition tables (w.r.t. their semantic interpretation) is an important task. In [46], we present a heterogeneous verification of the RCC8 composition table w.r.t. the interpretation in metric spaces. This verification goal can be split into two subgoals:

1. Verification that closed discs in a metric (cf. node `RCC_FO` in Fig. 7) satisfy some of Bennett's connectedness axioms [4] (cf. node `MetricSpace` in Fig. 7). `RCC_FO` consists of very *few* (actually, 4) theorems, so-called *bridge lemmas*. Since `MetricSpace` is a higher-order theory, they need to be translated to higher-order logic, and can then be proved using the *interactive* theorem prover Isabelle.
2. Verification that Bennett's connectedness axioms imply the standard RCC axioms (cf. nodes `ExtRCCByRCC5Rels` and `ExtRCCByRCC8Rels` in Fig. 7). The latter are *many* (actually, 95) first-order theorems, and can be proved using the *automated* theorem proving system SPASS.

```
view RCC_FO_IN_METRICSPACE :
    RCC_FO to
    {EXTMETRICSPACEBYCLOSEDBALLS[METRICSPACE]
     then %def
          pred __C__ : ClosedBall × ClosedBall;
               nonempty(x : ClosedBall) ⇔ x C x
          ∀ x, y : ClosedBall
          • x C y ⇔ ∃ s : S • rep x s ∧ rep y s                    %(C_def)%
    } =
    QReg ↦ ClosedBall
end
```

Fig. 6. Specification of a heterogeneous refinement expressing correctness of the RCC8 composition table

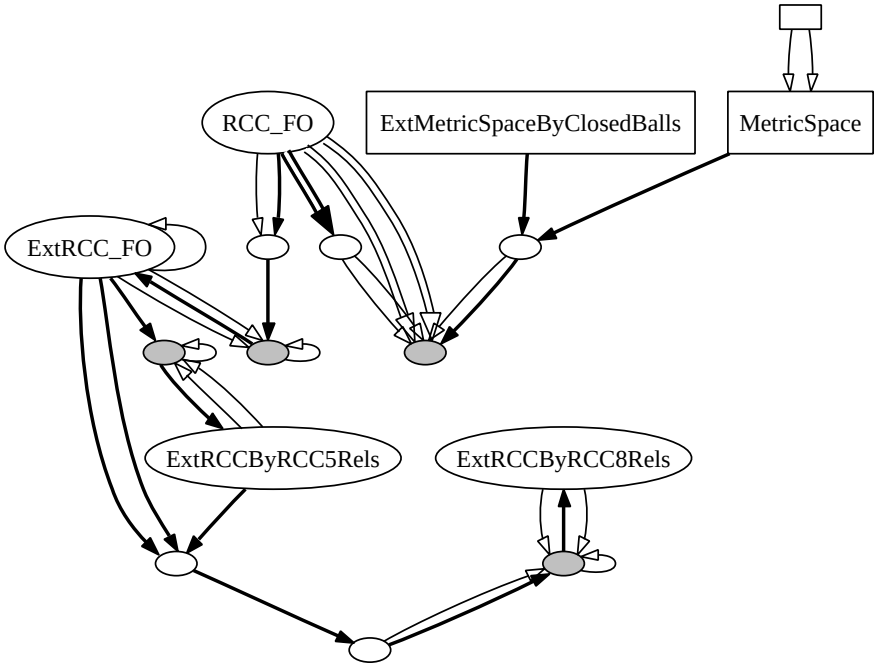


Fig. 7. Development graph for correctness proof of RCC8 composition table in CASL and HASCASL

9 Theorem Proving with HETS

Fig. 6 contains the heterogeneous refinement expressing the correctness of the RCC8 composition table. After parsing and static analysis of an heterogeneous specification (see Sect. 6), HETS constructs a heterogeneous development graph, see Fig. 7. This graph can be inspected, e.g. theories of nodes or signature morphisms of links can be displayed. Using the calculus mentioned in Sect. 7, the proof obligations in the graph can be (in most cases automatically) reduced to local proof goals at the individual nodes. Nodes with local proof goals are marked with a grey color in Fig. 7, while in the tool, red is used. The thick edges in the development graph are definition links and the thin ones are theorem links. A double arrow denotes a heterogeneous link (e.g.

between RCC_FO and the extension of EXTMETRICSPACEBYCLOSEDBALLS). Unnamed nodes show intermediate structuring of specifications and box-shaped nodes are imported from a different specification library, while the round nodes are theories specified locally.

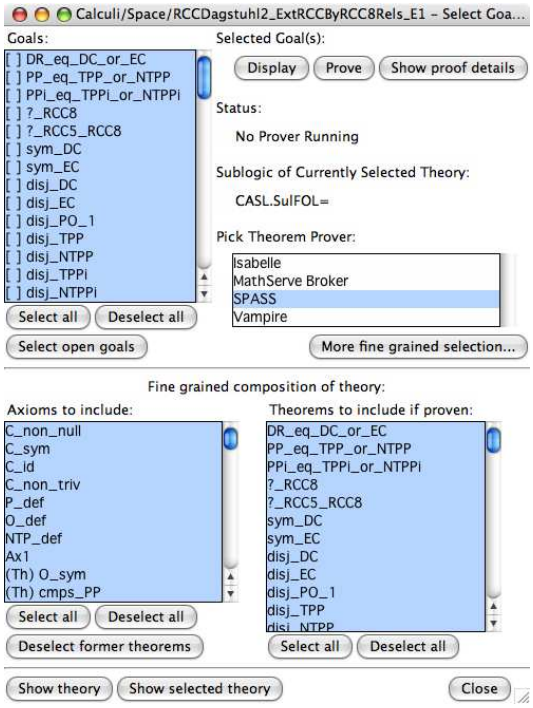


Fig. 8. Hets Goal and Prover Interface

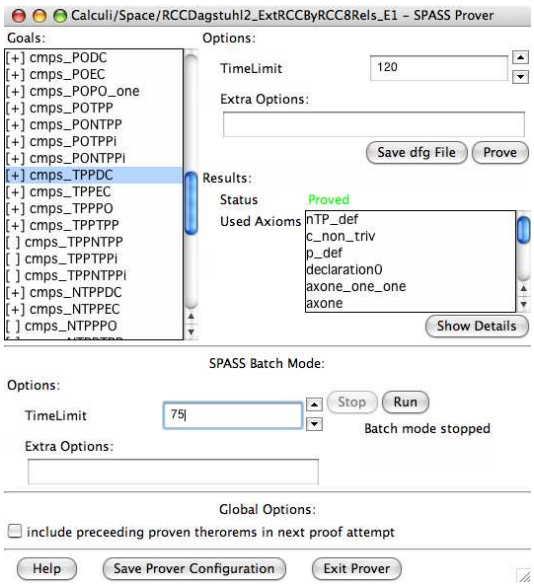


Fig. 9. Interface of the SPASS prover

The graphical user interface (GUI) for calling a prover is shown in Fig. 8. The list on the left shows all goal names prefixed with the proof status in square brackets. A proved goal is indicated by a '+', a '-' indicates a disproved goal and a space denotes an open goal. Within this list, one can select those goals that should be inspected or proved. A button 'Display' shows the selected goals in the syntax of this theory's logic.

The list 'Pick Theorem Prover:' lets you choose one of the connected provers. By pressing 'Prove' the selected prover is launched and the theory along with the selected goals is translated via the shortest possible path of comorphisms into the prover's logic. However, the shortest path need not always be the best one. Therefore, the button 'More fine grained selection...' lets you pick a specific path of comorphisms in the logic graph that leads into a prover supported logic. It is assumed that all comorphisms are model-expansive, which means that borrowing of entailment systems along the composite comorphism $\rho = (\Phi, \alpha, \beta)$ is sound and complete [7, 27]:

$$(\Sigma, \Gamma) \models^I_\Sigma \varphi \text{ iff } (\Phi(\Sigma), \alpha(\Gamma)) \models^J \alpha_\Sigma(\varphi).$$

That is, if the entailment $\vdash$ generated by the prover captures semantic consequence $\models$, we can re-use the prover along the (composite) comorphism. In the terminology of [1], $(\Sigma, \Gamma) \models^I_\Sigma \varphi$ in institution I captures the *what to prove*, while its translation to institution J captures the *how to prove*.

Additionally, this interface offers to select in detail the axioms and proven theorems which are included in the theory for the next proof attempt. Among the axioms theorems imported from other specifications are marked with the prefix '(Th)'. This is particularly useful for large theories with problematic theorems that blow up the search space of ATP systems. A detailed discussion of using ATPs for CASL can be found in [21].

If an ATP is selected, a new window is opened, which controls the prover calls (Fig. 9). Here we use the connection to SPASS [45], for the other ATPs listed (MathServ Broker and Vampire) see [21]. Isabelle [34], a semi automatic theorem prover, is started with ProofGeneral [2] in a separate Emacs from the GUI.

The 'Close' button allows for integrating the status of the goals' list back into the development graph. If all goals have been proved, this theory's node turns from red into green.

For the example presented in Sect. 8 we successfully used SPASS for proving the CASL proof obligations in the unnamed grey nodes between the nodes "RCC_FO" and "ExtRCC_FO" and below "ExtRCC_FO". To discharge the proof obligations in the node below "RCC_FO" with incoming heterogeneous theorem links on the right of the center of Fig. 7 the higher-order proof assistance system Isabelle was applied. The most interesting point here is that we used a first-order specification, namely RCC_FO, to prove as much as possible by the ATP SPASS (thus minimizing the number of proof obligations to be proven by a semi-automatic reasoner).

10 Conclusion

The Heterogeneous Tool Set is available at `http://www.dfki.de/sks/hets`; some specification libraries and example specifications (including those of this paper) under `http://www.cofi.info/Libraries`. A user guide is also available there. Brief introductions into HETS are given in [32] and [6].

There is related work about generic parsers, user interfaces, theorem provers etc. [34, 2]. However, these approaches are mostly limited to *genericity*, and do not support real *heterogeneity*, that is the simultaneous use of different formalisms. Technically, genericity often is implemented with generic modules that can be instantiated many times. Here, we deal with a potentially unlimited number of such instantiations, and also with translations between them.

```
logic CSP-CASL
spec BUFFER =
    data LIST
    channels read, write : Elem
    process let Buf(l : List[Elem]) =
                    read?x → Buf(cons(x, nil))
                    □ if l = nil then STOP
                      else write!last(l) → Buf(rest(l))
            in Buf(nil)
    with logic → MODALCASL
    then %implies • AGF ∃x : Elem . ⟨write.x⟩ true
end
```

Fig. 10. A specification of fair buffers in CASL, CSP-CASL and MODALCASL.

It may appear that HETS just provides a combination of some first-order provers and Isabelle, and the reader may wonder what the advantage of HETS is when compared to an ad-hoc combination of Isabelle and such provers, like [22]. But already now, HETS provides proof support for modal logic (via the translation to CASL, and then further to either SPASS or Isabelle), as well as for COCASL. Hence, it is quite easy to provide proof support for new logics by just implementing logic translations, which is at least an order of magnitude simpler than integrating a theorem prover. Although this can be compared to embedding the new logic in a HOL prover, our translational approach has the major advantage that several translations may exist in parallel (think of the standard and functional translations of modal logic), and the best one may be chosen depending on the theory at hand.

Future work will integrate more logics and interface more existing theorem proving tools with specific institutions in HETS. In [25], we have presented a heterogeneous specification with more diverse formalisms, namely CSP-CASL [40] (a combination of CASL with the process algebra CSP), and a temporal logic (as part of MODALCASL). An example is shown in Fig. 10. CSP-CASL is used to describe the system (a buffer implemented as a list), and some temporal logic is used to state fairness or eventuality properties that go beyond the expressiveness of the process algebra (here, we express the fairness property that the buffer cannot read infinitely often without writing).

In [29] we describe how heterogeneous specification and HETS could be used for proving a refinement of a specification in CASL into a Haskell-program. Another challenge is the integration of proof planners into HETS. Finally, there is work in progress about the meta-level specification of institutions and their comorphisms in Twelf [37], which shall lead to correctness proofs for the comorphisms integrated into HETS.

Acknowledgement

This work has been supported by the project MULTIPLE of the *Deutsche Forschungsgemeinschaft* under grant KR 1191/5-2. We thank Katja Abu-Dib, Mihai Codescu,

Carsten Fischer, Jorina Freya Gerken, Rainer Grabbe, Sonja Gröning, Daniel Hausmann, Wiebke Herding, Hendrik Iben, Cui "Ken" Jian, Heng Jiang, Anton Kirilov, Tina Krausser, Martin Kühl, Mingyi Liu, Dominik Lücke, Maciek Makowski, Immanuel Normann, Razvan Pascanu, Daniel Pratsch, Felix Reckers, Markus Roggenbach, Pascal Schmidt, Lutz Schröder, Paolo Torrini, René Wagner, Jian Chun Wang and Thiemo Wiedemeyer for help with the implementation of HETS, and Erwin R. Catesbeiana for testing the consistency checker.

References

1. Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
2. David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, LNCS 1785, pages 38–42. Springer, 2000.
3. Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors. *UML 2004*, LNCS 3273. Springer, 2004.
4. B. Bennett. *Logical Representations for Automated Reasoning about Spatial Relationships*. PhD thesis, School of Computer Studies, The University of Leeds, 1997.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
6. M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 of *LNCS*. Springer, 2004.
7. M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997.
8. CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.
9. Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, and Thomas F. Melham. The prosper toolkit. *STTT*, 4(2):189–210, 2003.
10. R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.
11. William M. Farmer. An infrastructure for intertheory reasoning. In *Automated Deduction - CADE-17*, LNCS 1831, pages 115–131. Springer, 2000.
12. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, 1993.
13. J. Goguen and G. Roşu. Institution morphisms. *Formal aspects of computing*, 13:274–307, 2002.
14. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.
15. Marc Herbstritt. zChaff: Modifications and extensions. report00188, Institut für Informatik, Universität Freiburg, July 17 2003. Thu, 17 Jul 2003 17:11:37 GET.
16. Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Wolpers Wolpers. Verification support environment (VSE). *High Integrity Systems*, 1(6):523–530, 1996.
17. Kestrel Development Corporation. Specware 4.1 language manual. `http://www.specware.org/`.
18. Michael Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*. LNCS 4180. Springer, 2006.
19. Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report. UU-CS-2001-35.
20. K. Lüttich, T. Mossakowski, and B. Krieg-Brückner. Ontologies for the Semantic Web in CASL. In José Fiadeiro, editor, *WADT 2004*, LNCS 3423, pages 106–125. Springer, 2005.
21. Klaus Lüttich and Till Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. WADT 2006, Springer LNCS, to appear.
22. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput*, 204(10):1575–1596, 2006.

23. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
24. T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *MFCS*, LNCS 2420, pages 593–604. Springer, 2002.
25. T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT 2002*, LNCS Vol. 2755, pages 359–375. Springer, 2003.
26. T. Mossakowski. HetCASL - heterogeneous specification. language summary, 2004.
27. T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.
28. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.
29. Till Mossakowski. Institutional 2-cells and Grothendieck institutions. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation. Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, LNCS 4060, pages 124–149. Springer, 2006.
30. Till Mossakowski, Joseph Goguen, Razvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser, 2005.
31. Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. CASL logic. In Peter D. Mosses, editor, *CASL Reference Manual*, LNCS 2960, part IV. Springer Verlag, 2004.
32. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.
33. Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*, 67(1-2):146–197, 2006.
34. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.
35. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: J. Funct. Programming **13** (2003).
36. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*. 1997.
37. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. pages 202–206.
38. D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In B. Nebel, W. Swartout, and C. Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, pages 165–176. Morgan Kaufmann, 1992.
39. Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.
40. Markus Roggenbach. Csp-casl - a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, 2006.
41. Judi Romijn, Graeme Smith, and Jaco van de Pol, editors. *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*, volume 3771 of *Lecture Notes in Computer Science*. Springer, 2005.
42. Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.
43. Klaus Schneider. *Verification of Reactive Systems*. Springer Verlag, 2004.
44. L. Schröder and T. Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Reingeissen, editors, *AMAST, 2002*, LNCS 2422, pages 99–116. Springer, 2002.
45. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, LNCS 2392, pages 275–279. Springer-Verlag, 2002.
46. Stefan Wölfl, Till Mossakowski, and Lutz Schröder. Qualitative constraint calculi: Heterogeneous verification of composition tables. In *20th International FLAIRS Conference*, 2007.
47. Jürgen Zimmer and Serge Autexier. The MathServe System for Semantic Web Reasoning Services. In U. Furbach and N. Shankar, editors, *3rd IJCAR*, LNCS 4130. Springer, 2006.
48. Jürgen Zimmer and Michael Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In Andrei Voronkov, editor, *18th CADE*, LNCS 2392, pages 139–143. Springer, 2002.

Fully Verified Java Card API Reference Implementation

Wojciech Mostowski

Computing Science Department, Radboud University Nijmegen, the Netherlands
woj@cs.ru.nl

Abstract. We present a formally verified reference implementation of the Java Card API. This case study has been developed with the KeY verification system. The KeY system allows us to symbolically execute the Java source code of the API in the KeY verification environment and, in turn, prove correctness of the implementation w.r.t. formal specification we developed along the way. The resulting formal API framework (the implementation and the specification) can be used to prove correctness of any Java Card applet code. As a side effect, this case study also serves as a benchmark for the KeY system. It shows that a code base of such size can be feasibly verified, and that KeY indeed covers all of the Java Card language specification.

1 Introduction

We present a formally verified reference implementation of the Java Card API version 2.2.1 [11]. Java Card is a technology used to program smart cards and it comprises a subset of the desktop Java; a subset of the programming language itself, and a cut down version of the API. Reference implementations of the API that are normally available for Java Card are developed for a particular running environment, usually a Java Card simulator or emulator. Some of the API routines are not implementable in Java alone. For example, Java Card transaction mechanism functionality is normally provided by the Java Card VM and/or the operating system of the actual smart card hardware. A simulator provides similar functionality through Java Native Interface (JNI). Thus, the API implementation consists of the Java part and the JNI part, the latter reflecting the low-level behaviour of the card. Both parts together enable the API implementation to run on a simulator. In contrast, our API implementation has been developed for the KeY interactive verification system[1] [1]. That is, the implementation is designed to be symbolically executed in a formal verification environment. Similarly to a simulator, the KeY verifier also needs to handle low-level Java Card specific routines. In this case the JNI functionality is provided by the formal model of the Java Card environment embedded in the KeY verifier logic; a set of specialised logic rules to symbolically execute Java Card native method calls. Figure 1 shows corresponding architectures of the API implementations.

Along with the implementation we developed a set of formal specifications for the API. And, naturally, we used the KeY system to prove the correctness

[1] http://www.key-project.org

of our implementation w.r.t. to the specification. The proving is done by means of symbolic execution of the Java source code of the API implementation in the KeY system and then evaluating the specification formulae on the resulting execution state.

This case study serves three main goals: (i) an API framework (implementation and specification) for verification of Java Card applet source code, (ii) consistency of the informal API specification [5], and (iii) as a benchmark for the KeY system and as a verification case study itself that explores the usability of formal methods in practice. We elaborate on these goals in the following paragraphs.

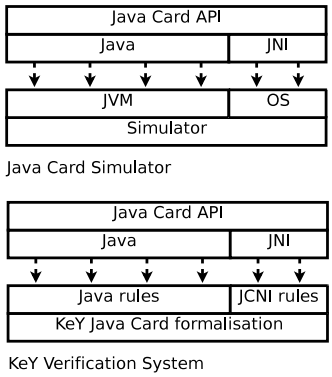


Fig. 1. Architecture of Java Card API implementations

In the current PinPas Java Card project[2] we investigate fault injection attacks on smart cards. A possibility for a fault injection calls for appropriate countermeasures. One of such countermeasures are simply modifications to Java applet source code to detect and neutralise faults. Such modifications can result in a complex code. One of our goals in the project is to be able to formally verify such modified source code, i.e., that the more complex fault-proof code behaves the same way as the simple fault-sensitive code, as described in earlier work [4]. For this we need a verification tool that faithfully reflects Java Card programming language semantics, and also a faithful reflection of the API behaviour. The verification tool of our choice, the KeY system, already provides the formalisation of the whole of Java Card programming language semantics. We said earlier that Java Card is a subset of Java. In practice, because of the specifics of the smart card technology, Java Card introduces additional complications to the language. Namely, two different kinds of writable memory (persistent EEPROM and transient RAM), an atomic transaction mechanism, and applet firewall mechanism. All these features are embedded into the Java Card Virtual Machine (JCVM) running on a card. In effect, this sometimes changes the semantics of the primitive Java statements in the context of a smart card application. The KeY system already supports all of the mentioned Java Card specific features [1, Chapter 9], with a notable exception of the firewall mechanism, which is being integrated into KeY in parallel to this work.

When it comes to the API behaviour, however, our approach so far was to specify and implement only those API methods that are required for a given verification task at hand [1, Chapter 14]. Ultimately, to be able to verify *any* given Java Card applet code w.r.t. wide range of properties, the specification *and* the implementation for the whole of the API should be present. The need

[2] `http://www.win.tue.nl/pinpasjc/`

for having *both* the specification and implementation is justified as follows. First of all, reasoning on the level of interfaces, i.e., relying on method contracts only, is not sufficient for some properties. In particular, strong invariant properties require the evaluation of all intermediate atomic states of execution, including the ones that result from the execution of an API method. Moreover, in principle, method contracts cannot be applied in the context of Java Card transactions. Here, sometimes a special "transaction aware" contract is needed, which in some cases can only be constructed by hand based on the actual implementation of the method. In essence, one needs to state the behaviour of the method in terms of conditionally updated data, rather than the actual data. For some API methods that interact heavily with the transaction mechanism giving a suitable contract is not possible at all [1, Chapter 9]. Finally, in some few cases, formal reasoning based on the code instead of the specification can be simply more efficient.

We tried to make the reference implementation as faithful as possible, i.e., closely reflecting the actual card behaviour. The official informal specifications were closely followed, and sometimes also the actual cards were tested to clarify the semantics of some API methods. However, some Java Card aspects had to be omitted in the implementation and it is obvious that certain gaps will always remain between our reference implementation executing in a formal verification environment and API implementations executing on actual smart cards. We discuss those issues in detail in Section 3.

One of the other goals of the PinPasJC project is to discover inconsistencies between the actual cards and the official informal Java Card specifications, i.e., to simply find implementation bugs on the cards. Implementing the whole of the Java Card API gave us a good chance to review the informal specification and identify "hot spots" – descriptions that are likely to be ignored or misinterpreted, unintentionally or on purpose. We mention this briefly in Section 4.

Finally, our reference implementation serves as a verification case study. First of all, it gives a general idea of how big a code base can be subjected to formal verification by the KeY tool. The Java Card API consists of 60 classes adding up to 205KB of Java source code (circa 5000 LoC) and 395KB of formal specifications (circa 10000 LoC). The KeY system managed to deal with it giving satisfying verification results, although not without problems, see Section 4. Moreover, the case study shows the compliance of the KeY system to the Java Card language specification – the reference implementation utilises practically all of the Java Card language subset and was successfully verified by the KeY system. Last, but not least, this case study will serve to further optimise and tune the KeY system in terms of performance (resource-wise) and minimising user interaction.

Related Work. The work we present here is one of many that formally treats the Java Card API library. However, to the best of our knowledge, we are the first

ones to give a relatively full, (soon) publicly available reference implementation of version 2.2.1 of the API together with specifications. The older Java Card development kits distributed by Sun contained a reference implementation of the API up to version 2.1.1.[3] Since that version the source code of the API is no longer available with the development kits. Our code borrows ideas from Sun's reference implementation, there are however two major differences to be noted. First, our implementation treats the newer API version which introduces many new features. Secondly, our back-end system is the KeY Java Card model, while Sun's API is implemented for the JCWDE simulator.

When it comes to formal treatment of the API, a full set of JML specifications[4] [9] for the API version 2.1.1 has been developed for the LOOP verification tool [6] and ESC/Java2 [2]. These efforts, however, do not include the implementation itself, only specifications. As a side effect of our work, we also constructed lightweight JML specifications of the API version 2.2.1 for ESC/Java2.[5] Moreover, in an earlier work we investigated specification of the Java Card API in OCL [7].

Recently a work has been published on a method to formally verify native API implementations based on specification refinement [10]. Three levels of specifications for the native API code are given in the Coq language: functional specification, high-level description, and low-level description. The last level is not yet the actual implementation of the code on the card (normally written in C or even assembly), but is claimed to have a one to one correspondence with the code running on the card. The correctness of the low-level description is verified by means of refinement relation between the three levels of specification. The main goal is to verify the actual API implementations found on cards, while we aim at providing a Java source code verification framework to be used *outside* of the card.

Structure of the Paper. The rest of this paper is organised as follows. Section 2 describes tools and methodologies we used: the KeY system, its logic, the Java Card formal model on top of which our reference implementation was written, and a discussion about the choice of the specification language. Section 3 describes the implementation and its specification in more detail with samples of both, while Section 4 gives some insights into the verification effort and discusses our experience. Finally, Section 5 concludes the paper.

2 Tools and Methodology

In this section we describe the main building blocks of our case study: the KeY system, the KeY model of the Java Card environment, and briefly the Java Card Dynamic Logic, which we used a specification language.

[3] `http://java.sun.com/products/javacard/dev_kit.html`
[4] `http://www.sos.cs.ru.nl/research/escjava/esc2jcapi.html`
[5] Available at `http://www.cs.ru.nl/~woj/software/software.html`

2.1 Verification Tool

The verification tool of our choice for this case study is the KeY system. The KeY system is a highly automated interactive program verifier for Java and Java Card programs. Currently KeY supports verification of sequential Java without floating point arithmetic and dynamic class loading, and practically all of Java Card[6] including the Java Card transaction mechanism and (non)persistence of the Java Card memory [1, Chapter 9]. The formalism behind the KeY system is the Java Card Dynamic Logic (Java Card DL) [1, Chapter 3] built on top of first order logic. The rules of the logic are used to symbolically execute Java programs in the KeY prover and then evaluate the properties to be verified. We describe this idea with a simple example. Take the following Java code:

```
public void decreaseCounter() { if(counter > 0) counter--; }
```

What we would like to specify and prove is that assuming the counter is non-negative it will stay non-negative after the execution of the method, i.e., that the method preserves the invariant `counter >= 0`. A corresponding Java Card DL formula would take the following form (we use the actual KeY system syntax):

```
self != null & self.counter >= 0 ->
  \<{ self.decreaseCounter(); }\> self.counter >= 0
```

The formula itself does not contain any class or method definitions, these are implicitly present in the prover. The left side of the implication (`->`) represents the state in which we are about to execute the program, i.e., the precondition of the program. The program itself, calling of a method `decreaseCounter` on the object `self`, is included in the diamond modality `\<·\>`. The formula attached to the modality is to be evaluated after the program in the modality executes, thus, the formula represents the postcondition of the program. In Java Card DL the diamond modality requires the program to terminate and do so in a non-abrupt fashion. In particular, the program is not allowed to throw any exceptions. Contrary to the diamond modality, the box modality `\[·\]` does not require (non-abrupt) termination. In our work, however, the stronger (for deterministic programs) diamond semantics is always used.

The first few simplification steps transform the formula above into a Java Card DL sequent:

```
self != null, self.counter >= 0 ==>
\<{ self.decreaseCounter(); }\> self.counter >= 0
```

The left side of the sequent (marked by `==>`), the antecedent, represents the assumptions and the right side, the succedent represents the proof goal. The next few rules of the symbolic execution unfold the sequent into two branches:

[6] With a notable exception of the Java Card firewall mechanism. However, this does not limit the set of Java Card programs that can be verified with KeY, only the set of properties, see Section 3 for details.

```
self != null, self = null, self.counter >= 0 ==>
\<{ throw new NullPointerException(); }\> self.counter >= 0

self != null, self.counter >= 0 ==>
\<{ if(self.counter > 0) self.counter--; }\> self.counter >= 0
```

The first branch represents the null value check for accessing the object `self`. Whenever an object is accessed (field access or method call), the Java Card DL calculus establishes non-nullness of the referenced object which, if not satisfied, would cause a null pointer exception. This branch is easily closed (proved) by a contradiction in the antecedent `self = null` and `self != null`. We skip further null pointer checks in the rest of the example.

The Java Card DL rule for the `if` statement splits the second sequent into two branches that correspond to the execution paths of the `if` statement:

```
self != null, self.counter >= 0, self.counter > 0 ==>
\<{ self.counter--; }\> self.counter >= 0

self != null, self.counter >= 0, self.counter <= 0 ==>
\<{ }\> self.counter >= 0
```

The second branch does not contain any program in the modality anymore, the modality is removed and the postcondition can be evaluated to true based on the assumptions. The first branch contains an assignment. Applying a corresponding Java Card DL rule results in a state update of the program under execution, which is represented in the following way:

```
self != null, self.counter >= 0, self.counter > 0 ==>
{self.counter := self.counter - 1}\<{ }\> self.counter >= 0
```

Java Card DL state updates are very much like Java assignments, except that they are only allowed to appear in their canonical form, in particular, the right hand side of an update has to be side effect free. Further steps in the proof remove the empty modality from the sequent and apply the state update on the formula that is attached to it:

```
self != null, self.counter >= 0, self.counter > 0 ==>
self.counter - 1 >= 0
```

This sequent is easily closed (proved) by simple integer arithmetic reasoning.

Obviously, this simple example cannot discuss all of the details of the logic, in particular, how dynamic method binding is done or how object aliasing is handled by the state update mechanism. All the details can be found in [1, Chapter 3]. However, the last thing we want to discuss here is the treatment of exceptions. The diamond modality requires that no exceptions are thrown. Nevertheless, one can easily construct a proof obligation stating that a method is allowed to throw a given kind of exception by simple program transformation:

```
exc = null & ... -> \<{ try { self.decreaseCounter(); }
  catch(SomeException e) { exc = e; } }\> self.counter >= 0
```

Here, `SomeException` possibly thrown by method `decreaseCounter` is caught by the `try-catch` block resulting in a non-abruptly terminating program. Moreover, it is possible to distinguish the abrupt termination state in the postcondition by making a case distinction on the value of `exc`. A non-null value of `exc` determines that an exception indeed occurred. This exception treatment is essential to how exceptions are treated when higher level specification languages are translated into JAVA CARD DL. The KeY system provides interfaces for both JML [8] and OCL [14]. A specification written in JML or OCL together with the associated code is automatically translated into JAVA CARD DL and then can be verified with the KeY prover. If the specification happens to state exceptional behaviour, e.g., with JML's `signals` clause, the mechanism described above is used during translation.

2.2 JAVA CARD Native Interface

Each JAVA CARD API implementation relies on a native interface to the underlying smart card execution environment (actual hardware or a simulator). Our implementation is meant to be symbolically executable in the KeY system. Thus, the native code interface has to be provided by KeY itself. For this purpose, we equipped the KeY system with a dedicated JAVA class with a number of JAVA CARD specific native methods. As a convention all such methods are named with a `jvm` prefix. Here is an excerpt from the `KeYJCSystem` class:

```
public static native byte jvmIsTransient(Object theObj);
public static native byte[] jvmMakeTransientByteArray(
  short length, byte event);
public static native void jvmBeginTransaction();
public static native void jvmCommitTransaction();
public static native void jvmArrayCopy(byte[] src, short srcOff,
  byte[] dest, short destOff, short  length);
```

Whenever the KeY system encounters a call to one of these methods an axiomatic JAVA CARD DL rule is used to reflect the result of execution in the KeY verifier. For example, a call like this:

```
\<{ transType = KeYJCSystem.jvmIsTransient(obj); ... }\> ...
```

results in a state update of the following form:

```
{transType := obj.<transient>}\<{ ... }\> ...
```

Here `<transient>` is an implicit attribute associated with each object in the KeY JAVA CARD model that indicates whether a given object is persistent (kept in card's EEPROM) or transient (kept in RAM). For example, the code resulting from the execution of `jvmMakeTransientByteArray` sets this attribute to `event` in the array that is being created. The value of `event` indicates on which event (card reset or applet deselection) the contents of the transient array should be cleared.

All of the native methods declared in the `KeYJCSystem` class have such corresponding Java Card DL axiomatic rules. Then it is possible to give the reference implementation of the Java Card API in terms of this native interface, for example:

```
public class JCSystem {
  public static byte isTransient(Object theObj){
      if(theObj == null) return NOT_A_TRANSIENT_OBJECT;
      return KeYJCSystem.jvmIsTransient(theObj); }

  public static byte[] makeTransientByteArray(short length, byte event)
    throws SystemException, NegativeArraySizeException {
      if(event != CLEAR_ON_RESET && event != CLEAR_ON_DESELECT)
        SystemException.throwIt(SystemException.ILLEGAL_VALUE);
      if(length < 0) throw KeYJCSystem.nase;
      return KeYJCSystem.jvmMakeTransientByteArray(length, event); } }
```

2.3 Specification Language

The KeY system supports specifications written in JML or OCL. OCL is not best suited for a case study like this one, it is relatively high level and not too closely coupled to Java [7]. A perfect solution would be to use JML, which provides a specification language closely related to Java. Moreover, large parts of existing JML specifications for Java Card API [9] could be reused. JML too, however, currently poses one major problem for this case study, namely, the semantics of the generated proof obligations (or rather method contracts associated with a given class and method specification), and the current inability of KeY to manipulate easily the way the contracts are generated. Without going into too much detail, we want our contracts to preserve invariants for the objects of our choice. In most cases this is simply the object a given method is invoked on, and possibly objects passed as parameters or stored in instance attributes. Currently the KeY system does not allow such fine grained selection of object invariants when generating proof obligations from JML specifications.

To solve this problem, we used Java Card DL itself as a specification language, i.e., provided readily generated Java Card DL contracts customised to our needs. This approach does not introduce any complication into the process of constructing specifications. The semantics of Java Card DL expressions and the actual syntax is very close to those of JML. The main difference is that a Java Card DL specification already constitutes a full method contract, thus, one has to manually specify which invariants for which objects are to be included in the precondition and the postcondition of the method. For example, suppose we have the following class with JML annotations:

```
public class MyClass {
  int a=0; //@ invariant a >= 0;
  /*@ requires val >= 0; ensures a = val; assignable a; @*/
  void setA(int val) { a = val; } }
```

Then, assuming that we want to establish preservation of the invariant only for one instance of the class (`self`), the corresponding Java Card DL contract takes the following form:

```
MyClass_setA_contract { \programVariables { MyClass self; int val; }
  self.a >= 0 & val >= 0 ->
    \<{ self.setA()@MyClass; }\> (self.a = val & self.a >= 0)
  \modifies { self.a } };
```

One more advantage of specifying contracts directly in Java Card DL is the possibility to take "shortcuts". For example, one can directly specify the persistency type of an object by referring to its `<transient>` attribute. In JML that would require including the `isTransient` method call in the specification. Such shortcuts improve considerably on the size of the resulting proofs.

Our approach of considering invariants for single object instances assumes that changes to one instance of an object cannot influence the invariant of another instance. That is, we assume there is no inter-object data aliasing. To get confidence that this is indeed the case we would also have to prove that data is properly encapsulated within objects. Currently we cannot do this in KeY in a simple way, proof obligations for proving encapsulation have to be created manually [1, Section 8.5.2]. For a case study of this size this is infeasible. It is of course also possible to employ other formal techniques to prove data encapsulation, for example data universes [3]. On the other hand, for the Java Card API this is not a big issue. Our implementation hardly ever copies data by reference and declares most of the relevant data private, which prohibits direct violation of other objects' invariants.

Finally, we should mention that the KeY JML front-end undergoes heavy refactoring at the moment (partly because of the described deficiencies). Once complete, verification based on JML version of our specification should be possible.

3 Implementation and Specification of the API

The Java Card API [11, 12] provides an interface for smart card specific routines. It is relatively small (60 classes and interfaces) and does not really share common features with the regular desktop Java API. Only the very basic classes (like `Object` and `NullPointerException`) are present in both APIs. Apart from that the Java Card API version 2.2.1 provides support for the following smart card specific features: Java Card applets, APDU (Application Protocol Data Units) communication, AID (Applet IDentifiers) registry lookup, owner PIN objects, the atomic transaction mechanism, Java Card inter applet object sharing through the Java Card applet firewall, the Java Card Remote Method Invocation (RMI) interface, cryptographic keys and ciphers, and simple Java Card utility routines. The specifics of the Java Card platform requires the API to have

a small memory footprint. Thus, Java Card does not support strings and associated classes, collections, etc. Moreover, most of the classes that are present in the API are modified to enable low resource usage. For example, cryptographic routines are implemented with a smaller amount of interfaces and methods (compared to Java Cryptography Extensions) and operate only on byte arrays.

Our reference implementation follows the official documentation as closely as possible. However, implementation of some features would be very difficult and the amount of work required would not compensate for the possible gains. Moreover, an over-engineered implementation would be very difficult to verify. Another reason for leaving out certain features is the inability to formally reason about them in KeY.

The first item on the unimplemented feature list are the cryptographic routines. Giving a functional implementation of ciphers and keys in Java that would be easy to understand and verify is simply infeasible. In fact, actual smart cards incorporate a cryptographic coprocessor and highly optimised native code is used for the implementation of ciphers and keys. Thus, our implementation does not contain any actual cryptographic routines. However, all the other features of the cipher and key classes are implemented. For example, the lengths of encryption blocks depending on the encryption algorithm are accurately calculated, or `CryptoExceptions` are reported on all conditions that do not involve checking the result of cryptographic calculations, e.g., that the key is initialised or that the plain text does not exceed its maximum allowed length.

The second unimplemented feature is the low-level APDU communication, i.e., the routines that are normally responsible for sending and receiving data from the card reader. Our implementation simply assumes that communication happens behind the scenes implicitly. This is not a real limitation. During formal verification of applet code it is sufficient to specify what the contents of the APDU buffer is. Knowing that it has in fact been transported to or from the card terminal is usually not necessary.

The third gap in the implementation are the routines related to RMI dispatching. Again, this would be possible, but very difficult to implement, resulting in a unjustifiably large code. On the other hand it is very easy to verify RMI based applet code without knowing the details of how RMI methods are dispatched. That is, it is not necessary to know how a given RMI method is marshaled or unmarshaled to verify its code. Moreover, even if we did implement the RMI dispatching routines in our API, it would not be possible to reason about them with the KeY system. Such reasoning would require (at least partial) support for class and method reflection which is not present in KeY at the moment.

Finally, the Java Card platform is capable of tracking and reporting memory consumption on the card through API methods. This is implementable only to certain extent, namely, dedicated methods for allocating transient memory can

keep track of transient memory usage. Tracking persistent memory usage is not possible. In principle, this would require hooking some Java code into the built-in `new` operator. On the other hand, it would be possible to delegate the job of tracking memory usage to KeY, i.e., in principle memory usage properties could be verified during symbolic execution. The support for reasoning about memory usage properties is yet another feature currently being integrated into KeY.

Apart from that our implementation includes all of the Java implementable features specified in the official Java Card documentation [11]. Notably, the following items are taken into account.

The Java Card firewall mechanism enforces object access checks on two levels, the Java Card VM level and the API level. All checks required on the API level are included in our implementation. The routines to provide shareable interface objects to client applets across the firewall are also implemented. In the KeY system, the modelling of the checks on the VM level requires changes to the Java Card DL. The work to incorporate the firewall mechanism formalisation into Java Card DL is underway. Without this formalisation, the API firewall checks are transparent during verification. All objects in verified Java Card programs are treated as if they are owned by the Java Card system, i.e., all objects are privileged and access is always allowed.

All features related to transaction mechanism and memory types (persistent or transient) are included. In particular, it means that (i) methods specified in the documentation to be atomic utilise the transaction mechanism in a suitable way, (ii) data that is required to be transient is kept in transient memory blocks, and (iii) updates to all data that are to be excluded from the transaction mechanism are implemented in a suitable way. A notable example of the last is the PIN try counter [4]. The KeY system fully supports the Java Card transaction mechanism and different memory types, and thus all of the code involving transactions can be faithfully specified and verified with KeY.

All cryptographic interfaces (ciphers and keys) have associated implementing classes, but do not include the actual cryptographic logic as described above. A possibility to declare a cipher to be shareable or non-shareable between different applets, or for a key to implement internal key data encryption (the `KeyEncryption` interface) are both included in the implementation.

A lightweight applet registry is implemented to track applet identifiers (AID registry) and applet installation, activation, and selection. A possibility of an applet to be multi-selectable is also taken into account. The registry is minimal in the sense that it is just sufficient to provide meaningful results to methods of the API that require the applet registry functionality, e.g., the method `getAID` of the `JCSystem` class.

In the remainder of this section we give two samples of our implementation and associated specifications. The first example is the implementation of the method `partialEquals` of the `AID` class. The method is simply responsible for

comparing `length` bytes of the provided byte array to the AID bytes stored in the object. The comparison itself is simply a call to the `arrayCompare` utility method. First, however, some checks for the firewall mechanism have to be performed:

```
public final boolean partialEquals(byte[] bArray,
  short offset, byte length) throws SecurityException,
    ArrayIndexOutOfBoundsException {
  if (bArray==null) return false; // resp. documentation
  if(length > _theAID.length) return false; // resp. documentation
  // Firewall check:
  if (KeYJCSystem.jvmGetContext(KeYJCSystem.jvmGetOwner(bArray))
      != KeYJCSystem.jvmGetContext(
           KeYJCSystem.jvmGetOwner(KeYJCSystem.previousActiveObject))
  && KeYJCSystem.jvmGetPrivs(bArray) != KeYJCSystem.P_GLOBAL_ARRAY)
     throw KeYJCSystem.se; // System owned singleton instance
  // Actual comparison:
  return Util.arrayCompare(bArray, offset, _theAID, (short)0, length)==0;
}
```

The firewall check establishes that the caller of this method (`previousActiveObject`) was privileged to access the `bArray` parameter. If not, a system owned singleton instance of `SecurityException` is thrown. The reason for storing singleton instances of all exceptions is to follow the Java Card paradigm of limiting the memory consumption, and also to separate system owned exceptions from applet owned ones. The calling of the method `arrayCompare` may result in an `ArrayIndexOutOfBoundsException`, which is allowed according to the documentation of `partialEquals`. The formal specification for this method is the following:

```
\programVariables { AID aidInst; boolean result;
  byte[] bArray; short offset; byte length; }

(bArray != null -> length >= 0 & offset >= 0 &
  offset + length <= bArray.length)
& {\subst AID aid; aidInst}(\includeFile "AID_inv.key";)
-> \<{
     result = aidInst.partialEquals(bArray, offset, length)@AID;
   }\> (
    (bArray = null | length > aidInst._theAID.length -> result = FALSE)
  & (bArray != null & length <= aidInst._theAID.length ->
      (result = TRUE <-> \forall int i; ( i >= 0 & i < length ->
         aidInst._theAID[i] = bArray[offset+i])))
  & {\subst AID aid; aidInst} (\includeFile "AID_inv.key";))
\modifies {result}
```

The first part of the precondition guarantees that no `ArrayIndexOutOfBoundsException` would be thrown. The second part assumes the class invariant (for easy reuse stored in a separate file) for the execution of the method:

```
  aid._theAID != null & aid._theAID.<created> = TRUE
& aid._theAID.<transient> = JCSystem.NOT_A_TRANSIENT_OBJECT
& aid._theAID.length >= 5 & aid._theAID.length <= 16
```

The byte array storing the AID should not be `null`, should be allocated in the persistent memory, and its length should be between 5 and 16 according to the documentation.

The postcondition describes the value of the result in detail. It is true if and only if the first `length` bytes in the provided array `bArray` starting at `offset` are equal to `length` bytes stored in the `_theAID` instance attribute. Additionally the invariant for the AID class has to be reestablished after the method executes. Finally, this method does not modify any data, except for the local `result` variable.

The second example we want to present is the `throwIt` method of one of the Java Card specific exception classes – `TransactionException`. Although the implementation and the specification of this and sibling methods are very simple they are quite important. Such methods are frequently used both in the rest of our API implementation as well as in many Java Card applets. The specific feature of these methods is that it only provides exceptional behaviour, i.e., its sole purpose is to throw a system owned instance of a given exception:

```
public static void throwIt(short reason) throws TransactionException {
  _instance.setReason(reason);
  throw _instance; }
```

The `throwIt` method is static and its execution is guarded with a corresponding static invariant, which simply says that the static attribute storing the singleton instance of the exception (`_instance`) is not `null`. Additionally, for this particular instance the instance invariant for the exception class should be maintained, which states that the `_reason` array is properly allocated in transient memory. Reason codes of exceptions should be cleared every time the card loses power, so the variable storing the reason code needs to be allocated in a transient memory. In Java Card only arrays can be allocated in transient memory. Thus, the reason code has to be stored in a `short` array of size 1 instead of a simple `short` attribute. The static and the instance invariant are part both of the method's precondition and postcondition:

```
    (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
        (\includeFile "TransactionException_inv.key";)
-> \<{ #catchAll(TransactionException t) {
        TransactionException.throwIt(reason)@TransactionException;
      } }\>
  ( t = TransactionException._instance & t._reason[0] = reason
  & (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
      (\includeFile "TransactionException_inv.key";))
\modifies { TransactionException._instance._reason[0] }
```

This contract describes the exceptional behaviour of the method. The `#catchAll` construct declares that the method can *possibly* throw an exception of the declared type. The value `t` representing the thrown exception can be checked in the

postcondition. A null value indicates no exception (normal behaviour), a non-null value indicates that the exception indeed occurred (exceptional behaviour). In the postcondition it is required that `t` is equal to the singleton instance of the exception, and so is not null by the assumption. Thus, this postcondition requires the method to throw the exception. Finally, the postcondition also specifies that the reason code of the thrown exception (a corresponding location is included in the `\modifies` clause) is equal to the parameter of the method.

One may argue that the specification for the method `throwIt` is over-engineered, the contract for the method is actually bigger than the code of the method itself. In fact, for most of the practical applications, a much simpler specification would suffice. However, we treat specifications like this as an exercise for the KeY system. It shows that detailed verification w.r.t. complex specifications is easily achieved.

4 Verification and Experience

All of the methods have been specified and verified with the KeY system. That includes the simplest methods that just return a value of an instance attribute, but also the most complex and elaborate methods, like the `buildKey` of the `KeyBuilder` class or all of the methods of the `Cipher` implementation. The proofs were performed in a fully modular way. Whenever a method was calling another method in the API, a corresponding contract was used to discharge the method call, i.e., the proofs were always performed by contract application, instead of in-lining the code of the called method. It turned out in the process that the approach of applying method contract is the only feasible one. For a case study like this one in-lining of method calls results in proofs of unmanageable size.

The level of automation of the proofs is satisfactory, the majority of the methods are proved fully automatically, most of the rest require minor interactions, like simple quantifier instantiations. The only really heavy spots w.r.t. user interaction are loops (10 in total). Since our proof obligations require termination, a suitable specification for each of the loops has to be provided: the loop invariant, modification set, and loop variant. For at least two of the loops the loop invariant turned out to be quite complex and far from obvious just by looking at the code, a careful analysis of the open proof goals was necessary. Finally, it was not necessary to involve external tools to support verification. The KeY system allows to employ external decision procedures to discharge first order logic formulae, e.g., the Simplify theorem prover. For this case study the KeY prover was able to discharge all proof goals on its own.

On a darker side, some of the proofs were very heavy on computing resources. It was not uncommon for the prover to use up to 1.5GB of heap space and run for over an hour to finish a proof for one method. Such performance certainly

makes the round-trip specification engineering infeasible. For this case study one possible solution to this problem is to rewrite parts of the API implementation to improve on the prover performance. It turned out, for example, that the `switch` statements sometimes cause large growth of the proof. It is our belief that rewriting those `switch` statements into highly optimised `if` statements would partly solve the problem. This matter is currently under investigation. Moreover, for some of the proofs a minor modification of the KeY's automatic rule application mechanism was necessary to prevent proof size blowup. The modification in question is more of a hack that happens to work for this case study and not yet a proper solution to the problem.

Finally, the careful analysis of the Java Card documentation allowed us to identify hot spots in the specification, places where actual card implementations are likely to be incorrect due to, e.g., documentation ambiguity or unusual complexity. Indeed, we did find a bug in one of the commercially sold cards. One of the firewall rules [12, Section 6.2.8.6] is ignored resulting in granting access to a shareable object in a situation where it is forbidden by the specification.

5 Conclusions and Future Work

We presented a formally verified reference implementation of the Java Card API. The level of detail of the implementation is relatively high considering that the running environment of the implementation is a symbolic execution environment, the KeY verification system. This API implementation will serve us as a framework for verifying various Java Card applets in our project. All of the implementation has been formally specified and verified with KeY. We found the verification process feasible, however, we do have some reservations to the performance of the KeY system. After some clean-ups and minor fixes to the code and the specification we will make the case study available on the web.

For the future we plan to look into the following. First we want to modify the API implementation code to improve on the verification performance. Secondly, our experience will be used to rectify the issues and problems we found in the KeY system (we have already communicated the most pressing issues to the KeY development team). Next we plan to implement the firewall functionality in the KeY logic. Then it will be possible to verify the API implementation again to make sure that the implemented firewall checks are consistent. The fourth step is to rewrite all of our specifications in JML. Here the work on improving the KeY's JML interface has to be finished first. Finally, it could be worthwhile to update our implementation to the newest stable version of the Java Card API 2.2.2 [13], which introduced some minor updates. At the time we started our work the version 2.2.2 was not yet official. Moreover, none of the cards on the market actually implement Java Card 2.2.2, thus, for the practical purpose of verifying realistic applet code the version 2.2.1 is sufficient.

Acknowledgements This work is supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs. We would also like to thank Christian Haack, Erik Poll, Jesús Ravelo, and anonymous reviewers for their helpful comments.

References

1. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
2. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.
3. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.
4. Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006.
5. Engelbert Hubbers and Erik Poll. Transactions and non-atomic API calls in Java Card: Specification ambiguity and strange implementation behaviours. Department of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
6. Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
7. Daniel Larsson and Wojciech Mostowski. Specifying Java Card API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.
8. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
9. Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*, volume 2140 of *LNCS*, pages 165–178. Springer, September 2001.
10. Quang Huy Nguyen and Boutheina Chetali. Certifying native Java Card API by formal refinement. In *Smart Card Research and Advanced Applications, 7th IFIP WG 8.8/11.2 International Conference, CARDIS 2006, Tarragona, Spain, April 19–21, 2006, Proceedings*, volume 3928 of *LNCS*, pages 313–328. Springer, 2006.
11. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.1 API Specification*, October 2003.
12. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.1 Runtime Environment Specification*, October 2003.
13. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 API Specification*, March 2006.
14. Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.

Automated Formal Verification of PLC Programs Written in IL

Olivera Pavlovic[1], Ralf Pinger[1] and Maik Kollmann[2]

[1] Siemens Transportation Systems,
Ackerstrasse 22, D-38126 Brunswick, Germany,
{Olivera.Jovanovic,Ralf.Pinger}@siemens.com
[2] Brunswick Technical University, Institute of Information Systems,
Mühlenpfordtstrasse 23, D-38106 Brunswick, Germany,
M.Kollmann@tu-bs.de

Abstract. Providing proof of correctness is of the utmost importance for safety-critical systems, many of which are based on Programmable Logic Controllers (PLCs). One widely used programming language for PLCs is Instruction List (IL). This paper presents a tool for the fully automated transformation of IL programs into models of the NuSMV (New Symbolic Model Verifier) model checker. For this, the tool needs a metadescription of the IL language. This broadens the scope of the software and allows the tool be used for programs written in many other low-level languages as well. Its application is demonstrated using a typical IL program, at the same time providing insights into the proposed automation of the process of formal verification of PLC programs. This automatic verification should provide a powerful analysis method with a wide industrial application.

Key words: automated verification, model checking, NuSMV (New Symbolic Model Verifier), Programmable Logic Controller (PLC), Instruction List (IL)

1 Introduction

Programmable Logic Controllers (PLCs) are a special type of computer used in automation systems. Generally speaking, they are based on sensors and actuators, which have the abilty to control, monitor and interact with a particular process, or collection of processes. These processes are diverse and can be found, for example, in household appliances, emergency shutdown systems for nuclear power stations, chemical process control and rail automation systems.

The programming of PLCs is achieved with the help of five languages, standardised by the International Electrotechnical Commission (IEC) in [IEC93]: (a) two textual languages: Instruction List (IL) and Structured Text (ST), and (b) three graphical languages: Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). This paper focuses on the formal verification of programs written in IL, which is a low-level, machine-orientated language.

The simulation of IL programs and their automated transformation to VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) are discussed in [Fig06]. The theoretical basics for the verification of IL programs can

be found in [CCL+00]. Further research on the topic was published in [PPKE07], which examines in more depth the handling of PLC hardware by the formal verification of IL programs. For this, a specific PLC is selected, although the same principles can be applied to any PLC. We present an enhancement of the works cited above describing how function calls can be handled by the PLC verification and presenting a tool for the fully automated transformation of IL programs into the NuSMV models. By applying the tool to a typical IL program, we demonstrate its successful application and show how the process of formal verification of PLC programs can be automated. Based on the lessons learnt from the tool, we also propose an improvement in the verificaton method.

The rest of the paper is structured as follows: Section 2 briefly reviews the method/formalism of model checking. In Section 3 the structure of an IL program is outlined and a detailed description of a behavioural model of the program also given. Section 4 presents the tool developed for the transformation of IL programs into NuSMV models. In Section 5 a case study illustrates the verification of IL programs. Finally, conclusions are drawn and plans for the future proposed.

2 Model Checking

Automated verification techniques such as model checking have become a standard for proving the correctness of state-based systems. Model checking is the process of checking whether a given model M satisfies a given logical formula φ. Model checking tools such as SPIN [Hol97] and SMV/NuSMV [McM96,CCB+02] incorporate the ability to illustrate that a model does not satisfy a checking condition using a textual, tabular or sequence chart-like representation of the relevant states.

The model M has to be translated into the input language of a model checking tool. For this, a state transition system can be used, which defines a kind of non-deterministic finite state machine representing the behaviour of a system. The transition system can be represented by a graph whose nodes represent the states of the system and whose edges represent state transitions. A state transition system is defined as follows.

Definition 1. *State transition system*
A system $T = (\mathcal{S}, \mathcal{S}_0, \rightarrow)$ *with*

- $\mathcal{S}$*: a non-empty set of states,*
- $\mathcal{S}_0 \subseteq \mathcal{S}$*: a non-empty set of initial states,*
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$*: a transition relation which is defined for all* $s \in \mathcal{S}$

is called a state transition system.

In the sections below we will limit ourselves to transition systems and temporal logic formulas, which are both suitable for capturing the behaviour of our programs and representing typical checking conditions. The model checking problem can be stated as follows: Let a checking condition be given by a temporal logic formula φ, and a model M with an initial state s, then it must be decided

$$M, s \models \varphi.$$

If M is finite, the model checking is reduced to a graph search. In our case, Linear Temporal Logic (LTL) [Pnu77] is suitable for the encoding of the properties. LTL is a subset of CTL* with modalities referring to time (cf. [HR00,CGP00]). The syntax of the LTL formula is given by the following Backus-Naur-Form (BNF) definition:

$$\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi\ \mathsf{U}\ \varphi) \mid (\mathsf{G}\ \varphi) \mid (\mathsf{F}\ \varphi) \mid (\mathsf{X}\ \varphi).$$

In addition to the propositional logic operators and predicates p, the temporal operators $\mathsf{X}, \mathsf{F}, \mathsf{G}$ and U are interpreted as follows:

- $\mathsf{X}\ \varphi : \varphi$ must hold at the ne**x**t state.
- $\mathsf{F}\ \varphi : \varphi$ must hold at some **f**uture state.
- $\mathsf{G}\ \varphi : \varphi$ must hold at the current state and all future state (**g**lobally).
- $\psi\ \mathsf{U}\ \varphi : \varphi$ holds at the current or a future state, and ψ must hold up **u**ntil this point. From this point, ψ no longer needs to hold.

LTL formulas are evaluated for a certain path π of states. If we let $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ be a path of states, then π_i is the suffix starting at s_i: $\pi^i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. All temporal logic operators can be related to path expressions, e.g. the ne**x**t-operator's semantics are given by $\pi \models X\phi$ iff $\pi^2 \models \phi$.

Techniques based on Büchi automata have been implemented in SPIN to check if a system meets its specifications. This is done by synthesising an automaton which generates all possible models of the given specification and then checking if the given system refines this most general automaton. SMV and NuSMV employ tableau-based model checking in order to evaluate whether a given LTL formula φ_{LTL} holds. They were originally symbolic model checking tools relying on binary decision diagrams. The set of states satisfying a CTL formula φ_{CTL} is computed as the BDD representation of a fixed point of a function. If all the initial system states are in this set, φ_{CTL} is a system property.

3 PLCs

As already stated, PLCs are a special type of computer based on sensors and actuators able to control, monitor and influence a particular process. There

are many standard tools for the configuration of PLCs, depending on the PLC product family. In these tools, the PLC programming languages standardised in [IEC93] are usually given different names to those in the IEC standard. Thus, the tool used for this study supports, among others, the Statement List (STL) programming language. STL corresponds in its expressiveness one to one to IL, having instructions with the same functionality. The syntax of the two languages differs, however. In the remainder of this paper the designation IL will be used to cover both IL and STL.

3.1 IL Program

An IL program can consist of a number of modules. Each of the modules and the main program contain variable declarations plus a program body. For the purpose of verification, we shall consider the program body as a limited set of lines of code executed in a defined sequence. Let us consider a program $\mathcal{P}$ having $maxpc$ lines of code and n modules $\mathcal{P}_1, \ldots, \mathcal{P}_n$ each having $maxpc_\imath, \imath = 1, \ldots, n$ lines of code. The program $\mathcal{P}$ can then be represented as follows:

$$\mathcal{P} = \{(\jmath, statement_\jmath) \mid \jmath = 1, \ldots, maxpc\} \cup \bigcup_{\imath \leq n} \mathcal{P}_\imath, \text{ where}$$

$$\mathcal{P}_\imath = \{(\jmath_\imath, statement_{\jmath_\imath}) \mid \jmath_\imath = 1, \ldots, maxpc_\imath\}, \text{ for all } \imath = 1, \ldots, n$$

where $statement_\jmath$ ($statement_{\jmath_\imath}$) designates the statement at line $\jmath$ ($\jmath_\imath$) of $\mathcal{P}$ ($\mathcal{P}_\imath$).

3.2 Behavioural Model of an IL Program

The behavioural model of an IL program $\mathcal{P}$ can be represented using a state transition system $\mathcal{T} = (\mathcal{S}, \mathcal{S}_0, \rightarrow)$, where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ a non-empty set of initial states and $\rightarrow$ a transition relation. $\mathcal{S}$, $\mathcal{S}_0$ and $\rightarrow$ are constructed as follows:

Set of states $\mathcal{S}$. The set of states $\mathcal{S} = \mathcal{S}_S \times \mathcal{S}_H \times \mathcal{S}_{PC}$ with

$\mathcal{S}_S$ *- a set of states of software-specific variables.* Software-specific variables are variables defined within the program $\mathcal{P}$. Although IL supports a wide range of data types (relating to its hardware-like nature), in this paper only Booleans and bounded integers are discussed. Let us consider the main program, $\mathcal{P}$, with n_P variables and each module $\mathcal{P}_\imath$ with $n_{P_\imath}$ variables, then $\mathcal{S}_S = \mathcal{S}_{SP} \times \mathcal{S}_{SP_1} \times \ldots \times \mathcal{S}_{SP_n}$ where $\mathcal{S}_{SP} = SP_{var_1} \times \ldots \times SP_{var_{n_P}}$ and $SP_{var_\jmath}$ is a domain of the variable $var_\jmath$ for $\jmath = 1, \ldots, n_P$. For example, for a Boolean variable $var_\jmath$, $SP_{var_\jmath} = \{true, false\}$. The sets $\mathcal{S}_{SP_\imath}$ are defined analogously to $\mathcal{S}_{SP}$.

$\mathcal{S}_H$ *- a set of states of hardware-specific variables.* Depending on the PLC family, various types of CPU register are required for the processing of IL statements. Some of the registers are not important for the verification of IL programs ([PPKE07]). For this reason, only the following registers are considered here: three status bits, two accumulators and a nesting stack (used to save certain items of information before a nesting statement is processed). These registers are represented in the behavioural model of an IL program by hardware-specific variables. The set of states of hardware-specific variables $\mathcal{S}_H$ is constructed as follows. $\mathcal{S}_H = \mathcal{S}_{HP} \times \mathcal{S}_{HP_1} \times \ldots \times \mathcal{S}_{HP_n}$ where the sets in the product correspond to sets of hardware-specific variables of $\mathcal{P}$, $\mathcal{P}_1$, ..., $\mathcal{P}_n$ respectively. Because these sets are equivalent, it is sufficient to define one of them, e.g. $\mathcal{S}_{HP}$. $\mathcal{S}_{HP} = HP_{StatusBits} \times HP_{Accumulators} \times HP_{NestingStack}$ and $HP_{StatusBits} = HP_{RLO} \times HP_{OR} \times HP_{FC}$ (HP_{RLO}, HP_{OR} and HP_{FC} are domains of status bits RLO, OR and FC, that is $\{true, false\}$) and $HP_{Accumulators} = HP_{ACC_1} \times HP_{ACC_2}$ (HP_{ACC_1} and HP_{ACC_2} are accumulator domains). If we let $HP_{NestingStack}$ have l layers, then $HP_{NestingStack} = HP_{Stack_1} \times \ldots \times HP_{Stack_l}$. Each of these stacks contains information to be pushed into the stack before opening a new nesting operation. These are the status bits RLO and OR, and the identifier of the operation before nesting. Thus, $HP_{Stack_\jmath} = HP_{RLO_\jmath} \times HP_{OR_\jmath} \times HP_{Operation_\jmath}$, $\jmath = 1, \ldots, l$, where $HP_{Operation_\jmath}$ is a domain of operation identifiers.

$\mathcal{S}_{PC}$ *- a set of states of program counters.* The program counter of each of the program modules together form the set $\mathcal{S}_{PC}$. Thus, $\mathcal{S}_{PC} = \{1, \ldots, maxpc\} \times \{1, \ldots, maxpc_1\} \times \ldots \times \{1, \ldots, maxpc_n\}$.

Set of initial states $\mathcal{S}_0$. $\mathcal{S}_0 \subseteq \mathcal{S}$, or more precisely $\mathcal{S}_0 = \mathcal{S}_{S_0} \times \mathcal{S}_{H_0} \times \mathcal{S}_{PC_0}$ with $\mathcal{S}_{S_0} \subseteq \mathcal{S}_S$, $\mathcal{S}_{H_0} \subseteq \mathcal{S}_H$ and $\mathcal{S}_{PC_0} \subseteq \mathcal{S}_{PC}$. Sets $\mathcal{S}_{S_0}$ and $\mathcal{S}_S$ may differ merely in the ranges of the variables which can have predifined values for $\mathcal{P}$. Only for this kind of variable can initial values be restricted. For all other software-specific variables, all possible values have to be considered from the very beginning of verification. The initial values of the hardware-specific variables are predefined and identical for each $\mathcal{P}$ and $\mathcal{P}_\imath$, $\imath = 1, \ldots, n$. These variables are initialised with all Booleans being set to $false$ and all integers to 0. Thus, $\mathcal{S}_{H_0}$ is defined by $\mathcal{S}_{H_0} = \prod_{\imath \leq n+1} \mathcal{S}_{HP_0}$, $\mathcal{S}_{HP_0} = HP_{StatusBits_0} \times HP_{Accumulators_0} \times HP_{NestingStack_0}$, $HP_{StatusBits_0} = \{false, false, false\}$, $HP_{Accumulators_0} = \{0, 0\}$ and $HP_{NestingStack_0} = \prod_{\imath \leq l} \{false, false, 0\}$.

Transition relation $\rightarrow$. $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ describes how the state of the model changes after the execution of each statement. These changes are reflected in the fact that new values are assigned to the software and hardware variables, and the program counter. After each statement, the program counter is given a

new value, pointing to the next statement to be executed. Only one software-specific variable, which at the same time is the statement argument, can be changed by a single statement. On the other hand, one statement can change a number of hardware-specific variables. For more details of, how the transition relation and behavioural model are constructed, see [PPKE07].

4 Automated Transformation of IL Programs

The automated transformation of IL programs described in [PPKE07] was developed as a part of a masters thesis [Fen07]. As shown in Fig.1, besides the program to be transformed, the software also needs a description of the IL language. This description is supplied in an IL metafile (cf. Fig.2). The result of the automated transformation of the IL program is a corresponding NuSMV model. In some cases it is possible to reduce the state space of the resulting NuSMV model by manual optimisation. More details of the above steps are given in the sections below. On the basis of the NuSMV model and specification being proven, the next step in the verification is performed by the NuSMV model checker.

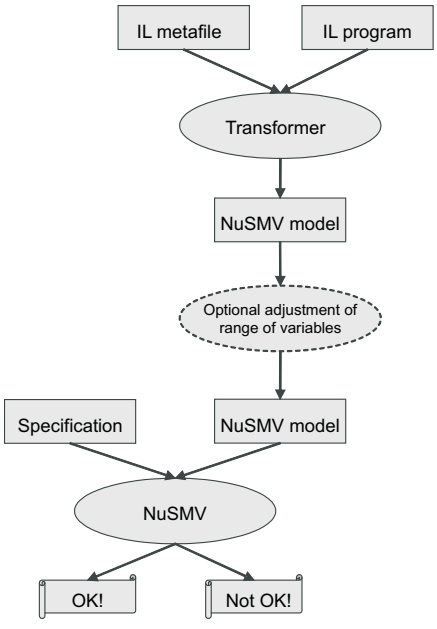


Fig. 1. Transformation process

4.1 Metafile

The metadescription of the IL language is defined in a simple text file with a special format (metafile). Part of the contents of the IL metafile is shown in Fig.2. This simple description of the language makes the software more universal and allows us to use it for the transformation of programs written in a number of other low-level languages as well, provided a metadescription of the language.

As shown in Fig.2, the system hardware variables must be described at the beginning of the metadescription. This is done using *identifier, type, initial value* triples. For example, *RLO,0,0* means that the status bit *RLO* is a Boolean (type 0) and has the initial value 0, and *ACC1,1,0* means that *ACC*1 is an integer (type 1) with the initial value 0. In the second part of the metafile, the IL statements are described. There are four types: 0-statements - with an argument, 1-statements - with a nesting operation, 2-statements - with an effect on the program counter, and 3-statements - with no argument. Accordingly, the conjunction *A argument* is of type 0 and described by

$$A,0,if(FC=1)\{RLO:=OR||RLO\&\&\_ARG\_);\}$$
$$else\{RLO:=\_ARG\_;FC:=1;\}$$

This means: if *FC* is true, the *RLO* bit is set to *OR ||(RLO && argument)*, otherwise *RLO* is set to *argument* and *FC* is set to *true*.

```
[variables]
RLO,0,0
OR,0,0
FC,0,0
ACC1,1,0
ACC2,1,0
...[meta]
A,0,if(FC=1){RLO:=OR||(RLO&&_ARG_);}else{RLO:=_ARG_;FC:=1;}
JU,2,PC:=_ARG_;
+I,3,ACC1:=ACC2+ACC1;
*I,3,ACC1:=ACC2*ACC1;
>I,3,if(ACC2>ACC1){RLO:=1;OR:=0;FC:=1;}else{RLO:=0;OR:=0;FC:=1;}
<I,3,if(ACC2<ACC1){RLO:=1;OR:=0;FC:=1;}else{RLO:=0;OR:=0;FC:=1;}
...
```

Fig. 2. Metadescription of the IL language

4.2 Manual Optimisation of the NuSMV Model

In some cases the NuSMV model resulting from the transformation of the IL program will have an optimisation facility. The model optimisation is optional and has to be performed manually. An illustration of this will now be given. Let

us consider some integer variables in an IL program having a restricted range of integer values. Despite the limitation of the variables, a whole range of integers is reserved for them. These variables do not need the entire range of integers in the corresponding NuSMV model, and problems may result due the excessive size of the model's state space. Provided the ranges of the variables are known, they can be adjusted accordingly and the space requirement reduced.

5 Case Study

This section takes a closer look at the process of formal verification of IL programs already described. An IL program and the corresponding NuSMV model are presented. To show the behavioural equivalence between the IL program and its NuSMV model, the method proposed in [PH07] can be applied. The most complex issue in the verification process turns out to be the implementation of a function call. We have therefore chosen to demonstrate how this is done on the basis of a sample IL program. For more about IL programming [Gie03] and [Sie04] should be consulted.

5.1 IL Program

An outline of the program considered here (*DemonstrateFormByte*) is shown in Fig.3. This simple IL program demonstrates the call of the function *FormByte* which takes 8 bits as input (*Bit0, Bit1,...*, *Bit7*) and combines them into one byte (*Byte*).

5.2 NuSMV Model

A NuSMV program consists of several modules. There must be one module with the name *main* and no formal parameters ([CCB+02]). Accordingly, the IL program is implemented by the main module in NuSMV and the function called by a further module, which is instantiated in the main module (cf. Fig.4). For more information about how IL statements are transformed into NuSMV model, see [PPKE07].

Let us consider the transitions given in Fig.4. The program *DemonstrateFormByte* has 2 lines, in the first of which the function *FormByte* is called. This call is implemented in the NuSMV model by saying in the main module that if program counter is equal to 1 and *FormByte* has not finished executing, the program counter of the main module does not change its value. Only when *FormByte* has finished its execution may the main module program counter increment its value.

```
//Program "DemonstrateFormByte"
//8 boolean and 1 integer variables
//program body
CALL "FormByte" (
   InputBit0 := Bit0, InputBit1 := Bit1, InputBit2 := Bit2,
   InputBit3 := Bit3, InputBit4 := Bit4, InputBit5 := Bit5,
   InputBit6 := Bit6, InputBit7 := Bit7, OutputByte := Byte);
//Function "FormByte"
//input: 8 booleans (InputBit0,...,InputBit7)
//output: 1 integer (OutputByte)
//1 temporary variable (Value) which keeps temporary result
//function body
   L 0; //Initialise the temporary result
   T Value; //Value=0
   AN InputBit0; //Check if Bit0 is set
   JC BIT1; //if not jump to BIT1
   L 1; //Adjust the temporary result by 2**0
   T Value; //Value=1
BIT1: AN InputBit1; //Check if Bit1 is set
   JC BIT2; //if not jump to BIT2
   L Value; //Adjust the temporary result by 2**1
   L 2;
   +I ;
   T Value; //Value=Value+2
   ...
```

Fig. 3. An outline of the IL program demonstrating the calling of the function which combines the eight bits supplied into one byte

```
MODULE FormByte(param0,param1,param2,param3,param4,param5,param6,param7)
...
MODULE main
//...variable declaration
//the instantiation of the FormByte module is realised in the next
line
CALL_FormByte : FormByte(Bit0,Bit1,Bit2,Bit3,Bit4,Bit5,Bit6,Bit7);
ASSIGN
next(PC) :=
   case
     PC=1 & CALL_FormByte.PC<63: 1;
     PC=1 & CALL_FormByte.PC=63: 2;
     1 : PC;
   esac;
init(PC) := 1;
...
```

Fig. 4. An outline of the NuSMV model corresponding to the IL program DemonstrateFormByte

5.3 Specification and Verification Results

To prove the correctness of the NuSMV model we need to check if the byte value obtained corresponds to the eight bits supplied. This property can be represented by the following LTL formula:

$$\mathsf{G}(PC = 2 \Rightarrow Byte = (Bit0 + 2 * Bit1 + 4 * Bit2 + 8 * Bit3 + 16 * Bit4 + 32 * Bit5 + 64 * Bit6 + 128 * Bit7))$$

Unfortunately, proving this property by the method described is inefficient. It took over eight hours to do so. The ultimate aim of this work study, however, is to apply the proving technique to far more complex case studies. Hence, the approach needed to be improved. How this was done, is described in the next section.

5.4 Improvement of the Method

The reason for the inefficiency of the verification lay in the enormous number of transitions which had to be considered by the NuSMV model checker when instantiating a new module in a main module. More precisely, all the variables that formed the state space of the module *FormByte* ware also part of the state space of the main module. These variables are, however, of no importance before and after the module *FormByte* is referenced in the main module (hereafter this situation will to be referred to as *module FormByte is not active*).

Considering the above, a constraint was required in the main module with the meaning: "if the *FormByte* module is not active, the model checker only checks the states in which the *FormByte* variables are set to their initial values". This could be achieved by means of the following invariant:

```
INVAR (PC!=1 -> (CALL_FormByte.PC=1  |  CALL_FormByte.PC=64) &
CALL_FormByte.Bit0=0 & CALL_FormByte.Bit1=0 & CALL_FormByte.Bit2=0 &
CALL_FormByte.Bit3=0 & CALL_FormByte.Bit4=0 & CALL_FormByte.Bit5=0 &
CALL_FormByte.Bit6=0 & CALL_FormByte.Bit7=0 & CALL_FormByte.Byte=0 &
CALL_FormByte.Value=0 & CALL_FormByte.RLO=0 & CALL_FormByte.OR=0 &
CALL_FormByte.ACC1=0 & CALL_FormByte.ACC2=0)
```

Besides the addition of this invariant to the main module, some changes to *FormByte* were necessery. In order to enable the *FormByte* variables to have their initial values when the module was inactive, some new transitions had to be added. These transitions needed to set the variables to the predefined values once execution of the *FormByte* function had terminated. For this, the *FormByte* program counter was incremented by 1. Additionally, for each variable a new transition was formed, which set the variable to its initial value.

By adding this invariant to the model we succeeded proving the relevant property in 113.8 seconds, a vast improvement on the previous result. Thus the changes described brought about a marked improvement in our method.

6 Conclusion and Future Work

The safety demands of many systems based on PLC are considerable. The formal verification of the PLC software is thus of great importance. The verification method demonstrated in this paper is a powerful instrument for analysing safety-related software.

An approach was presented for the automated transformation of IL programs into NuSMV models. This is supported by a tool, which was also described. The efficiency and convenience of the tool were demonstrated by means of a case study. Because the transformation can be automated, the approach has the potential for a wide application in industry.

Although the approach is stable there is, however, still scope for improvement. While the tool was being developed aspects for optimisation were identified and appropriate features implemented directly; others are due to be incorporated in the near future (the invariant described in the previous section, for example, is to be automatically added to the NuSMV model). The aim of the project is to efficiently verify PLC software of as high a complexity as possible. In order to achieve this goal we will need to continue refining the technique. Thus, the development of the approach itself and the accompanying tool are ongoing. Further work in this field should provide a suitable set of reduction methods which will allow the state explosion problem, arising from the growing model size, to be resolved.

References

[CCB+02] Roberto Cavada, Alessandro Cimatti, Marco Benedetti, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. NuSMV: a new symbolic model checker. http://nusmv.itc.it/, 2002.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA, Oct. 2000*, pages 2449–2454, 2000.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 2000.

[Fen07] G. Fendoglu. Überführung eines AWL-Modells in ein NuSMV-Modell. Masters thesis. Technische Universität Braunschweig, 2007.

[Fig06] C. Figura. Überdeckungstests für fehlersichere Funktionspläne auf Basis einer geeigneten Überführung. Master thesis. Martin-Luther-Universität, 2006.

[Gie03] Walter Giessler. *SIMATIC S7 SPS-Einsatzprojektierung und -programmierung*. VDE Verlag GMBH, 2003.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295, 1997.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[IEC93] IEC. *International Electrotechnical Commission Standard 61131-3, Programmable controllers - Part 3*, 1993.

[McM96] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, second edition, 1996.

[PH07] J. Peleska and E. Haxthausen. Object Code Verification for Safety-Critical Railway Control Systems. In E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*. GZVB, 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[PPKE07] O. Pavlovic, R. Pinger, M. Kollmann, and H. D. Ehrich. Principles of Formal Verification of Interlocking Software. In E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*. GZVB, 2007.

[Sie04] Siemens. *SIMATIC Anweisungsliste (AWL) fuer S7-300/400. Referenzhandbuch (SIMATIC Instruction List for S7-300/400. Reference Manual)*, 2004.

Combining Deduction and Algebraic Constraints for Hybrid System Analysis*

André Platzer

University of Oldenburg, Department of Computing Science, Germany
platzer@informatik.uni-oldenburg.de

Abstract. We show how theorem proving and methods for handling real algebraic constraints can be combined for hybrid system verification. In particular, we highlight the interaction of deductive and algebraic reasoning that is used for handling the joint discrete and continuous behaviour of hybrid systems. We illustrate proof tasks that occur when verifying scenarios with cooperative traffic agents. From the experience with these examples, we analyse proof strategies for dealing with the practical challenges for integrated algebraic and deductive verification of hybrid systems, and we propose an iterative background closure strategy.

Keywords: modular prover combination, analytic tableaux, verification of hybrid systems, dynamic logic

1 Introduction

Safety-critical systems occurring in traffic scenarios [9, 27] often are hybrid systems [17, 11], i.e., they combine discrete and continuous behaviour. Discrete behaviour typically originates from a digital controller, which regulates driving and switches to various modes in order to react to changes in the traffic situation. Continuous behaviour is more inherent in the physical process dynamics and results from continuous changes of quantities such as positions over time. Models to describe interacting dynamics use differential equations for continuous evolution and use discrete jumps for discrete state changes [17, 24, 20].

Most verification tools for hybrid systems such as HyTech [3], CheckMate [25], or PHAVer [14], follow the model checking paradigm [7] and work by successive computation of images under hybrid transitions [23].

Because of intricacies of complex continuous dynamics, numerical issues during computations, and general limits of numerical approximation [23], hybrid system model checkers are still much more successful in falsification than in verification. In this work, we are primarily interested in verifying hybrid systems rather than finding bugs. Consequently, we favour a fully symbolic technique, and we follow a deductive approach. Further, unlike model checking, deductive techniques support free parameters [11, 20], which occur in our applications.

We have introduced a family of logics for deductive verification of hybrid systems [20, 22, 21]. We have introduced [20] a dynamic logic $\mathsf{d}\mathcal{L}$ and a calculus

* This research was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center (SFB/TR 14 AVACS, see www.avacs.org).

for verifying hybrid systems. We have also presented [22] an extension with nominals to investigate compositionality. Moreover, we have introduced [21] a temporal extension for verifying correct behaviour at intermediate states.

While the theoretical background and technical details of our approach can be found in [20, 22, 21], here we discuss the practical aspects of combining deduction and algebraic constraints techniques. In particular, we highlight the principles how both techniques interact for verifying hybrid systems. For this, we analyse the degrees of freedom in implementing our calculus in terms of the nondeterminisms of our proof procedure. We illustrate the impact that various choices of proof strategies have on the overall performance. For hybrid system verification, we observe that the nondeterminisms in the interaction between deductive and real algebraic reasoning have considerable impact on the practical feasibility. In this paper, we analyse and explain the causes and consequences of this effect and propose a proof strategy that avoids these complexity pitfalls.

In this paper, we study the modular combination in the $\mathsf{d}\mathcal{L}$ calculus. Our observations are of more general interest, though, and we conjecture that similar results hold for other tableaux prover combinations of logics with interpreted function symbols that are handled using background decision procedures for computationally expensive theories including real arithmetic, approximations of natural arithmetic, or arrays.

Related Work. There are a selected number of logics dedicated to hybrid systems [24, 11, 28]. They focus on other aspects like topological aspects [11] or parallel composition [24], and they do not provide calculi with a constructive integration of arithmetic reasoning that can be used easily for practical verification. Our calculus, however, can be used for verifying actual operational hybrid system models [20, 21], which is of considerable practical interest [9, 27, 17, 11].

A few other approaches [19, 1] use deduction for verifying hybrid systems and actually integrate arithmetic reasoning in STeP [19] or in PVS [1], respectively. Their working principle is, however, quite different from ours. Given a hybrid automaton [17] and a global system invariant, they compile, in a single step, a verification condition expressing that the invariant is preserved under all transitions of the hybrid automaton. Hence, the hybrid aspects and transition structure vanish completely before the deduction even start. All that remains is a quantified mathematical formula. In contrast, our dynamic logic works by symbolic decomposition and preserves the transition structure during the proof, which simplifies traceability of results considerably. The structure in this symbolic decomposition can be exploited for deriving invariants or parametric constraints [20, 21]. Consequently, in $\mathsf{d}\mathcal{L}$, invariants do not necessarily need to be given beforehand.

Several other approaches combine deductive and arithmetic reasoning, e.g. [6, 2]. Their focus, however, is on general mathematical reasoning in classes of

higher-order logic and is not tailored to verify hybrid systems. Our work, instead, is intended to make practical verification of hybrid systems possible. For a discussion of work related to the logic $\mathsf{d}\mathcal{L}$ itself, we refer to [20].

Structure of this Paper. In Section 2, we summarise the syntax, semantics, and calculus of the differential logic $\mathsf{d}\mathcal{L}$, that we introduced [20]. In Section 3, we report on the kind of applications that we are interested in, and we illustrate typical proof tasks. In Section 4, we analyse the principles how the $\mathsf{d}\mathcal{L}$ calculus combines deductive with algebraic reasoning and illustrate the consequences of various proof strategies in our applications from Section 3. Finally, we draw conclusions and discuss future work in Section 5.

2 Differential Logic

In this section, we briefly recapitulate the differential logic $\mathsf{d}\mathcal{L}$ that we have introduced [20] and point out the characteristic traits of $\mathsf{d}\mathcal{L}$. We only develop the theory as far as necessary and refer to [21, 20, 22] for more background. The logic $\mathsf{d}\mathcal{L}$ is a dynamic logic [16] with programs extended to hybrid programs [20].

The principle of dynamic logic [16] is to combine system operations and correctness statements about system states within a single specification language. By permitting system operations α as actions of a labelled multi-modal logic, dynamic logic provides formulas of the form $[\alpha]\phi$ and $\langle\alpha\rangle\phi$, where $[\alpha]\phi$ expresses that all (terminating) runs of system α lead to states in which condition ϕ holds. Likewise, $\langle\alpha\rangle\phi$ expresses that there is at least one (terminating) run of α after which ϕ holds. In $\mathsf{d}\mathcal{L}$, hybrid programs play the role of α.

Hybrid programs generalise discrete programs to hybrid change. In addition to the operations of discrete while programs, they have continuous evolution along differential equations as a fundamental operation. For example, the evolution of a train with constant braking can be expressed with a system action for the differential equation $\ddot{z} = -b$ with second time-derivative $\ddot{z}$ of z.

2.1 Syntax of Differential Logic

Terms and Formulas. The formulas of $\mathsf{d}\mathcal{L}$ are built over a finite set V of real-valued variables and a signature Σ containing the usual function and predicate symbols for real arithmetic, such as $0, 1, +, \cdot, =, \leq, <, \geq, >$. Observe that there is no need to distinguish between discrete and continuous variables in $\mathsf{d}\mathcal{L}$.

The set $\mathrm{Trm}(V)$ of *terms* is defined as in classical first-order logic yielding polynomial expressions. The set $\mathrm{Fml}(V)$ of *formulas* of $\mathsf{d}\mathcal{L}$ is defined as in first-order dynamic logic [16]. That is, they are built using propositional connectives $\wedge, \vee, \rightarrow, \neg$ and quantifiers $\forall, \exists$ (first-order part). In addition, if ϕ is a $\mathsf{d}\mathcal{L}$ formula and α a hybrid program, then $[\alpha]\phi, \langle\alpha\rangle\phi$ are formulas (dynamic part).

Hybrid Programs. In $\mathsf{d}\mathcal{L}$ elementary discrete jumps and continuous evolutions interact using regular control structure to form hybrid programs.

Definition 1 (Hybrid programs). *The set* $\mathrm{HP}(V)$ *of* hybrid programs *is inductively defined as the smallest set such that*

- *If* $x \in V$ *and* $\theta \in \mathrm{Trm}(V)$*, then* $(x := \theta) \in \mathrm{HP}(V)$*.*
- *If* $x \in V$*,* $\theta \in \mathrm{Trm}(V)$*, then* $(\dot{x} = \theta) \in \mathrm{HP}(V)$*.*
- *If* $\chi \in \mathrm{Fml}(V)$ *is a quantifier-free first-order formula, then* $(?\chi) \in \mathrm{HP}(V)$*.*
- *If* $\alpha, \beta \in \mathrm{HP}(V)$ *then* $(\alpha; \beta) \in \mathrm{HP}(V)$*.*
- *If* $\alpha, \beta \in \mathrm{HP}(V)$ *then* $(\alpha \cup \beta) \in \mathrm{HP}(V)$*.*
- *If* $\alpha \in \mathrm{HP}(V)$ *then* $(\alpha^*) \in \mathrm{HP}(V)$*.*

The effect of $x := \theta$ is a discrete jump in state space by an instantaneous assignment. That of $\dot{x} = \theta$ is an ongoing continuous evolution controlled by the differential equation $\dot{x} = \theta$. Systems of differential equations, higher-order derivatives, and evolution invariant regions [20] are defined accordingly.

The test action $?\chi$ is used to define conditions. Its semantics is that of a no-op if χ is true in the current state, and that of a failure divergence blocking all further evolution, otherwise. The non-deterministic choice $\alpha \cup \beta$, sequential composition $\alpha; \beta$ and non-deterministic repetition α^* of hybrid programs are as usual. They can be combined with $?\chi$ to form other control structures, see [16].

2.2 Semantics of Differential Logic

The interpretations of $\mathsf{d}\mathcal{L}$ consist of states assigning real values to state variables, which progress along a sequence of states. A potential behaviour of a hybrid system corresponds to a sequence of states that contain the observable values of system variables during its hybrid evolution. The semantics of a hybrid program α is captured by the state transitions that are possible by running α.

A state is a map $\nu : V \to \mathbb{R}$; the set of all states is denoted by $\mathrm{Sta}(V)$. Further, we use $\nu[x \mapsto d]$ to denote the *modification* of a state ν that is identical to ν except for the interpretation of the symbol x, which is $d \in \mathbb{R}$.

For discrete operations, the semantics, $\rho(\alpha)$, of hybrid program α as a state transition relation in $\mathsf{d}\mathcal{L}$ is as customary in dynamic logic (Def. 3). For continuous evolutions, the transition relation holds for pairs of states that can be interconnected by a continuous system flow respecting the differential equation.

Definition 2 (Valuation of terms and formulas). *For terms and formulas, the* valuation $val(\nu, \cdot)$ *with respect to state* ν *is defined as usual for first-order modal logic (e.g. [16]), i.e., using the following definitions for modal operators*

1. $val(\nu, [\alpha]\phi) = true :\Longleftrightarrow val(\omega, \phi) = true$ *for all* ω *with* $(\nu, \omega) \in \rho(\alpha)$
2. $val(\nu, \langle\alpha\rangle\phi) = true :\Longleftrightarrow val(\omega, \phi) = true$ *for some* ω *with* $(\nu, \omega) \in \rho(\alpha)$

Definition 3 (Semantics of hybrid programs). *The* valuation, $\rho(\alpha)$, of a hybrid program α, *is a* transition relation *on states. It specifies which state ω is reachable from a state ν by operations of the hybrid system α and is defined as*

1. $(\nu,\omega) \in \rho(x := \theta) :\Longleftrightarrow \omega = \nu[x \mapsto val(\nu,\theta)]$
2. $(\nu,\omega) \in \rho(\dot{x} = \theta) :\Longleftrightarrow$ *there is a function* $f : [0,r] \to \mathrm{Sta}(V)$ *with* $r \geq 0$ *such that* $f(0) = \nu, f(r) = \omega$, *and* $val(f(\zeta),x)$ *is continuous in* ζ *on* $[0,r]$ *and has a derivative of value* $val(f(\zeta),\theta)$ *at each time* $\zeta \in (0,r)$. *For* $y \neq x$ *and* $\zeta \in [0,r]$, $val(f(\zeta),y) = val(\nu,y)$. *Systems of differential equations are defined accordingly.*
3. $\rho(?\chi) = \{(\nu,\nu) \ : \ val(\nu,\chi) = true\}$
4. $\rho(\alpha;\beta) = \rho(\alpha) \circ \rho(\beta) = \{(\nu,\omega) : (\nu,z) \in \rho(\alpha), (z,\omega) \in \rho(\beta)$ *for some state* $z\}$
5. $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$
6. $(\nu,\omega) \in \rho(\alpha^*)$ *iff there are* $n \in \mathbb{N}$ *and* $\nu{=}\nu_0, \ldots, \nu_n{=}\omega$ *with* $(\nu_i,\nu_{i+1}) \in \rho(\alpha)$ *for all* $0 \leq i < n$.

2.3 A Calculus for Differential Logic

In this section, we briefly review the $\mathsf{d}\mathcal{L}$ sequent calculus that we introduced [20]. It can be used for verifying hybrid systems in $\mathsf{d}\mathcal{L}$. With the basic idea being to perform a symbolic evaluation, it successively transforms hybrid programs into logical formulas describing their effects.

The $\mathsf{d}\mathcal{L}$ calculus combines deduction and handling of real algebraic constraints modularly. Simply speaking, the purely deductive part of the $\mathsf{d}\mathcal{L}$ calculus handles the discrete part, whereas the continuous part is tackled by real algebraic constraint techniques. On this basis, hybrid system behaviour of interacting discrete-continuous dynamics is handled by a modular calculus combination [20].

The $\mathsf{d}\mathcal{L}$ sequent calculus is summarised in Fig. 1. A *sequent* is of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of formulas. Its semantics is that of the formula $\bigwedge_{\phi\in\Gamma} \phi \ \rightarrow \ \bigvee_{\psi\in\Delta} \psi$. Sequents will be treated as an abbreviation. As usual in sequent calculus—although the direction of entailment is from premisses (above rule bar) to conclusion (below)—the order of reasoning is *goal-directed*: Rules are applied in tableau-style, that is, starting from the desired conclusion at the bottom (goal) to the premisses (sub-goals).

The rule schemata in Fig. 1 can be applied anywhere in the sequent, in particular after adding an arbitrary context Γ, Δ, see [20] for details. Moreover, the symmetric schemata D1–D10 can be applied on either side of the sequent. Finally, in D7 and D8, the schematic modality $\langle\!|\cdot|\!\rangle$ stands for either $[\cdot]$ or $\langle\cdot\rangle$.

For propositional logic, standard rules P1–P9 are listed in Fig. 1. The other rules transform hybrid programs into simpler logical formulas, thereby relating the meaning of programs and formulas. Rules D1–D7 are as in discrete dynamic logic [16, 5]. D8 uses generalised substitutions [5] for handling discrete change. Unlike in uninterpreted first-order logic [13], quantifiers are dealt with using

$$\text{(P1)}\ \frac{\vdash \phi}{\neg\phi \vdash} \qquad \text{(P4)}\ \frac{\phi, \psi \vdash}{\phi \wedge \psi \vdash} \qquad \text{(P7)}\ \frac{\phi \vdash \qquad \psi \vdash}{\phi \vee \psi \vdash}$$

$$\text{(P2)}\ \frac{\phi \vdash}{\vdash \neg\phi} \qquad \text{(P5)}\ \frac{\vdash \phi \qquad \vdash \psi}{\vdash \phi \wedge \psi} \qquad \text{(P8)}\ \frac{\vdash \phi, \psi}{\vdash \phi \vee \psi}$$

$$\text{(P3)}\ \frac{\phi \vdash \psi}{\vdash \phi \to \psi} \qquad \text{(P6)}\ \frac{\vdash \phi \qquad \psi \vdash}{\phi \to \psi \vdash} \qquad \text{(P9)}\ \frac{}{\phi \vdash \phi}$$

$$\text{(D1)}\ \frac{\phi \wedge \psi}{\langle ?\phi \rangle \psi} \qquad \text{(D5)}\ \frac{\phi \vee \langle \alpha; \alpha^* \rangle \phi}{\langle \alpha^* \rangle \phi} \qquad \text{(D9)}\ \frac{\exists t{\geq}0\, \langle x := y_x(t) \rangle \phi}{\langle \dot{x} = \theta \rangle \phi}$$

$$\text{(D2)}\ \frac{\phi \to \psi}{[?\phi]\psi} \qquad \text{(D6)}\ \frac{\phi \wedge [\alpha; \alpha^*] \phi}{[\alpha^*]\phi} \qquad \text{(D10)}\ \frac{\forall t{\geq}0\, [x := y_x(t)] \phi}{[\dot{x} = \theta]\phi}$$

$$\text{(D3)}\ \frac{\langle \alpha \rangle \phi \vee \langle \gamma \rangle \phi}{\langle \alpha \cup \gamma \rangle \phi} \qquad \text{(D7)}\ \frac{[\![\alpha]\!][\![\gamma]\!]\phi}{[\![\alpha; \gamma]\!]\phi} \qquad \text{(D11)}\ \frac{\vdash p \qquad \vdash [\alpha^*](p \to [\alpha]p)}{\vdash [\alpha^*]p}$$

$$\text{(D4)}\ \frac{[\alpha]\phi \wedge [\gamma]\phi}{[\alpha \cup \gamma]\phi} \qquad \text{(D8)}\ \frac{\phi_x^\theta}{[\![x := \theta]\!]\phi}$$

$$\text{(F1)}\ \frac{\mathrm{QE}(\forall x \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma \vdash \Delta, \forall x\, \phi} \qquad \text{(F3)}\ \frac{\mathrm{QE}(\exists x \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma \vdash \Delta, \exists x\, \phi}$$

$$\text{(F2)}\ \frac{\mathrm{QE}(\forall x \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma, \exists x\, \phi \vdash \Delta} \qquad \text{(F4)}\ \frac{\mathrm{QE}(\exists x \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma, \forall x\, \phi \vdash \Delta}$$

Rule D8 is only applicable if the substitution of x by θ in ϕ_x^θ introduces no new bindings. In D9–D10, t is a fresh variable, and, for any v, y_v is the solution of the initial value problem $(\dot{x} = \theta, x(0) = v)$. In F1–F4, x does not occur in Γ, Δ. Further, the $\Gamma_i \vdash \Delta_i$ are obtained from the resulting sub-goals of a side deduction. The side deduction is started from the goal $\Gamma \vdash \Delta, \phi$ at the bottom (or $\Gamma, \phi \vdash \Delta$ for F2 and F4). In the resulting sub-goals $\Gamma_i \vdash \Delta_i$, variable x is assumed to occur in first-order formulas only, as quantifier elimination (QE) is then applicable.

Fig. 1: Rule schemata of the $\mathsf{d}\mathcal{L}$ verification calculus.

quantifier elimination [8] over the reals (QE in F1–F4) in a way that is compatible with dynamic modalities. D9–D10 handle continuous evolution given a first-order definable flow y_x for the differential equation $\dot{x} = \theta$ with symbolic initial value x. D11 is an induction schema with inductive invariant p.

At this point, the full details of how F1–F4 use side deductions to lift quantifier elimination to dynamic logic are not important (they can be found in [20]). What is important to note, however, is that quantifier rules and rules for handling modalities need to interact because the actual constraints on quantified symbols depend on the effect of the hybrid programs within modalities [20]. Thus, at some point, after a number of rule applications that handle the dynamic part, rules F1–F4 will be used to discharge (or at least simplify) a proof obligation over real algebraic or semialgebraic constraints by quantifier elimination [8]. The remaining sub-goals will be analysed further again using dynamic rules. The rules F1–F4 constitute the modular interface that combines deduction for handling dynamic reasoning with algebraic constraint techniques for handling continuous reasoning about $\mathbb{R}$. We discuss the consequences and principles of this combination in Section 4 and analyse proof strategies.

3 Analysis of the European Train Control System

In this section, we report on the applications in safety-critical system verification that we verify in the $\mathsf{d}\mathcal{L}$ calculus, see [21, 20]. We illustrate the typical kinds of proof obligations that occur during our deductive analysis of such hybrid systems. Our experience with verifying these applications forms the basis for our analysis of prover combinations and will be used for illustration in Section 4.

Train Control Applications. In the European Train Control System (ETCS) [9, 20], trains are only allowed to move within their current movement authority block (MA). When their MA is not extended before reaching its end, trains always have to stop within the MA because there can be open gates or other trains beyond. Here, we identify a single component which is most responsible for the hybrid characteristics of safe driving. The speed supervision is responsible for locally controlling the movement of a train such that it always remains within its MA. Depending on the current driving situation, the speed supervision determines a safety envelope s around the train, within which driving is safe, and adjusts its acceleration a in accordance with s (called *correction* in [9]).

We assume that an MA has been granted up to track position m and the train is located at position z, heading with initial speed v towards m. In this situation, $\mathsf{d}\mathcal{L}$ can verify safety properties of speed supervision of the form

$$\psi \rightarrow [(\mathit{corr}; \mathit{drive})^*]z \leq m \tag{1}$$

$$\text{where } \mathit{corr} \equiv (?m - z < s; a := -b) \cup (?m - z \geq s; a := \dots)$$

$$\mathit{drive} \equiv \tau := 0; (\dot{z} = v, \dot{v} = a, \dot{\tau} = 1; ?v \geq 0 \wedge \tau \leq \varepsilon)$$

$$\psi \equiv v^2 \leq 2b(m - z) \wedge b > 0 \wedge \epsilon > 0 \ . \tag{2}$$

It expresses that a train will *always* remain within its MA m, assuming a constraint ψ for the parameters. In *corr*, the train corrects its acceleration or brakes with force b (as a failsafe recovery manoeuvre [9]) on the basis of the remaining distance $(m-z)$. Then, the train continues moving according to *drive*. There, the position z of the train evolves according to the system $\dot{z} = v, \dot{v} = a$ (i.e., $\ddot{z} = a$). The evolution stops when the speed v drops below zero (or earlier). Simultaneously, clock τ measures the duration of the current *drive* phase before the controllers react to situation changes (we model this to bridge the gap of continuous-time models and discrete-time control design). Clock τ is reset to zero when entering *drive*, constantly evolves along $\dot{\tau} = 1$, and is bound by the invariant region $\tau \leq \varepsilon$. The effect is that a *drive* phase is interrupted for reassessing the driving situation after at most ε seconds, and the *corr*; *drive* loop repeats.

Parameter Constraint Discovery. In addition to proof tasks for safety verification, the $\mathsf{d}\mathcal{L}$ approach is also useful for parameter constraint discovery. That is,

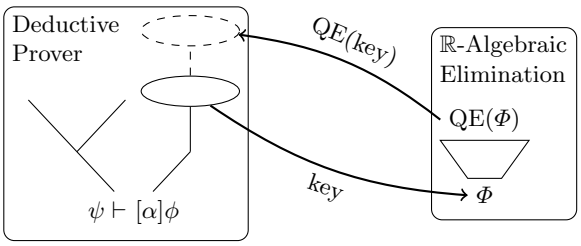


Fig. 2: Prover combination of deduction and algebraic constraint elimination.

instead of starting with a concrete instantiation for ψ in formula (1), the $\mathsf{d}\mathcal{L}$ calculus can be used to identify the required constraints ψ on the free parameters of (1) during the proof. In particular, the constraint ψ in (2) has been discovered by a (semi)automatic discovery process with the $\mathsf{d}\mathcal{L}$ calculus [20].

Finding Inductive Invariants. As a related proof task, the $\mathsf{d}\mathcal{L}$ calculus can be useful for identifying inductive invariants that D11 needs during a proof [20] by analysing partial proofs of individual cases.

4 Combining Deduction and Algebraic Constraints

4.1 Modular Combination of Provers

The principle how the $\mathsf{d}\mathcal{L}$ calculus in Fig. 1 combines deduction technology with methods for handling real algebraic constraints complies with the general background reasoning principles [4, 26, 12]. Unlike in the approaches of Dowek *et al.* [12] and Tinelli [26], the information given to the background prover is not restricted to ground formulas [26] or to atomic formulas as in [12]. From an abstract perspective, the $\mathsf{d}\mathcal{L}$ calculus selects a set Φ of (quantified) formulas from an open branch (Φ is called *key*) and hands it over to the quantifier elimination procedure. The resulting formula obtained by applying QE to Φ is then returned to the main sequent prover as a result, and the main proof continues, see Fig. 2.

In this context, the propositional rules and D-rules (D1–D11) constitute the *foreground rules* in the main prover (left box of Fig. 2) and the arithmetic rules F1–F4 form the set of rules that invoke the *background prover* (right box).

The tableaux procedure [13] for the $\mathsf{d}\mathcal{L}$ calculus is presented in Fig. 3. Observe that the tableaux procedure for our $\mathsf{d}\mathcal{L}$ calculus has a modified set of nondeterministic steps (indicated by $\mathcal{B}$, $\mathcal{M}$, and $\mathcal{F}$, respectively in Fig. 4):

$\mathcal{B}$: *selectBranch*, i.e., which open branch to choose for further rule applications.
$\mathcal{M}$: *selectMode*, i.e., whether to apply foreground $\mathsf{d}\mathcal{L}$ rules (P1–P9 and D1–D11) or background arithmetic rules (F1–F4).
$\mathcal{F}$: *selectFormula*, i.e., which formula(s) to select for rule applications from the current branch in the current mode.

```
while tableaux T has open branches do
  B := selectBranch(T)          (* B–nondeterminism *)
  M := selectMode(B)            (* M–nondeterminism *)
  F := selectFormulas(B,M)      (* F–nondeterminism *)
  if M = foreground then
    B2 := result of applying a D–rule or P–rule to F in B
    replace B by B2 in T
  else
    send key F to background decision procedure QE
    receive result R from QE
    apply a rule F1–F4 to T with QE–result R
  end if
end while
```

Fig. 3: Tableaux procedure for $\mathsf{d}\mathcal{L}$.

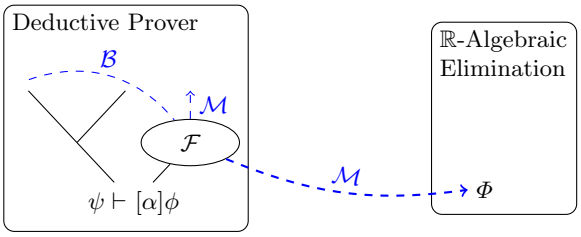


Fig. 4: Nondeterminisms in the tableaux procedure for $\mathsf{d}\mathcal{L}$.

A further, but minor, nondeterminism is whether to expand loops using D6 or to go for an induction by D11. The other $\mathsf{d}\mathcal{L}$ rules do not produce any conflicts once a formula has been selected as they apply to formulas of distinct structures.

At this point, notice that, unlike the classical tableaux procedure [13], we have three rather than four points of nondeterminism, since $\mathsf{d}\mathcal{L}$ does not need closing substitutions. The reason for this is that $\mathsf{d}\mathcal{L}$ has an interpreted domain. Rather than having to try out instantiations that have been determined by unification as in uninterpreted first-order logic [13], we can make use of the structure in the interpreted case of first-order logic over the reals. In particular, arithmetic formulas can be reduced equivalently by QE to simpler formulas in the sense that the quantified symbols no longer occur. As this transformation is an equivalence, there is no loss of information and we do not need to backtrack [13] or simultaneously keep track of multiple local closing instantiations [15].

Despite this, the influence of nondeterminism on the practical prover performance is remarkable. Even though the theory of real arithmetic is decidable by quantifier elimination [8], its complexity is *doubly exponential* in the number of quantifier alternations [10]. While more efficient algorithms exist for linear fragments [18], the practical performance is an issue in nonlinear cases. The computational cost of individual rule applications is quite different from the linear complexity of applying closing substitutions in uninterpreted tableaux.

In principle, exhaustive fair application of background rules by the nondeterminisms $\mathcal{M}$ and $\mathcal{F}$ remains complete for appropriate fragments of $\mathsf{d}\mathcal{L}$. In practice, however, the complexity of real arithmetic quickly makes this naïve approach infeasible. In the remainder of this section, we discuss the consequences of the nondeterminisms and sketch guidelines to overcome the combination problems.

4.2 Nondeterminisms in Branch Selection

In classical uninterpreted tableaux, branch selection has no impact on completeness but can have impact on the proving duration as closing substitutions can sometimes be found much earlier on one branch than on the others. In the interpreted case of $\mathsf{d}\mathcal{L}$, branch selection is even less important. As $\mathsf{d}\mathcal{L}$ has no closing substitutions, there is no direct interference among multiple branches. Branches with (explicitly or implicitly) universally quantified variables have to be closed independently, hence the branch order is not important. For instance, when x is an implicitly universally quantified variable, the branches in the following proof can be handled separately (branches are implicitly combined by conjunction and universal quantifiers distribute over conjunctions):

$$\dfrac{\dfrac{\dfrac{\mathrm{QE}(\forall x\ \dots)}{\Gamma, b>0 \vdash bx^2 \geq 0}\,\mathrm{F1} \qquad \dfrac{\mathrm{QE}(\forall x\ \dots)}{\Gamma, b>0 \vdash bx^4+x^2 \geq 0}\,\mathrm{F1}}{\Gamma, b>0 \vdash (bx^2 \geq 0 \land bx^4+x^2 \geq 0)}\,\mathrm{P5}$$

For existentially quantified variables, the situation is a bit more subtle as multiple branches interfere indirectly in the sense that a simultaneous solution needs to be found for all branches at once. In $\exists v\,(v>0 \land v<0)$, for instance, the two branches resulting from the cases $v>0$ and $v<0$ cannot be handled separately as the existential quantifier claims the existence of a simultaneous solution for $v>0$ and $v<0$, not two different solutions. Thus, when v is an implicitly existentially quantified variable, the branches in the following proof need to synchronise before quantifier elimination is applied:

$$\dfrac{\dfrac{\dfrac{\mathrm{QE}(\exists v\ \dots)}{b>2 \vdash b(v-1)>0}\,\mathrm{D8} \qquad \dfrac{}{b>2 \vdash (v+1)^2+b\epsilon(v+1)>0}\,\mathrm{D8}}{b>2 \vdash [v:=v-1]bv>0 \qquad b>2 \vdash [v:=v+1]v^2+b\epsilon v>0}}{b>2 \vdash ([v:=v-1]bv>0 \land [v:=v+1]v^2+b\epsilon v>0)}$$

The order in which the intermediate steps at two branches are handled has no impact on the proof. Branches like these *synchronise* on an existential variable v in the sense that all occurrences of v need to be first-order for quantifier elimination to work. Consequently, the only fairness assumption for $\mathcal{B}$ is that whenever a formula of a branch is selected that is waiting for synchronisation with another branch to become first-order, then it propagates its rule application to the other

branch. In the above case the left branch synchronises with the right branch on v. Hence, rule F3 can only be applied to $b(v-1) > 0$ on the left branch after D8 has been applied on the right branch to yield first-order occurrences of v.

4.3 Nondeterminisms in Formula Selection

In background proving mode, it turns out that nondeterminism $\mathcal{F}$ is important for the practical performance. When a branch closes or, at least, can be simplified significantly by a quantifier elimination call, then the running time of a single decision procedure call seems to depend strongly on the number of irrelevant formulas that are selected in addition to the relevant ones by $\mathcal{F}$.

Clearly, when Φ is a set of formulas that yields a tautology such that applying F1–F4 closes a branch, then selecting any superset $\Psi \supseteq \Phi$ of Φ from a branch yields the same answer in the end (a sequent forms a disjunction of its formulas hence it can be closed to true when any subset closes). However, the running time until this result will be found in the larger Ψ is strongly disturbed by the presence of complicated additional but irrelevant formulas. From our experience with Mathematica, decision procedures for full real arithmetic seem to be distracted considerably by such irrelevant additional information.

Yet, such additional information accumulates in tableaux procedures quite naturally, because the purpose of a proof branch in $\mathsf{d}\mathcal{L}$ is to keep track of all that is known about a particular (symbolic) case of the system behaviour. Generally, not all of this knowledge finally turns out to be relevant for that case but only plays a role in other branches. Nevertheless, throwing away part of this knowledge light-heartedly would, of course, endanger completeness.

For instance, the safety statement (1) in Section 3 depends on a constraint on the safety envelope s that regulates braking versus acceleration by the condition $m - z \geq s$ in *corr*. A maximal acceleration of a is permitted in case $m - z \geq s$, when adaptively choosing s depending on the current speed v, maximum braking force b, and maximum controller response time ϵ in accordance with the following constraint (which can be discovered by the $\mathsf{d}\mathcal{L}$ calculus [20]):

$$s \;\geq\; \frac{v^2}{2b} + \left(\frac{a}{b}+1\right)\left(\frac{a}{2}\varepsilon^2 + \varepsilon v\right) \quad . \tag{3}$$

This constraint is necessary for some but not for all cases of the safety analysis, though. In the case where the braking behaviour of ETCS is analysed, for instance, the constraint on s is irrelevant, because braking is the safest operation that a train can do to avoid crashing into preceding trains. The unnecessary presence of several quite complicated constraints like, for instance, (3), however, can distract quantifier elimination procedures considerably.

A possible solution for this is to iteratively consider more formulas of the sequent and attempt decision procedure calls. There, only those additional formulas need to be considered that share variables with any of the other selected

formulas. Further, timeouts can be used to discontinue lengthy decision procedure calls and continue along other choices of the nondeterminisms in Fig. 3. For complicated cases with a prohibitive complexity, this heuristic process, which we followed manually, worked well on our examples.

4.4 Nondeterminisms in Mode Selection

In its own right, nondeterminism $\mathcal{M}$ has less impact on the prover performance than $\mathcal{F}$. Every part of a branch could be responsible for closing it. In particular, the foreground closing rule P9 of the main prover can only close branches for comparatively trivial reasons like $b > 0, \epsilon > 0 \vdash \epsilon > 0$. Hence, mode selection has to give a chance to the background procedure every once in a while, following some fair selection strategy. From the observation that some decision procedure calls can run for hours without terminating, we can see, however, that $\mathcal{M}$ needs to be devised with considerable care.

As the reason for closing a branch can be hidden in any part of the sequent, some expensive decision procedure calls can be superfluous if the branch can be closed by continuing $\mathsf{d}\mathcal{L}$ reasoning on the other parts. For instance, if F is some complicated algebraic constraint, decision procedure calls triggered by nondeterminism $\mathcal{M}$ can lead to nontermination within any feasible time for

$$\ldots, \epsilon > 0, m - z \geq s \vdash F, [\mathit{drive}]\epsilon > 0, \ldots$$

Instead, if $\mathcal{M}$ chooses foreground rules, then an analysis of $[\mathit{drive}]\epsilon > 0$ by $\mathsf{d}\mathcal{L}$ rules will quickly discover that the maximum reaction-time ϵ remains constant while driving. Then, this part of the induction step closes without the need to solve constraint F at all. For this reason, proof strategies that eagerly check for closing branches by background procedure calls are not successful in practice.

Unfortunately, converse strategies that strongly favour foreground $\mathsf{d}\mathcal{L}$ rule applications in $\mathcal{M}$, are not appropriate either. There, splitting rules like P5 and P7 can eagerly split the problems onto multiple branches without necessarily making them any easier to solve. If this happens, then slightly different but similar arithmetic problems of about the same complexity need to be solved on multiple branches rather than just one resulting in runtime blow-up.

The reason why this can happen is that there is a syntactic redundancy in the sequent encoding of formulas. For instance, the sets of sequents before and after the following rule application are equivalent:

$$\text{P5}\frac{\psi \vdash v^2 \leq 2b(m-z) \qquad \psi \vdash \epsilon > 0 \qquad \psi \vdash (z \geq 0 \rightarrow v \leq 0)}{\psi \vdash v^2 \leq 2b(m-z) \wedge \epsilon > 0 \wedge (z \geq 0 \rightarrow v \leq 0)}$$

Yet, closing the three sequents above the bar by quantifier elimination is not necessarily easier than the single sequent below (neither conversely). Even worse,

if the sequents close by applying rules to ψ, then similar reasoning has to be repeated for three branches. This threefold reasoning may not be detected as identical when ψ is again split differently on the three resulting branches.

Further, the representational equivalence in sequents is purely syntactic, i.e., up to permutation, the representations share the same disjunctive normal form. In the uninterpreted case, this syntactic redundancy is exploited by the rules P1–P9 in order to transform sequents towards a canonical form with atomic formulas, where partial closing situations are more readily identifiable. In the presence of a background decision procedure, however, reduction to sequents with atomic formulas is no longer necessary as it will be undone when handing the formulas over to the background decision procedure.

Even worse, algebraic constraint handling techniques as in Mathematica can come up with a result that is only a restated version of the input when a selected (open) formula cannot be simplified or closed. For instance, the sequent $z < m \vdash v^2 \leq 2b(m-z)$ "reduces" to $\vdash b \geq v^2/(2m-2z) \lor m \leq z$ without any progress. Such reformulation can easily lead to infinite proof loops when the outcome is split by P8 and again handled by the background procedures.

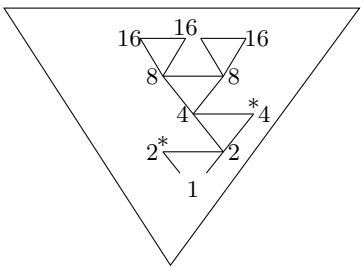
4.5 Iterative Background Closure Strategy

As a strategy to solve the previously addressed issues, we propose the priorities for rule applications in Fig. 5a (with rules at the top taking precedence over rules at the bottom). In this strategy, algebraic constraints are left intact as opposed to being split among multiple branches, because arithmetic rules have a higher priority than propositional rules on first-order constraints. Further, the result of the background procedure is only accepted when the number of variables has decreased to avoid proof loops. Arithmetic background rules have priority 1 *or* 6.

The effect of using priority 1 is that branches are checked eagerly for closing conditions or variable reductions. If reasoning about algebraic constraints does not yield any progress (no variables can be eliminated), then $\mathsf{d}\mathcal{L}$ rules further analyse the system. For this choice, it is important to work with timeouts to prevent lengthy decision procedure calls from blocking $\mathsf{d}\mathcal{L}$ proof progress.

1. *arithmetic rules* F1–F4 if variable eliminated
2. propositional rules P1–P4, P8–P9 on modalities
3. dynamic rules D1–D4, D7–D8
4. dynamic evolution rules D9–D10
5. splitting rules P5–P7 on modalities
6. *arithmetic rules* F1–F4 if variable eliminated
7. propositional rules P1–P9 on first-order formulas

5a: Proof strategy priorities.



5b: Iterative background closure.

This problem is reduced significantly when priority 6 is used for arithmetic rules instead. The effect of priority 6 is that formulas containing modalities are analysed as much as possible before arithmetic reasoning is applied to algebraic constraint formulas. Then, however, the prover can again take too much time analysing the effects of programs on branches which would already close due to simple arithmetic facts like in $\epsilon > 0, \epsilon < 0 \vdash [\alpha]\phi$.

A simple compromise is to use a combination of background rules with priority 1 for quick linear arithmetic [18] and to fall back to expensive quantifier elimination calls for nonlinear arithmetic with priority 6.

As a more sophisticated control strategy on top of the static priorities in Fig. 5a, we propose *iterative background closure* (IBC). There, the idea is to periodically apply arithmetic rules with a timeout T that increases by a factor of 2 after background procedure runs have timed out, see Fig. 5b. Thus, background rules interleave with other rule applications (triangles in Fig. 5b), and the timeout for the sub-goals increases as indicated until the background procedure successfully eliminated variables on a branch (marked by $*$). The effect is that the prover avoids splitting in the average case but is still able to split cases when combined handling turns out to be prohibitively expensive.

5 Conclusions and Future Work

From the experience of using our $\mathsf{d}\mathcal{L}$ calculus [20] for verifying parametric hybrid systems in traffic applications, we have investigated combinations of deductive and algebraic reasoning from a practical perspective. We have analysed the principles of this prover combination, identified the nondeterminisms that remain in the $\mathsf{d}\mathcal{L}$ tableaux procedure, and analysed their impact. We have proposed proof strategies that navigate among these nondeterminisms, including an iterative background closure strategy. Similar to the huge importance of subsumption in resolution, background-style tableaux proving requires quick techniques to rule out branches closing for simple arithmetic reasons. In our preliminary experiments with verifying cooperating traffic agents, our proof strategies significantly reduced the number of interactions and the overall running time significantly.

Future work includes validation of the IBC strategy by experiments in other case studies using a full implementation in our verification tool. Further, we will develop techniques that guide the selection of algebraic constraints by term weight and variable occurrence to discharge simple cases quickly.

Acknowledgements. I thank the anonymous referees for their helpful comments.

References

1. Ábrahám-Mumm, E., Steffen, M., Hannemann, U.: Verification of hybrid systems: Formalization and proof rules in PVS. In: ICECCS, IEEE Computer Society (2001) 48–57

2. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Boulton, R.J., Jackson, P.B., eds.: TPHOLs. Volume 2152 of LNCS., Springer (2001) 27–42
3. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic symbolic verification of embedded systems. IEEE Trans. Software Eng. **22**(3) (1996) 181–201
4. Beckert, B.: Equality and other theories. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999)
5. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions. In Furbach, U., Shankar, N., eds.: IJCAR. Volume 4130 of LNCS., Springer (2006) 266–280
6. Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuta, E., Vasaru, D.: A survey of the Theorema project. In: ISSAC. (1997) 384–391
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, USA (1999)
8. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. **12**(3) (1991) 299–328
9. Damm, W., Hungar, H., Olderog, E.R.: On the verification of cooperating traffic agents. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 3188 of LNCS., Springer (2003) 77–110
10. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. **5**(1/2) (1988) 29–35
11. Davoren, J.M., Nerode, A.: Logics for hybrid systems. Proc. IEEE **88**(7) (Jul 2000) 985–1010
12. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. J. Autom. Reasoning **31**(1) (2003) 33–72
13. Fitting, M.: First-Order Logic and Automated Theorem Proving. Second edn. Springer (1996)
14. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In Morari, M., Thiele, L., eds.: HSCC. Volume 3414 of LNCS., Springer (2005) 258–273
15. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: IJCAR. Volume 2083 of LNCS., Springer (2001) 545–560
16. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. MIT Press (2000)
17. Henzinger, T.A.: The theory of hybrid automata. In: LICS, IEEE Computer (1996) 278–292
18. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. **36**(5) (1993) 450–462
19. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In Henzinger, T.A., Sastry, S., eds.: HSCC. Volume 1386 of LNCS., Springer (1998) 305–318
20. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In Olivetti, N., ed.: TABLEAUX. Volume 4548 of LNCS., Springer (2007) 216–232
21. Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. In Artemov, S., Nerode, A., eds.: LFCS. Volume 4514 of LNCS., Springer (2007) 457–471
22. Platzer, A.: Towards a hybrid dynamic logic for hybrid dynamic systems. In Blackburn, P., Bolander, T., Braüner, T., de Paiva, V., Villadsen, J., eds.: Proc., LICS International Workshop on Hybrid Logic, HyLo 2006, Seattle, USA. Volume 174 of ENTCS. (Jun 2007) 63–77
23. Platzer, A., Clarke, E.M.: The image computation problem in hybrid systems model checking. In Bemporad, A., Bicchi, A., Buttazzo, G., eds.: HSCC. Volume 4416 of LNCS., Springer (2007) 473–486
24. Rönkkö, M., Ravn, A.P., Sere, K.: Hybrid action systems. Theor. Comput. Sci. **290**(1) (2003) 937–973
25. Silva, B.I., Richeson, K., Krogh, B.H., Chutinan, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: ADPM 2000. (2000)
26. Tinelli, C.: Cooperation of background reasoners in theory reasoning by residue sharing. J. Autom. Reasoning **30**(1) (2003) 1–31
27. Tomlin, C., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: a study in multi-agent hybrid systems. IEEE Transactions on Automatic Control **43**(4) (April 1998) 509–521
28. Zhou, C., Ravn, A.P., Hansen, M.R.: An extended duration calculus for hybrid real-time systems. In Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H., eds.: Hybrid Systems. Volume 736 of LNCS., Springer (1992) 36–59

A Sequent Calculus for Integer Arithmetic with Counterexample Generation

Philipp Rümmer

Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University, Sweden
philipp@chalmers.se

Abstract. We introduce a calculus for handling integer arithmetic in first-order logic. The method is tailored to Java program verification and meant to be used both as a supporting procedure and simplifier during interactive verification and as an automated tool for discharging (ground) proof obligations. There are four main components: a complete procedure for linear equations, a complete procedure for linear inequalities, an incomplete procedure for nonlinear (polynomial) equations, and an incomplete procedure for nonlinear inequalities. The calculus is complete for the generation of counterexamples for invalid ground formula in integer arithmetic. All parts described here have been implemented as part of the KeY verification system.

1 Introduction

We introduce a Gentzen-style sequent calculus for integer arithmetic that is tailored to integrated automated and interactive Java software verification. The calculus was developed for dynamic logic for the Java language [1, Chapter 3] (a classical first-order logic) and integrates well-known as well as new algorithms, with the goal to meet the following features:

- Simplification of arithmetic expressions or formulas with the goal to keep everything small and readable. A calculus for this purpose should always terminate and should not cause proof splitting; completeness is a secondary.
- Transparency and the ability to create human-readable proofs and sequences of simplification steps, otherwise it is difficult for a user to resume interactive proving after a number of automated proof steps. The fastest way to understand a proof goal is often to look at the history that led to the goal.
- Handling of nonlinear arithmetic guided by the user, which is necessary for programs that happen to contain multiplication or division operations. The cost of interactive software verification should be justified by the ability to also handle more complex programs than automatic tools.
- Generation of counterexamples for invalid formulas, which is useful during specification and when proving with induction and invariants.
- Handling of the actual modular Java integers, which in our system is modelled by a mapping to the mathematical integers [1, Chapter 12]. Reasoning in this setting requires good support for simplifying expressions, for instance by

(implicitly) proving the absence of overflows. The methods that we developed to this end are beyond the scope of the paper, but are based on the presented calculus.
– Most importantly: it should be easy to use!

Only some of these points can be solved using external procedures and theorem provers (which are, nevertheless, extremely useful for dealing with simpler proof obligations). As a complementary approach, we have developed a novel calculus for integer arithmetic that is directly implemented in our theorem prover KeY [1] in form of derived (i.e., verified) proof rules. The rules are deliberately kept as elementary as possible and are here presented in sequent calculus notation. The calculus is driven by a proof strategy that controls the rule application and realises the following components: (i) a simplification procedure that works on single terms and formulas and is responsible for normalisation of polynomials (Sect. 2), (ii) a complete procedure for systems of linear equations, based on Gaussian elimination and the Euclidian algorithm (Sect. 3), (iii) a complete procedure for systems of linear inequalities, based on Fourier-Motzkin variable elimination (Sect. 4), (iv) an incomplete procedure for nonlinear (polynomial) equations, based on Gröbner bases (Sect. 5), (v) an incomplete procedure for nonlinear inequalities using cross-multiplication of inequalities and systematic case analysis (Sect. 6).

The development of the method was mostly an engineering process with the goal of handling cases that practically occur in Java program verification. It was successful in the sense that many proofs that before only were possible with the help of external provers can now be handled by KeY alone (e.g., the case study [2]), and that many proofs that before were impossible have become feasible.

We do not consider quantifiers or uninterpreted functions in this paper. The calculus is proof confluent (cf. [3]) and can basically be used in two different modes: (i) for simplification, which disables the handling of nonlinear inequalities, prevents case splits and guarantees termination (Procedure 4 in Sect. 5), and (ii) for proving and countermodel construction, which enables all parts (Procedure 5 in Sect. 6).

Introductory example. We start with an example and show how the following statement can be proven within our calculus (in the "full" mode):[1]

$$11a + 7b \doteq 1 \;\vdash\; \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;} \,\rangle\, \mathit{true} \tag{1}$$

In Java dynamic logic, this sequent expresses that the program in angle brackets terminates normally, i.e., in particular does not raise exceptions, given the assumption $11a + 7b \doteq 1$. A proof is conducted by rewriting the program following the symbolic execution paradigm [4], whereby the calculus presented in this

[1] On an Intel Pentium M processor with 1.6 GHz, the KeY implementation of the procedure needs about 460 inference steps and 2 seconds to find this proof.

$$\dfrac{5c \dot{\geq} -7e-8,\ e \dot{\leq} -1,\ c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash}{ce \dot{\geq} -c-e-1,\ e \dot{\leq} -1,\ c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash} \quad (13)$$

$$\dfrac{ce \dot{\geq} -c-e-1,\ e \dot{\leq} -1,\ c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash}{e \dot{\leq} -1,\ c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash} \quad (12)$$

$$\dfrac{\cdots \qquad e \dot{\leq} -1,\ c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash}{\cdots \qquad c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash} \quad (11)$$

$$\dfrac{\cdots \qquad c \dot{\leq} -1,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash}{\ldots,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash} \quad (10)$$

$$\dfrac{\ldots,\ c \dot{\geq} 7e+2,\ 7ce \doteq -2c+1 \vdash}{a \doteq 7e+2,\ b \doteq -11e-3,\ c \dot{\geq} 7e+2 \vdash 7ce+2c-1 \not\doteq 0} \quad (9)$$

$$\dfrac{\cdots \qquad a \doteq 7e+2,\ b \doteq -11e-3,\ c \dot{\geq} 7e+2 \vdash \{b := 7ce+2c-1\}\langle\, \texttt{a=a/b;}\, \rangle\ true}{a \doteq 7e+2,\ b \doteq -11e-3,\ c \dot{\geq} 7e+2 \vdash \{b := 7ce+2c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\, \rangle\ true} \quad (8)$$

$$\dfrac{a \doteq 7e+2,\ b \doteq -11e-3 \vdash \{b := 7ce+2c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\, \rangle\ true}{a \doteq 7e+2,\ b \doteq -11e-3 \vdash \{b := a \cdot c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\, \rangle\ true} \quad (7)$$

$$\dfrac{a \doteq 7e+2,\ b \doteq -11e-3 \vdash \{b := a \cdot c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\, \rangle\ true}{a \doteq 7e+2,\ b \doteq -11e-3,\ d \doteq 3e+1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true} \quad (6)$$

$$\dfrac{a \doteq 7e+2,\ b \doteq -11e-3,\ d \doteq 3e+1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true}{3a \doteq 7d-1,\ b \doteq -2a+d \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true} \quad (5)$$

$$\dfrac{3a \doteq 7d-1,\ b \doteq -2a+d \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true}{7b \doteq -11a+1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true} \quad (4)$$

$$\dfrac{7b \doteq -11a+1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true}{11a+7b \doteq 1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true} \quad (3)$$

$$\dfrac{11a+7b \doteq 1 \vdash \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\, \rangle\ true}{} \quad (2)$$

$$(1)$$

Fig. 1. Proof tree for the introductory example

paper is permanently applied on the *path condition* (in (1), $11a + 7b \doteq 1$) and the *symbolic variable assignment* (in (1), the identity).

The complete proof is shown in Fig. 1. As first step, all formulas are normalised: we choose an arbitrary well-ordering $<_r$ on the variables in the problem ($a <_r b <_r c$) and move big variables to the left and small variables to the right of the relations $\doteq$, $\dot{\leq}$, $\dot{\geq}$, resulting in (2). We then concentrate on the equation in (2), in order to (eventually) turn the leading coefficient 7 into a 1, by means of the extended Euclidian algorithm (cf. [5]). A basis transformation is performed that replaces b with a fresh variable d (such that $a <_r b <_r c <_r d$). One can minimise the coefficient of $11a$ by choosing $b \doteq -2a + d$ and replace the occurrence of b in the original equation with $-2a + d$ (afterwards, the equation is again normalised, sequent (3)). Because the leading coefficient of the first equation is still not 1, a second basis transformation $a \rightarrow 2d + e$ is performed (with $d <_r e$). This turns the leading coefficients of all equations into 1 (sequent (4)).

We can now leave out the equation $d \doteq 3e + 1$, because d does not occur in the sequent anymore. No further inferences are possible in the path condition and the first statement of the program is executed, *updating* the variable assignment accordingly (for simplicity, we assume that no overflows are possible). The assignment $b := 7ce + 2c - 1$ is written in front of the program in (5) and is rewritten and simplified using the equations in (6). The next program statement causes the proof to split on the condition $c \dot{\geq} a$. One branch ($c \dot{<} a$) can immediately be closed because the program contains no further statements. On the other branch (7), we obtain a new assumption $c \dot{\geq} a$ that can be simplified.

The execution of the last assignment yields a new proof obligation (8) in order to prevent division by zero. We prove by contradiction and normalise the

new equation in (9) (and also leave out the first two equations, which are no longer needed for the proof). Because all other possibilities fail in the resulting situation, a case split on the sign of one of the "independent" variables c or e is performed. Here, we will choose c and consider the cases $c \dot{\leq} -1$, $c \doteq 0$, and $c \dot{\geq} 1$. The case $c \doteq 0$ contradicts $7ce \doteq -2c + 1$, and the other two cases can be handled in essentially the same way, so we show only the first one in (10).

By transitivity, from the two inequalities in (10) the inequality $7e + 2 \dot{\leq} -1$ can be derived, which is rounded to $e \dot{\leq} -1$ in (11). No further linear inference steps are possible, but we can at this point deduce properties of product ce by *cross-multiplying* the inequalities $e \dot{\leq} -1$ and $c \dot{\leq} -1$, which yields the new inequality $0 \dot{\leq} (-c - 1) \cdot (-e - 1)$ in (12). After multiplying this inequality with 7, it can in (13) be rewritten using the equation $7ce \doteq -2c + 1$ and turned into $-2c + 1 \dot{\geq} 7 \cdot (-c - e - 1)$.

Now, a contradiction can be derived by reasoning about linear inequalities. From $5c \dot{\geq} -7e - 8$ and $c \dot{\leq} -1$ we derive $7e \dot{\geq} -3$, which is rounded to $e \dot{\geq} 0$ and a contradiction to $e \dot{\leq} -1$.

2 Normalisation of Arithmetic Expressions

Before starting a derivation and permanently during a proof, our calculus normalises (atomic) formulas. This was already demonstrated in the introductory example, and in a proof tree we denote such simplification steps with SIMP. We always fully expand polynomial expressions and represent them as a sum of monomials $\alpha_1 \cdot m_1 + \cdots + \alpha_n \cdot m_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integer literals and $m_1, \ldots, m_n$ are pairwise distinct products of variables (possibly 1 as the empty product, and possibly 0 as the empty sum). Full expansion is in general obviously a bad idea, but we found that it is a reasonable approach in interactive Java program verification that in the vast majority of cases improves the readability of formulas.

Sorting Terms. We put polynomial expressions into a canonical form by ordering the factors in a monomial and the monomials in a polynomial. The ordering $<_r$ that is used in both cases is a strict monomial ordering [6, 7]:

- We assume that a graded monomial ordering $<_r$ [6, 7] on products of variables is given, i.e., a well-ordering (a total, well-founded ordering) with the properties: (i) $\deg m < \deg m'$ implies $m <_r m'$, and (ii) $m <_r m'$ implies $x \cdot m <_r x \cdot m'$ for all variables x. In practice, we define $<_r$ as a graded lexicographic ordering: we assume that a well-ordering $<_r$ on variables[2] is given and then define $c_1 \cdots c_n <_r d_1 \cdots d_k$ if and only if $n < k$ or $n = k$ and $\{\{c_1, \ldots, c_n\}\} <_r \{\{d_1, \ldots, d_k\}\}$ (in the multiset extension of $<_r$, cf. [9]).

[2] In reality, instead of variables we have to deal with arbitrary terms whose head-symbol is not $+$ or $\cdot$, which are compared with a lexicographic path ordering [8].

- We extend $<_r$ by constructing a well-ordering on integer literals: $0 <_r 1 <_r -1 <_r 2 <_r -2 <_r 3 <_r \cdots$.
- We extend $<_r$ on monomials by $\alpha \cdot m <_r \alpha' \cdot m'$ if and only if $m <_r m'$ or $m = m'$ (modulo associativity and commutativity of $\cdot$) and $\alpha <_r \alpha'$.
- We extend $<_r$ on polynomials by $\alpha_1 m_1 + \cdots + \alpha_n m_n <_r \alpha'_1 m'_1 + \cdots + \alpha'_k m'_k$ if and only if $\{\{\alpha_1 m_1, \ldots, \alpha_n m_n\}\} <_r \{\{\alpha'_1 m'_1, \ldots, \alpha'_n m'_n\}\}$ (again using the multiset extension of $<_r$).

For sake of brevity, we will also compare arbitrary terms with $<_r$ and implicitly assume that the terms are first normalised.

Normalisation of Formulas. Atomic formulas are always written in the form $\alpha s * t$ with $* \in \{\dot{\leq}, \doteq, \dot{\geq}\}$, employing equivalences like $s \dot{<} t \Leftrightarrow s + 1 \dot{\leq} t$, and transformed so that the left-hand side αs is the $<_r$-greatest monomial of the polynomial $\alpha s - t$ and $\alpha > 0$. Furthermore, all inequalities are moved to the antecedent, and in case $\alpha s - t$ is a constant polynomial an equation or inequality is directly replaced with *true* or *false*.

We always demand that the coefficients of non-constant terms in an equation or inequality are coprime (do not have non-trivial factors in common), and otherwise divide all coefficients by the greatest common divisor. This also detects that equations like $2y \doteq 1 - 6c$ are unsolvable and equivalent to *false*, and that an inequality like $2y \dot{\leq} 1 - 6c$ can be simplified and rounded to $y \dot{\leq} -3c$ thanks to the discreteness of the integers.

Finally, we add a simple subsumption check for inequalities that eliminates an inequality $s \dot{\leq} t$ from the antecedent in case there is a second inequality $s \dot{\leq} t - \beta$ with $\beta \geq 0$ (correspondingly for $\dot{\geq}$).

3 Equation Handling: Gaussian Variable Elimination

In contrast to many decision procedures or SMT provers, equation and inequality handling for integers are kept separate in our system. The initial reason for this was that we believe that a reduction of equations to inequalities is not an option for interactive proving. Much later we became aware that we also can design more efficient, elegant and practical calculi for linear integer equations than for inequalities, which afterwards justifies the decision. We believe that this is also an important insight when working with the modular Java arithmetic, where the handling of such equations is essential. The sequent calculus described in this section is based on Gaussian elimination and the Euclidian algorithm.[3] It is complete, does not involve proof splitting, and is fast for all problems and benchmarks that we so far have looked at.

[3] The calculus is in parts inspired by [5, Chapter 4.5.2], but in contrast to [5] we perform both row and column operations.

Row Operations. The primary rule of the calculus reduces an expression with the help of an equation in the antecedent. The application of the rule is only allowed if s' is not a subterm of $s \doteq t$ (u is an arbitrary term):[4]

$$\frac{\Gamma, s \doteq t \;\vdash\; \phi[s' + u \cdot (s - t)], \Delta}{\Gamma, s \doteq t \;\vdash\; \phi[s'], \Delta}\ \text{RED} \qquad \text{if } \; s' + u \cdot (s - t) <_r s'$$

Example 1. We show how the rules RED and SIMP are used to solve a system of linear equations (with the ordering $x <_r y$):

$$\dfrac{\dfrac{\dfrac{*}{x \doteq -5,\ y \doteq -1 \;\vdash\; x \doteq -5}}{3y \doteq x + 2,\ y \doteq -1 \;\vdash\; x \doteq -5}\ \text{RED, SIMP}}{3y \doteq x + 2,\ 5y - (3y - x - 2) \doteq x \;\vdash\; x \doteq -5}\ \text{SIMP}}{3y \doteq x + 2,\ 5y \doteq x \;\vdash\; x \doteq -5}\ \text{RED}$$

Column Operations. It is well-known that this kind of reduction alone does not yield a complete calculus for integer equations. An example is the formula $11a + 7b \doteq 1$ in the introductory example, for which no reduction steps are possible. To obtain a complete calculus, we also perform *column operations*—referring to the usual matrix representation of the Gaussian elimination method. Assuming that no more applications of RED are possible in a sequent, and given an equation $\alpha x \doteq s$ of the antecedent, we introduce a fresh unknown x' and perform a basis transformation $x \to u + x'$:

$$\frac{\Gamma, \alpha \cdot (u + x') \doteq s,\ x \doteq u + x' \;\vdash\; \Delta}{\Gamma, \alpha x \doteq s \;\vdash\; \Delta}\ \text{COL-RED}$$

$$\text{if: } x \text{ a variable},\ \alpha > 1,\ (s - \alpha u) = \min_{<_r} \{s - \alpha u' \mid u' \text{ a term}\},$$
$$x' \text{ a fresh variable}, <_r\text{-smaller than all previous symbols}$$

The term u is chosen such that the difference $s - \alpha u$ becomes $<_r$-minimal. One subsequent application of SIMP will thus turn the new equation $\alpha(u + x') \doteq s$ into a formula $\beta y \doteq t$ with $\beta <_r \alpha$. Likewise, βy is $<_r$-smaller than the left-hand sides of other equations $\beta' y = t'$, because RED was applied exhaustively prior to COL-RED. This ensures the overall termination of the procedure (Lem. 1 below) and allows to continue with reduction steps as long as linear equations are present whose left-hand side has a non-unit-coefficient.

We do not apply the rule COL-RED to nonlinear equations, due to the experience that the basis transformations performed by COL-RED cause more harm than good in the nonlinear setting. This is because the usage of a good monomial ordering $<_r$ becomes far more important than in the linear setting (COL-RED effectively alters the ordering by introducing a new smallest variable, possibly in a harmful way). We further discuss this issue in Sect. 5.

[4] In the rule, we write $\phi[s']$ in the succedent to denote that the term s' can occur in an arbitrary position in the sequent, in particular also in the antecedent.

Procedure 1. *Apply* SIMP *with the highest priority,* RED *with second-highest priority, and* COL-RED *with the lowest priority.*

Lemma 1. *Procedure 1 terminates (for sequents containing arbitrary equations and inequalities). For sequents that only contain linear equations, it is complete and proof confluent.*

Example 2. If a proof branch does not get closed by this procedure, the remaining equations are an explicit description of all solutions (counterexamples) of the equations:

$$\frac{\begin{array}{c} x_0 \doteq 125x_3'' - 4,\ x_1 \doteq 25x_3'' - 1,\ x_2 \doteq 20x_3'' - 1,\ x_3 \doteq 16x_3'' - 1, \\ x_0' \doteq 16x_3'',\ x_3' \doteq -3x_3'' \end{array} \vdash}{\begin{array}{c} \vdots \\ x_0 \doteq 5x_1 + 1,\ 4x_1 \doteq 5x_2 + 1,\ 4x_2 \doteq 5x_3 + 1 \vdash \end{array}}$$

The equations that define x_0' and x_3' can be removed afterwards, because these symbols do not occur in the original problem and have no impact on its validity. A concrete counterexample is obtained by assigning arbitrary values to the variables that only occur in the right-hand sides of equations (x_3'').

4 Handling of Linear Inequalities: Fourier-Motzkin Variable Elimination and Case Splits

Although Fourier-Motzkin variable elimination (cf. [10]) generally has a high complexity, it is one of the most popular methods to handle linear inequalities and used in proof assistants like PVS [11], Coq [12] or ACL2 [13, 14]. We found Fourier-Motzkin to be a suitable base method both for linear and nonlinear inequality handling: most reasoning during verification is rather shallow and most inequalities only share symbols with a small number of other inequalities (sparse constraints), which is a situation where Fourier-Motzkin works well. At the same time, the Fourier-Motzkin elimination rule is suited for interactive proving due to its simplicity and the fact that it directly works on integers, in contrast to more efficient linear programming techniques. The full procedure given in this section is complete over the integers, but it involves proof splitting and does usually not terminate for invalid sequents, which means that it cannot (directly) be used as a simplifier for interactive proving. We therefore also identify a subset of the method that does not cause splitting and always terminates, but which is no longer complete (which hardly ever matters in practice). An example for a program that can be verified using the incomplete procedure (together with axioms for division, modulo and Java arithmetic) is shown in Fig. 2.

The Incomplete Procedure. As equations have already been handled in the previous section, we can implement Fourier-Motzkin with a single rule for "cancelling" two inequalities:

$$\frac{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t', \beta t \dot{\leq} \alpha t' \vdash \Delta}{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t' \vdash \Delta}\ \text{FM-ELIM} \qquad \text{if } \alpha > 0,\ \beta > 0$$

The resulting inequality $\beta t \dot{\leq} \alpha t'$ does no longer contain the monomial s and is therefore $<_r$-smaller than both previous inequalities (after a subsequent application of SIMP). To ensure termination, the rule must never be applied twice on a proof branch to the same pair of inequalities.

The performance of Fourier-Motzkin can be improved by adding a rule that turns two inequalities into an equation, based on the law of anti-symmetry:

$$\frac{\Gamma, s \doteq t \vdash \Delta}{\Gamma, s \dot{\leq} t, s \dot{\geq} t \vdash \Delta}\ \text{ANTI-SYMM}$$

Procedure 2. *Apply Procedure 1 (linear equations) with the highest priority, the rule* ANTI-SYMM *with second highest priority and the rule* FM-ELIM *with lowest priority.*

Lemma 2. *The procedure obtained in this way terminates when applied to a sequent containing arbitrary equations and inequalities.*

The Complete Procedure. Fourier-Motzkin is complete for rationals, but incomplete for integers. Our calculus is already more complete than pure Fourier-Motzkin due to the normalisation from Sect. 2 (rounding of inequalities) and the different equation handling of Procedure 1, which are enough to handle many cases that occur in practice (e.g., to show the inconsistency of $4x \dot{\geq} 5 \wedge 4x \dot{\leq} 7$). Making the calculus actually complete has therefore not been of great importance for us. The following approach to this end is rather simplistic, but it has a counterexample generation property that is practically more relevant.

Our calculus becomes complete by performing a systematic case analysis, i.e., by doing proof splitting, in a way similar to Gomory's cutting-planes (cf. [10]). This is realised by the following rule for investigating the borderline case of an inequality:

$$\frac{\Gamma, s \dot{<} t \vdash \Delta \quad \Gamma, s \doteq t \vdash \Delta}{\Gamma, s \dot{\leq} t \vdash \Delta}\ \text{STRENGTHEN}$$

There is a corresponding rule for $\dot{\geq}$. The application of these rules does obviously not terminate in general, but it does for valid sequents (of linear inequalities), provided that a fair application strategy[5] is used and the rule is combined with

[5] In the presence of subsumption checks (Sect. 2), we consider a strategy as fair if STRENGTHEN is eventually applied to each inequality or to any subsuming inequality.

```
/*@
  @ normal_behavior
  @ requires -Decimal.PRECISION < f && f < Decimal.PRECISION
  @          && e + intPart < 32767 && -32768 < e + intPart;
  @ requires -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @ modifiable intPart, decPart;
  @ ensures intPart * Decimal.PRECISION + decPart ==
  @         (\old(intPart) + e) * Decimal.PRECISION + \old(decPart) + f;
  @ ensures -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @*/
public void add(short e, short f) {
  intPart += e;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION ); }
  decPart += f;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION );
  } else {
    short retenue = 0; short signe = 1;
    if ( decPart < 0 ) {
      signe = -1; decPart = (short)( -decPart ); }
    retenue = (short)( decPart / PRECISION );
    decPart = (short)( decPart % PRECISION );
    retenue *= signe; decPart *= signe; intPart += retenue;
  } }
```

Fig. 2. Addition method of class `Decimal` taken from [15], where it was verified using the Loop tool and PVS [11]. This method is part of the JavaCard Purse applet by Gemplus [16]. Using the KeY implementation of our calculus, it takes about 200 seconds and 26000 rule applications to automatically verify that the method adheres to its specification, reasoning about the modular arithmetic of Java.

Procedure 2. For an invalid sequent, a fair strategy eventually produces goals in which all inequalities have been replaced with equations and where Procedure 1 can take over and produce a counterexample.

Case distinctions are also necessary to handle equations in the succedent:

$$\frac{\Gamma \vdash s \dot{\leq} t, \Delta \quad \Gamma \vdash s \dot{\geq} t, \Delta}{\Gamma \vdash s \doteq t, \Delta} \text{ SPLIT-EQ}$$

Procedure 3. *Apply Procedure 2 (the incomplete method) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rule* STRENGTHEN *with lowest priority and in a fair manner.*

Lemma 3. *This procedure is complete and proof confluent, and it eventually produces a counterexample for an invalid sequent.*

Example 3. Consider the following example taken from [17]: Because Procedure 2 is not able to derive a contraction, we apply STRENGTHEN to $x \dot{\leq} 2$ and obtain two cases $x \doteq 1$, $x \doteq 2$ (thanks to ANTI-SYMM), the second of which leads to a counterexample:

$$\dfrac{\dfrac{\dfrac{*}{y \dot{\geq} 1,\ y \dot{\leq} 0,\ x \doteq 1\ \vdash}\ \text{FM-ELIM}}{\vdots} \qquad \dfrac{\dfrac{y \doteq 1,\ x \doteq 2\ \vdash}{y \dot{\geq} 1,\ y \dot{\leq} 1,\ x \doteq 2\ \vdash}\ \text{ANTI-SYMM}}{\vdots}}{\dfrac{4y \dot{\geq} x+1,\ 4y \dot{\leq} x+2,\ x \doteq 1\ \vdash \qquad 4y \dot{\geq} x+1,\ 4y \dot{\leq} x+2,\ x \doteq 2\ \vdash}{\vdots}}$$

$$\dfrac{\vdots}{4y \dot{\geq} x+1,\ 4y \dot{\leq} x+2,\ x \dot{\leq} 2,\ x \dot{\geq} 1\ \vdash}\ \text{STRENGTHEN}$$

5 Handling of Nonlinear Polynomial Equations: Pseudo-Reduction and Gröbner Bases

The validity of equations or inequalities over arbitrary (possibly nonlinear) polynomials over the integers is known to be undecidable [18]. This means that all rules and procedures that we give from now on can never be complete and have been employed or developed with the aim of handling the common cases: when verifying programs, a large amount of the occurring nonlinear proof obligations can and should be taken care of automatically by incomplete calculi. The most important step to this end is to normalise nonlinear expressions (Sect. 2). We describe a comparatively cheap extension—that does not cause any proof splitting and is suited for interactive proving—of Procedure 1 to deal with nonlinear equation.

Pseudo-Reduction. As in Sect. 3, the primary rule for rewriting with (nonlinear) equations is RED. Because we do not apply the rule COL-RED to nonlinear equations, however, there are cases where equations $\alpha s \doteq t$ with $\alpha > 1$ remain in the antecedent that cannot be simplified further. In the sequent $x \dot{\geq} 1,\ y \dot{\geq} 1,\ 2z^2 \doteq y\ \vdash\ xz^2 \dot{\leq} xy$, for instance, none of the rules so far can be applied. In order to handle such cases, we introduce a further reduction rule that is based on pseudo-division and works by first multiplying the target expression with a constant (cf. [5]). The rule must only be applied if $\alpha s \doteq t$ and $u \cdot t \doteq \alpha t'$ are different equations:

$$\dfrac{\Gamma, \alpha s \doteq t\ \vdash\ \phi[u \cdot t \doteq \alpha t'], \Delta}{\Gamma, \alpha s \doteq t\ \vdash\ \phi[s' \doteq t'], \Delta}\ \text{PSEUDO-RED} \qquad \text{if}\quad \deg s > 1,\ \alpha > 1,\ s' = u \cdot s$$

There are similar rules for inequalities $s' \dot{\leq} t'$, $s' \dot{\geq} t'$. We apply PSEUDO-RED only if the left-hand side of the equation $\alpha s \doteq t$ is nonlinear and $\alpha > 1$. Otherwise, the normal reduction rule RED can be used, possibly after turning α into 1 with help of COL-RED.

Gröbner Bases. Rewriting with nonlinear equations using the rules RED and PSEUDO-RED is not confluent and is not able to decide ideal membership in a ring of polynomials. Ideal membership is an approximation of semantic entailment of (nonlinear) equations that we can practically decide: we complete the set of antecedent equations by computing a Gröbner basis [6].

The simplest way to generate a Gröbner basis is to saturate the antecedent with "S-polynomial"-equations by considering all critical pairs of existing integer equations—the Buchberger algorithm [6]. Our calculus produces a non-reduced Gröbner basis over the field of rational numbers that only consists of polynomial equations with integer coefficients, which are easier to compute and almost as useful for reduction as actual Gröbner bases over the integers. Given two equations with overlapping left-hand sides, S-polynomials are added as follows:

$$\frac{\Gamma, s \doteq t, s' \doteq t', s'_r \cdot t \doteq s_r \cdot t' \vdash \Delta}{\Gamma, s \doteq t, s' \doteq t' \vdash \Delta}\ \text{S-POLY} \qquad \begin{array}{l} s = \gcd(s, s') \cdot s_r, \\ s' = \gcd(s, s') \cdot s'_r, \\ 0 < \deg s_r < \deg s, \\ 0 < \deg s'_r < \deg s' \end{array}$$

Similarly to the Fourier-Motzkin elimination rule, this rule must not be applied repeatedly for the same pair of equations to ensure termination. The performance of this naive implementation of Buchberger's algorithm is not comparable with more advanced methods, of course. We have yet to find, however, a verification problem where this would be a problem.

Procedure 4. *Apply Procedure 1 (linear equations) with highest priority, the rule* PSEUDO-RED *with second highest priority, the rule* S-POLY *with third highest priority, and Procedure 2 (linear inequalities) with lowest priority.*

Lemma 4. *Procedure 4 terminates when applied to a sequent containing arbitrary equations and inequalities.*

6 Handling of Nonlinear Polynomial Inequalities: Cross-Multiplication and Case Splits

The handling of nonlinear polynomial inequalities is realised as an extension of the linear inequality handling (Sect. 4). In order to apply linear reasoning to nonlinear arithmetic, we generate linear approximations of products and incrementally strengthen the precision of the approximations through case distinctions. Likewise, case splits are used to ensure the *existence* of linear approximations. Our method has been developed as a heuristic, and we do not have an exact description of the fragment of nonlinear arithmetic that it can handle. The main application areas where the method has proven to be extremely useful are correctness proofs for lemma rules that can be loaded by the prover KeY [1], and the verification of programs with the actual modular integer semantics of Java.

Similarly to the approach in ACL2 [14, 19] (and using their terminology), the primary rule to handle nonlinear inequalities is *cross-multiplication:*

$$\frac{\Gamma, s \dot{\leq} t, s' \dot{\leq} t', 0 \dot{\leq} (t - s) \cdot (t' - s') \vdash \Delta}{\Gamma, s \dot{\leq} t, s' \dot{\leq} t' \vdash \Delta} \text{ CROSS-MULT}$$

There are corresponding rules for $\dot{\geq}$ and for mixed pairs of inequalities. As usual in order to ensure termination, CROSS-MULT must not be applied repeatedly to the same pair of inequalities.

We can give a geometric interpretation of cross-multiplication: for two linear inequalities $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$, cross-multiplication introduces a linear approximation of the product (the bilinear term) xy. In this particular case, the right-hand side of the new inequality $xy \dot{\geq} \beta x + \alpha y - \alpha\beta$ is the greatest plane that bounds the expression xy from below (under the assumptions $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$). More generally, the result of cross-multiplication is a bound on the value of a monomial in terms of $<_r$-smaller monomials. Deriving such bounds is, in practical cases, often sufficient to prove statements in nonlinear arithmetic.

Restricting Cross-Multiplication. An unrestricted application of the rule CROSS-MULT can produce arbitrarily many inequalities and does not terminate. As a heuristic, we only use CROSS-MULT if the product $s \cdot s'$ already occurs as a factor within a left-hand side of an equation or inequality (ignoring the coefficient of $s \cdot s'$). Although this is not strong enough to ensure termination, it guarantees that the total degree of occurring monomials is bounded. We found this heuristic to work reasonably well for most cases (a counterexample is Ex. 5 below).

Case Splits. For two reasons, it is crucial to combine cross-multiplication with case distinctions: (i) nonlinear monomials over the complete set of integers do in general not have linear bounds (observe, for instance, that the term xy is not bounded from above or below by any linear expression in x and y). (ii) case distinctions are in general the only way to strengthen linear bounds (again, consider the term xy under the assumptions $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$, for which no more precise linear lower bound exists than $\beta x + \alpha y - \alpha\beta$).

To account for (i), we introduce a rule that splits over the sign of the value of a term. We apply this rule for variables x that occur in the left-hand side of equations or inequalities:

$$\frac{\Gamma, x \dot{<} 0 \vdash \Delta \quad \Gamma, x \doteq 0 \vdash \Delta \quad \Gamma, x \dot{>} 0 \vdash \Delta}{\Gamma \vdash \Delta} \text{ SIGN-CASES}$$

Ternary splits are motivated by the observation that the case $x \doteq 0$ usually is easy to handle (significantly easier than the original problem), while at the

same time a strict inequality $x \dot{>} 0$ appears to be of much greater use in cross-multiplication than $x \dot{\geq} 0$ (and correspondingly for $x \dot{<} 0$). In our experience, the rule SIGN-CASES outperforms binary cuts.

Point (ii) is accommodated by using the rule STRENGTHEN from Sect. 4, which we apply to linear inequalities in order to incrementally restrict the domain of a variable. For the example above, after strengthening the inequality $x \dot{\leq} \alpha$ to $x \dot{\leq} \alpha - 1$, we can also derive a better bound $\beta x + (\alpha - 1)y - \alpha\beta + \beta$ for the value of xy.

Procedure 5. *Apply Procedure 4 (equations handling and the incomplete procedure for linear inequalities) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rules* CROSS-MULT, SIGN-CASES *and* STRENGTHEN *with the lowest priority and in a fair manner.*

Example 4. We give three further examples that can be proven using Procedure 5 (the last two ones are taken from [14, 19]). In practice, it can often be observed that Procedure 5 is able to solve nonlinear equational problems that cannot be proven using Procedure 4 (only using Gröbner bases).

$$xy \doteq 0 \vdash x \doteq 0,\ y \doteq 0 \qquad x^2 \doteq 2 \vdash \qquad 0 \dot{<} ab,\ 0 \dot{<} cd,\ 0 \dot{<} ac \vdash 0 \dot{<} bd$$

Example 5. A valid sequent that is not provable due to the restriction on the application of CROSS-MULT is $ac \dot{\leq} bd - 1,\ de \dot{\leq} a,\ c \dot{\geq} 1,\ ce \doteq b \vdash$. The problem can be solved by cross-multiplying $de \dot{\leq} a$ and $c \dot{\geq} 1$.

Lemma 5. *When applied to an invalid sequent (containing arbitrary equations and inequalities), Procedure 5 will eventually produce a counterexample.*

7 Related Work

Most similar to our approach is the arithmetic handling in ACL2 [13, 14], which also employs Fourier-Motzkin for linear and cross-multiplication for nonlinear arithmetic. Concerning differences, ACL2 runs arithmetic handling as a purely automated procedure, supports also rationals, does not have separate procedures for equations and does not seem to perform a systematic case analysis.

An method for handling linear equations and inequalities similar to our approach (but lacking counterexample generation) is described in [17] and implemented in the Tecton tool. Related is also [20] about the extension of linear reasoning to nonlinear reasoning.

Higher-order proof assistants usually support integer arithmetic and are so general that arbitrary procedures can be implemented on top of them, often targeting mathematical proofs. In comparison, we tried to develop a simple calculus/procedure specifically for Java verification that works "out of the box"

and requires little expertise. The PVS proof assistant [11] can handle linear integer arithmetic and can simplify nonlinear expressions (involving multiplication and division) to some degree, but does (apparently) not go as far as our approach or ACL2. The Coq system [12] implements an incomplete version of the Omega method for deciding Presburger arithmetic (linear integer arithmetic with quantifiers) that essentially boils down to Fourier-Motzkin. Coq can also simplify ring expressions like polynomials. For HOL light [21], a number of tactics and decision procedures for arithmetic have been implemented, including Cooper's method for deciding Presburger arithmetic, handling of congruences and simplification of polynomial expressions.

Linear arithmetic is one of the most important theories supported by SMT solvers (which generally provide incomparably better performance for linear arithmetic than our implementation based on a general theorem prover framework), see [22] for a list. To the best of our knowledge, no SMT solver offers support for nonlinear arithmetic similar to our approach or ACL2. SMT solvers typically use linear programming techniques like Simplex, combined with methods like branch-and-bound or Gomory's cutting planes to realise completeness on the integers.

8 Conclusions and Future Work

We have presented the main components of a proof procedure for linear and nonlinear integer arithmetic, represented as sequent calculus rules together with application strategies. The procedure is completely implemented, and the soundness of the implementation is verified in the prover KeY itself. In addition to the calculus shown here, KeY also supports division and modulo operations and provides further methods like polynomial division. Based on this, we have formalised the Java semantics of integer operations.

For the future, we are considering a more efficient stand-alone implementation of the calculus, possibly based on the DPLL(T) framework. As a more conceptual extension, we plan to combine the calculus with free-variable reasoning for handling quantifiers. The general approach for this is described in [23], but needs to be investigated more carefully. Finally, we would like to add support for bit-wise operations (as they can be found in Java).

Acknowledgements. I want to thank Wolfgang Ahrendt and Richard Bubel for many inspiring discussions and comments on this paper. Thanks are also due to the anonymous referees for helpful comments.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)

2. Mostowski, W.: Fully verified JavaCard API reference implementation. In: 4th International Verification Workshop. (2007) To appear.
3. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
4. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
5. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison-Wesley (1997) Third edition.
6. Buchberger, B.: An algorithmical criterion for the solvability of algebraic systems. Aequationes Mathematicae **4** (1970) 374–383 (German).
7. Buchberger, B.: A critical-pair/completion algorithm for finitely generated ideals in rings. In: Proceedings of the Symposium "Rekursive Kombinatorik" on Logic and Machines: Decision Problems and Complexity, London, UK, Springer-Verlag (1984) 137–161
8. Dershowitz, N.: Termination of rewriting. J. Symb. Comput. **3** (1987) 69–116
9. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM **22** (1979) 465–476
10. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
11. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Proceedings, CAV. Volume 1102 of LNCS., Springer (1996) 411–414
12. Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., Werner, B.: The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France (1993) Version 5.8.
13. Kaufmann, M., Moore, J.S.: ACL2: An industrial strength version of nqthm. In: Compass'96: Eleventh Annual Conference on Computer Assurance, Gaithersburg, Maryland, National Institute of Standards and Technology (1996)
14. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Linear and nonlinear arithmetic in ACL2. In Geist, D., Tronci, E., eds.: CHARME. Volume 2860 of Lecture Notes in Computer Science., Springer (2003) 319–333
15. Breunesse, C.B., Jacobs, B., van den Berg, J.: Specifying and verifying a decimal representation in java for smart cards. In: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology, London, UK, Springer-Verlag (2002) 304–318
16. : Gemplus purse applet. (`http://www.gemplus.com/smart/r_d/publications/case-study/`)
17. Kapur, D., Nie, X.: Reasoning about numbers in tecton. In: International Symposium on Methodologies for Intelligent Systems, Charlotte, North Carolina. (1994)
18. Matijasevic, Y.: Enumerable sets are diophantine (Russian). Dokl. Akad. Nauk SSSR **191** (1970) 279–282 Translation in Soviet Math Doklady, Vol 11, 1970.
19. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Integrating nonlinear arithmetic into into ACL2. In: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004). (2004)
20. Kapur, D., Cyrluk, D.: Reasoning about nonlinear inequality constraints: a multi-level approach. In: Proceedings of a workshop on Image understanding workshop, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1989) 904–915
21. Harrison, J.: The HOL light manual (1.1) (2000)
22. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2006)
23. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In: International Conference on Tests And Proofs (TAP). LNCS, Springer (2007) To appear.

A Proofs (-Sketches)

Proof. (Lem. 1) Termination: the termination of SIMP and RED is immediate. We call the left-hand sides x of equations $x \doteq s$ (x a variable) in the antecedent

"defined variables," and all other variables "independent variables." When applying RED exhaustively, each defined variable will eventually occur in exactly one place in the sequent (namely, in the defining equation).

For proving termination when COL-RED is added, we show that the leading coefficients $\alpha > 1$ of equations $\alpha x \doteq s$ constantly get smaller. We introduce a well-founded ordering on the set of multisets over $\mathbb{N} \cup \{\infty\}$ by lexicographic comparison: for $a_1 \leq \cdots \leq a_n$, $b_1 \leq \cdots \leq b_m$, we define:

$$\{\{a_1, \ldots, a_n\}\} <_m \{\{b_1, \ldots, b_m\}\} \quad \text{iff}$$
$$n < m \text{ or } (n = m \text{ and } (a_1, \ldots, a_n) <_{\text{lex}} (b_1, \ldots, b_m))$$

For a sequent and an independent variable x, we then consider the divisors $\gcd(\alpha_1, \ldots, \alpha_n) \in \mathbb{N} \cup \{\infty\}$, where $\alpha_1, \ldots, \alpha_n$ are all coefficients of equations $\alpha_i x \doteq s_i$ in the antecedent (we define $\gcd() = \infty$). The multiset of such gcds for all independent variables gets $<_m$-smaller for each application of COL-RED, and it gets $<_m$-smaller or stays the same when RED is applied (each time potentially followed by an application of SIMP). This proves termination.

Completeness and proof confluence: assume that no further rules can be applied, but the proof branch at hand is not closed. This implies that the coefficient of the left-hand side of all equations is 1 (otherwise, SIMP or COL-RED can be applied), and that no left-hand side term occurs in two places in the sequent (otherwise, RED can be applied). Due to the fact that 0 is the only polynomial whose value is constantly 0 (and correspondingly for tuples of polynomials), there is a countermodel for the equations in the succedent (a valuation of the independent variables). We extend this valuation on the defined variables according to the equations in the antecedent. When investigating RED and COL-RED, it can be seen that this countermodel also is a countermodel of the original sequent.

Proof. (Lem. 2) To see that the application of FM-ELIM terminates, consider the multiset of pairs of inequalities in the antecedent to which FM-ELIM can but has not yet been applied. Pairs of inequalities can be compared lexicographically using $<_r$, and multisets of pairs can be compared using the multiset extension of this ordering. As the multiset gets smaller in this well-founded ordering each time FM-ELIM is applied, termination is guaranteed.

The rule ANTI-SYMM can introduce new equations. Such a new equation is either trivially true and is eliminated, or it is a contradiction and the proof branch is closed, or it reduces the number of independent variables by one. In the last case, Fourier-Motzkin basically has to start over once Procedure 1 has done its job, but this can only happen a finite number of times.

Inferring Invariants by Symbolic Execution

Peter H. Schmitt and Benjamin Weiß

University of Karlsruhe
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
{pschmitt,bweiss}@ira.uka.de

Abstract. In this paper we propose a method for inferring invariants for loops in JAVA programs. An example of a simple while loop is used throughout the paper to explain our approach. The method is based on a combination of symbolic execution and computing fixed points via predicate abstraction. It reuses the axiomatisation of the JAVA semantics of the KeY system. The method has been implemented within the KeY system which allows to infer invariants and perform verification within the same environment. We present in detail the results of a non-trivial example.

1 Introduction

A notorious difficulty in the formal verification of programs is the treatment of loop constructs. An array of techniques has been developed to address this problem. Among these techniques the use of loop invariants is particularly attractive since it does not compromise on the rigour of verification and runs completely automatically, once the correct invariants are provided. In this paper we will try to answer the question how to find invariants.

Several techniques for automatically inferring invariants of a program exist. One general approach is *dynamic analysis*, i.e., analysing the program by executing it with concrete input values. A tool implementing dynamic invariant inference is Daikon [10]: to infer invariants for a program, Daikon first instruments it with state saving code at interesting program points. The instrumented program is then run through a user-specified test suite. Finally, the resulting data base of program states is analysed for properties which held in all of the test runs. These properties are somewhat likely to be invariants, but this is not guaranteed, because the test suite in general cannot cover all cases.

Stronger guarantees can be provided by *static analysis*, i.e., analysing the program by examining its source code without actually executing it on concrete inputs. A common paradigm in static analysis, which is also used in program verification, is *symbolic execution* [16]: the analysed program is "executed", but with symbolic instead of concrete values for the program variables. Static invariant inference techniques are usually based on *abstract interpretation* [7]. Abstract interpretation can be understood as an approximative ("abstract") symbolic execution of the program, which deals with loops through fixed-point iteration.

Termination of this fixed-point iteration is ensured by the approximative nature of the used symbolic execution. An example of a tool which uses abstract interpretation for invariant inference is the static verifier Boogie [17].

Predicate abstraction [12] is a special variant of abstract interpretation, which has been used for invariant inference [11]. Here, the symbolic execution is itself not approximative, but as precise as possible (which corresponds to computing strongest postconditions). Instead, the necessary approximation is performed by explicit "abstraction" operations, which make use of an arbitrary, finite set of predicates over the variables of the program. The inferred invariants are constructed from these predicates. Thus, the problem of finding invariants is reduced to the simpler problem of guessing potentially useful predicates, which can be done heuristically or, when necessary, manually by the user.

In the invariant inference method described in [11], the semantics of the programming language is implicitly incorporated in the algorithms of the system. In our approach we start from the axiomatic semantics for JAVA CARD developed within the KeY project. JAVA CARD [15] is roughly a subset of JAVA which contains all object-oriented features but lacks concurrency, floating-point arithmetic, and dynamic class loading. The KeY system [1, 3] is a deductive verification system for JAVA CARD programs. It is based on an axiomatisation of the JAVA CARD semantics within a program logic calculus, which follows the symbolic execution paradigm. The axiomatisation covers 100% of the JAVA CARD language specification and a bit more with great precision. It has so far mainly been used for program verification, e.g., the Demoney case study, an electronic purse application provided by Trusted Logic S.A. [3, Chapt. 14], the verification of a JAVA CARD implementation of the Schorr-Waite graph marking algorithm [3, Chapt. 15], and the verification of a part of a flight management system from Thales Avionics [14]. The most recent targets of verification with KeY have been an implementation of the Mondex banking card case study [23, 22] and an implementation of the JAVA CARD API [19]. The KeY symbolic execution rules for JAVA CARD have lately also been used for model-based test generation [2, 9]. In this paper we explore the possibility to use the KeY calculus and prover for inferring invariants, by incorporating fixed-point iteration and predicate abstraction. Besides the obvious benefit of reusing an existing formal semantics, this also has the advantage that the generation of loop invariants is an integrated part of the verification effort.

The organisation of this paper is as follows: in Sect. 2, we review our program logic for JAVA CARD and its axiomatisation of the programming language semantics. In Sect. 3, we introduce a simple example for a program and its invariants. Using this example, we then explain our approach for inferring invariants in Sect. 4. The implementation of the approach within the KeY system is sketched in Sect. 5, and the results of initial experiments with the implementation are documented in Sect. 6. Finally, we conclude in Sect. 7.

2 Background

Our approach is based on the program logic JAVA CARD DL [3, Chapt. 3], which is a version of dynamic logic [13]. It extends first-order predicate logic by modal operators $[\mathtt{p}]$ ("box") and $\langle \mathtt{p} \rangle$ ("diamond") for every legal sequence of JAVA CARD statements `p`: the formula $[\mathtt{p}]\psi$ states that if execution of `p` terminates in a state s, then ψ holds in state s; the formula $\langle \mathtt{p} \rangle \psi$ additionally requires that `p` does indeed terminate. In the following we only make use of the box modality $[\mathtt{p}]$. The reason is that in this paper, we are interested in invariants, and invariants are not concerned with the issue of termination: they are safety properties, whereas termination is a liveness property.

Formulas of the form $\varphi \rightarrow [\mathtt{p}]\psi$ are similar to Hoare triples $\{\varphi\}\mathtt{p}\{\psi\}$: they express that if `p` terminates after being started in a state which satisfies φ, then the resulting state satisfies ψ. For example, the meaning of the formula $\mathtt{o.f} \doteq 27 \rightarrow [\mathtt{o.f{+}{+};}]\mathtt{o.f} \doteq 28$ is: if the current value of the field `f` of the object pointed to by the program variable `o` is 27, then after executing the statement `o.f++;` the value of the field has changed to 28. This formula is *valid*, i.e., it holds in all possible states.

Proofs of the validity of JAVA CARD DL formulas can be performed by means of a sequent calculus. A sequent is a construct $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. Its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$, and in the following we do not strictly distinguish between sequents and their equivalent formulas. An example for a sequent calculus rule is andRight:

$$\frac{\Gamma \vdash \varphi,\ \Delta \qquad \Gamma \vdash \varphi',\ \Delta}{\Gamma \vdash \varphi \wedge \varphi',\ \Delta}\ \mathsf{andRight}$$

The (schematic) sequents $\Gamma \vdash \varphi, \Delta$ and $\Gamma \vdash \varphi', \Delta$ are the *premisses* of the rule, and the sequent $\Gamma \vdash \varphi \wedge \varphi', \Delta$ is the *conclusion* of the rule. A rule is *sound* if validity of its premisses implies validity of its conclusion. A proof for a sequent is constructed by applying rules from bottom to top; if all leaves of the resulting tree are valid, then the root sequent must be valid as well. The particular rule andRight deals with a conjunction on the right side of the sequent arrow by splitting the proof tree into two branches.

Formulas with modal operators are handled by rules which perform a symbolic execution of the JAVA CARD program within the modality. These rules operate on the *active statement* of the program, i.e., the first basic statement following a non-active prefix of opening braces, beginnings of `try` blocks and the like. This prefix is denoted by π, and the rest of the program behind the active statement by ω. For example, the active statement of the following program is `i = 0;`:

$$\underbrace{\texttt{\{ try \{}}_{\pi}\ \texttt{i = 0;}\ \underbrace{\texttt{i++; \} catch(Exception e) \{ i = 27; \} \}}}_{\omega}$$

The JAVA CARD DL calculus, as it is implemented in the KeY system, currently contains approximately 1700 rules, of which about 1300 formalise the semantics of JAVA CARD. As we cannot describe all of them here, we restrict our presentation to representative rules for the three basic programming constructs in imperative languages: assignments, conditional statements, and loops. We begin with assignments, which can be symbolically executed with

$$\frac{\Gamma',\ \mathtt{x} \doteq \mathtt{e}' \ \vdash\ [\pi\ \omega]\psi,\ \Delta'}{\Gamma\ \vdash\ [\pi\ \mathtt{x = e;}\ \omega]\psi,\ \Delta}\ \mathsf{assignment}$$

where the expression `e` must not have side effects, and where Γ', $\mathtt{e}'$ and Δ' result from Γ, `e` and Δ, respectively, by substituting a fresh program variable x' for `x`. This rule replaces the assumptions about the initial state of the program by their strongest postcondition under the assignment statement. For example, it transforms the sequent $\mathtt{i} \doteq 27 \vdash [\mathtt{i=i+1;}]\mathtt{i} \doteq 28$ into $i' \doteq 27, \mathtt{i} \doteq i'+1 \vdash \mathtt{i} \doteq 28$.

As formulated here, the assignment rule only works for assignments to local program variables. Assignments to fields or array slots are more complex because of *aliasing*, i.e., the phenomenon that the same memory location may be referred to by different names. They can nevertheless be handled along the same lines as assignments to local variables, but this leads to somewhat complicated formulas containing case distinctions for the possible aliasing situations. For example, symbolically executing $\mathtt{o1.f} \doteq 27 \vdash [\mathtt{o2.f} = 0;]\mathtt{o1.f} \doteq 27$ in this way yields $\mathtt{o1}.f' \doteq 27, \forall x.(x\,\mathtt{.f} \doteq \mathit{if}(x \doteq \mathtt{o2})\mathit{then}(0)\mathit{else}(x\,.\,f')) \vdash \mathtt{o1.f} \doteq 27$. The KeY system normally avoids these complications as far as possible by treating assignments in a different way, which is based on a concept called *updates* [21]. However, for the purpose of inferring invariants, the classical way to handle assignments fits our needs better. The details of how this works out for complex assignments are given in [24]. In the following we restrict ourselves to the simple case of assignments to local variables, which amply suffices to explain our method.

Conditional statements can be handled with this rule:

$$\frac{\Gamma\ \vdash\ (\mathtt{e} \doteq \mathtt{true} \rightarrow [\pi\ \mathtt{p}\ \omega]\psi) \wedge (\neg\mathtt{e} \doteq \mathtt{true} \rightarrow [\pi\ \mathtt{q}\ \omega]\psi),\ \Delta}{\Gamma\ \vdash\ [\pi\ \mathtt{if(e)\ p\ else\ q}\ \omega]\psi,\ \Delta}\ \mathsf{ifElse}$$

Again, the occurring JAVA CARD expression must not have side effects. This restriction is never severe, because a program can always be transformed such that an expression is separated from its side effects; the JAVA CARD DL calculus contains rules which perform such transformations. The ifElse rule symbolically executes a conditional statement by creating two conjuncts which describe the case that the guard expression is true and the case that it is false, respectively. Typically, the next step is to split the proof tree by applying the andRight rule.

Loops can be symbolically executed with the loopUnwind rule

$$\frac{\Gamma\ \vdash\ [\pi\ \mathtt{if(e)\{p\ while(e)\ p\}}\ \omega]\psi,\ \Delta}{\Gamma\ \vdash\ [\pi\ \mathtt{while(e)\ p}\ \omega]\psi,\ \Delta}\ \mathsf{loopUnwind}$$

where, as usual, the expression `e` must not have side effects. This is a simplified version of the actual rule which is sound only if the loop body does not contain `break` or `continue` statements. It transforms the loop into a conditional statement: if the guard expression is satisfied, then the loop body is executed once before getting to the loop again; otherwise, the loop is not entered. Since its premiss again contains the loop, the unwind rule on its own only works for loops which terminate after a statically known and sufficiently small number of iterations. In the general case, loops cannot be handled by symbolic execution alone. Instead, an *invariant rule* can be used, i.e., a rule which makes use of a loop invariant. This loop invariant normally has to be provided manually from the outside.

3 Running Example

As a simple example for a program and its invariants, we will use the following piece of JAVA CARD code, which computes the maximal positive element of an integer array `a`:

```
max = 0;
i = 0;
while(i < a.length) {
  if(a[i] > max) max = a[i];
  i++;
}
```

The program is visualised as a control flow graph in Fig. 1. The nodes of this graph represent the basic commands and guard expressions of the program, and the edges stand for flow of control between the nodes. Control enters the program through the node marked `entry`, and leaves it through the node marked `exit` (for the sake of readability, we ignore here that the program terminates abruptly if $\mathtt{a} \doteq \mathtt{null}$ holds). The control flow graph is annotated with exemplary invariants at interesting program points. Note that in this paper we are talking about invariants in the classical sense, i.e., first-order formulas which always hold when control flow reaches a specific program point such as a loop entry. The notion of "class" or "object" invariants for object-oriented programs [18] is related but lives at a different level of abstraction.

Our example invariants for the program are as follows: at the `entry` node, we assume nothing about the program state, so the invariant here is *true*. After the first assignment statement, $\mathtt{max} \doteq 0$ always holds–this is the strongest postcondition of *true* under the assignment statement. Next is the loop invariant: every time control flow reaches the loop entry, $\forall x.(0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \leq \mathtt{max})$ is satisfied. Unlike the loop invariant, the remaining invariants can again easily be derived from their predecessors as strongest postconditions. In particular, the

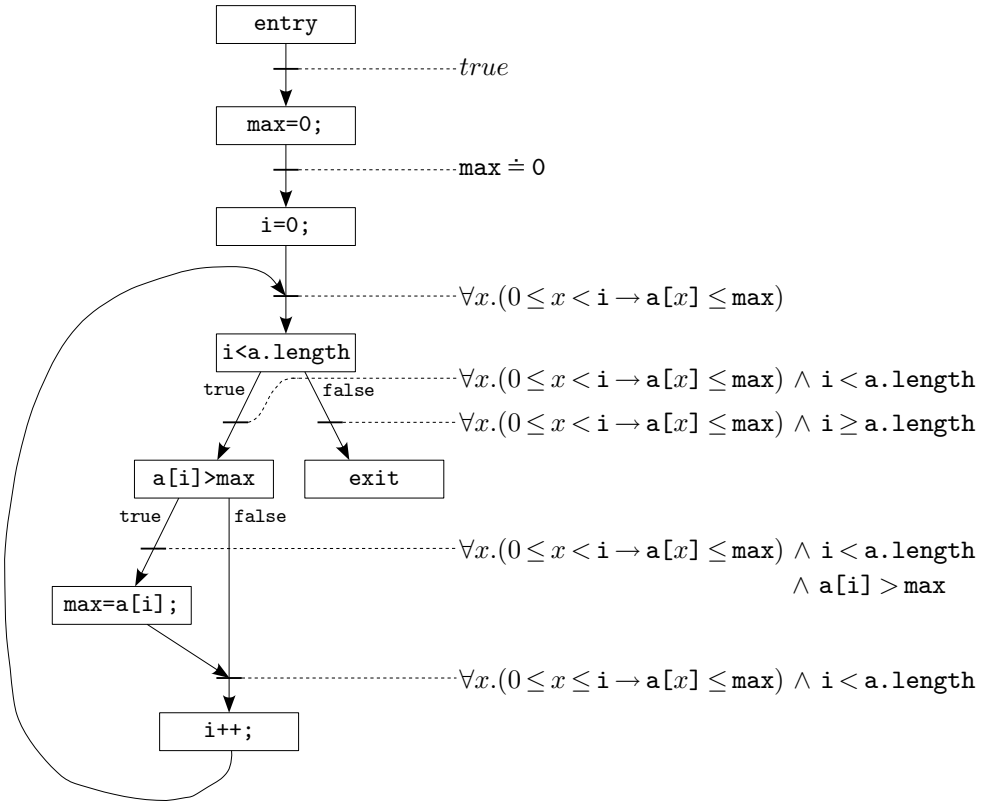


Fig. 1. Control flow graph for the example program, annotated with invariants.

invariant attached to the `exit` node, i.e., the postcondition of the program as a whole, immediately follows from the loop invariant and the negation of the loop guard expression.

4 Our Approach

Our approach is embedded in the overall process of program verification which we imagine has reached a state where a typical goal $\varphi \to [\mathtt{p}]\psi$ has to be proved. The basic idea is to extend the usual symbolic execution of `p` in the JAVA CARD DL calculus by fixed-point iteration and predicate abstraction, and thereby turn the proving process into a form of abstract interpretation. To help the reader understand how this works out exactly, we outline our approach with the particular instantiations of φ, `p` and ψ given in Fig. 2(1): the goal is to prove that after executing the program introduced in Sect. 3, all elements of the array are less than or equal to `max`. Symbolic execution of the program begins with applying the assignment rule to the first two assignments, which leads to the new proof obligation shown in Fig. 2(2). The effect of the two assignments manifests itself

in the two additional assumptions $\texttt{max} \doteq 0$ and $\texttt{i} \doteq 0$. Now, the active statement of the program is the while loop.

The guiding principle of our construction is to always view the formula on the left hand side of the implication as a candidate for an invariant at the program point reached before the active statement of the program occurring in the modality on the right hand side of the implication. According to this principle, the formula φ_1 in Fig. 2(2) is a first candidate for the loop invariant.

The next step in the symbolic execution has to deal with two cases: the loop is not entered and the loop is unfolded at least once. Technically, we apply the loopUnwind rule followed by the ifElse rule, and obtain the conjunction of the two implications shown in Fig. 2(3).

We refrain from splitting the proof by applying the andRight rule, and for the moment concentrate on the first conjunct. The active statement of its box modality on the right hand side is a conditional statement. Thus, the next symbolic execution step is to apply the ifElse rule, which produces two conjuncts in place of one. Again, we do not split this conjunction with the andRight rule. Instead, the assignment `max = a[i];` in the body of the conditional statement is executed, yielding the proof goal shown in Fig. 2(4). Notice that for the first time the assignment rule necessitates the introduction of a new program variable max' to hold the previous value of `max`.

There is a fundamental difference between the first two conjuncts in Fig. 2(4) and the third: the first two refer to the same point in program execution. More precisely, the right hand sides of the first two implications coincide, so we can apply the merge rule

$$\frac{\Gamma \;\vdash\; (\varphi \vee \varphi') \rightarrow \psi,\; \Delta}{\Gamma \;\vdash\; (\varphi \rightarrow \psi) \wedge (\varphi' \rightarrow \psi),\; \Delta}\;\text{merge}$$

which replaces the first two implications by one, logically equivalent, implication, Fig. 3(5). This one implication describes the combined effects of the two execution paths, just like in the control flow graph (Fig. 1) there is only one node for the assignment `i++;`, even though it can be reached via several paths. Making such merging steps possible is the reason why we did not and will not apply the andRight rule to split the proof.

After symbolic execution of `i++;`, the last statement of the loop body, we reach in Fig. 3(6) the same program point again that we had already considered in Fig. 2(2), namely the loop entry. Now, we have two invariant candidates φ_1 from (2) and φ_2 from (6) for the same program point. Naturally, we consider their disjunction $\varphi_1 \vee \varphi_2$ as our new invariant candidate, which is shown in Fig. 3(7). Technically this is achieved by applying the loopMerge rule

$$\frac{\Gamma \;\vdash\; (\varphi \vee \varphi') \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi,\; \Delta}{\Gamma \vdash (\varphi \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi) \wedge (\varphi' \wedge \neg \texttt{e} \doteq \texttt{true} \rightarrow [\pi\,\omega]\psi), \Delta}\;\text{loopMerge}$$

$$
\begin{array}{l}
\mathit{true} \rightarrow [\{\ \texttt{max = 0;} \\
\qquad \texttt{i = 0;} \\
\qquad \texttt{while(i < a.length) \{} \\
\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\quad \texttt{i++;} \\
\qquad \texttt{\}} \\
\quad \}] \underbrace{\forall x.(0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{max})}_{\psi_0}
\end{array}
\tag{1}
$$

$$
\begin{array}{l}
\underbrace{\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0}_{\varphi_1} \rightarrow [\{\ \texttt{while(i < a.length) \{} \\
\qquad\qquad\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\quad \texttt{i++;} \\
\qquad\qquad\qquad \texttt{\}} \\
\qquad\qquad\quad \}]\psi_0
\end{array}
\tag{2}
$$

$$
\begin{array}{l}
\big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \rightarrow [\{\ \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{i++;} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{while(i < a.length) \{} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{i++;} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{\}} \\
\qquad\qquad\qquad\qquad\qquad\quad \}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array}
\tag{3}
$$

$$
\begin{array}{l}
\big(\mathit{max}' \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} > \mathit{max}' \wedge \texttt{max} \doteq \texttt{a[i]} \\
\rightarrow [\{\ \texttt{i++;} \\
\qquad \texttt{while(i < a.length) \{} \\
\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\quad \texttt{i++;} \\
\qquad \texttt{\}} \\
\quad \}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} \leq \texttt{max} \\
\quad \rightarrow [\{\ \texttt{i++;} \\
\qquad\quad \texttt{while(i < a.length) \{} \\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad \texttt{i++;} \\
\qquad\quad \texttt{\}} \\
\qquad \}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array}
\tag{4}
$$

Fig. 2. Invariant inference for the example program (first part).

$$
\begin{array}{l}
\big((max' \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} > max' \wedge \texttt{max} \doteq \texttt{a[i]} \\
\quad \vee \texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} \leq \texttt{max}) \\
\rightarrow [\{\ \texttt{i++;} \\
\qquad \texttt{while(i < a.length) \{} \\
\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\quad \texttt{i++;} \\
\qquad \texttt{\}} \\
\quad \}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array}
\tag{5}
$$

$$
\begin{array}{l}
\left.\begin{array}{l}\big((max' \doteq 0 \wedge i' \doteq 0 \wedge i' < \texttt{a.length} \wedge \texttt{a[}i'\texttt{]} > max' \wedge \texttt{max} \doteq \texttt{a[}i'\texttt{]} \\ \quad \vee \texttt{max} \doteq 0 \wedge i' \doteq 0 \wedge i' < \texttt{a.length} \wedge \texttt{a[}i'\texttt{]} \leq \texttt{max}) \\ \wedge\ \texttt{i} \doteq i' + 1\end{array}\right\}\varphi_2 \\
\rightarrow [\{\ \texttt{while(i < a.length) \{} \\
\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\quad \texttt{i++;} \\
\qquad \texttt{\}} \\
\quad \}]\psi_0\big) \\
\wedge\ \big(\underbrace{\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0}_{\varphi_1} \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array}
\tag{6}
$$

$$
\begin{array}{l}
\varphi_1 \vee \varphi_2 \rightarrow [\{\ \texttt{while(i < a.length) \{} \\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad \texttt{i++;} \\
\qquad\quad \texttt{\}} \\
\qquad \}]\psi_0
\end{array}
\tag{7}
$$

$$
\begin{array}{l}
0 \leq \texttt{i} \wedge \texttt{i} \leq 1 \wedge \forall x.(0 \leq x < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max}) \\
\rightarrow [\{\ \texttt{while(i < a.length) \{} \\
\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\quad \texttt{i++;} \\
\qquad \texttt{\}} \\
\quad \}]\psi_0
\end{array}
\tag{8}
$$

$$
0 \leq \texttt{i} \wedge \forall x.(0 \leq x < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max}) \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0 \tag{9}
$$

Fig. 3. Invariant inference for the example program (second part).

where e must not have side effects. The same principle guides both the merge and the loopMerge rule: the effects of several execution paths are combined in one implication.

If φ_1 was logically equivalent to $\varphi_1 \vee \varphi_2$, we could stop here and declare φ_1 to be our prime candidate for the loop invariant: it would be a fixed point of our iterative inference process. But as you can easily see, this is not the case in our example. So, we have to go on, unfold the loop body once more, obtain a loop invariant candidate φ_3 after the second iteration, check whether $\varphi_1 \vee \varphi_2$ is logically equivalent to $\varphi_1 \vee \varphi_2 \vee \varphi_3$, and then stop or go on accordingly. The problem with this plan of action is that it might (and, in the example, would) not terminate. This is where predicate abstraction as it is, e.g., described in [12], comes into play. To apply this method we first need to fix a set P of predicates. For the example, we choose

$$P = \{\underbrace{\mathtt{i} \doteq 0}_{p_1},\ \underbrace{0 \leq \mathtt{i}}_{p_2},\ \underbrace{\mathtt{i} \leq 1}_{p_3},\ \underbrace{\forall x.(0 \leq x < \mathtt{i} \rightarrow \mathtt{a[}x\mathtt{]} \leq \mathtt{max})}_{p_4}\} \ .$$

In general P might be chosen by following heuristics, e.g., include all parts of the invariant candidate accumulated before the first unfolding of the loop, the loop guard, and parts of the postcondition ψ. In addition one might include in P all the *usual suspect* invariants, as is, e.g., done in [10]. As a final resort P could be customised by user interaction and trial and error. Once P is agreed upon we continue in the above example by replacing $\varphi_1 \vee \varphi_2$ with its abstraction, which is the conjunction of all predicates $p \in P$ for which $(\varphi_1 \vee \varphi_2) \rightarrow p$ is a tautology. Formally, we apply the abstraction rule

$$\frac{\Gamma \ \vdash \ \varphi' \rightarrow \psi,\ \Delta}{\Gamma \ \vdash \ \varphi \rightarrow \psi,\ \Delta}\ \mathsf{abstraction}$$

where $\varphi' = \bigwedge\{p \in P \mid \varphi \rightarrow p \ \text{ is valid}\}$. In our example we get $\varphi' = p_2 \wedge p_3 \wedge p_4$, see Fig. 3(8).

After symbolically executing the loop body for the second time and again applying loopMerge and abstraction, we arrive at $p_2 \wedge p_4$. Since P is of finite size, this process of eliminating predicates must eventually terminate. In the example, it does so after just one more iteration: if we symbolically execute the loop body a third time, the new invariant candidate reached after applying loopMerge and abstraction is again $p_2 \wedge p_4$. We have reached a *fixed point* and stop iterating the while loop. Technically speaking, instead of applying the loopUnwind rule we apply

$$\frac{\Gamma \ \vdash \ \varphi \wedge \neg \mathsf{e} \doteq \mathtt{true} \rightarrow [\pi\, \omega]\psi,\ \Delta}{\Gamma \ \vdash \ \varphi \rightarrow [\pi\ \mathtt{while(e)p}\ \omega]\psi,\ \Delta}\ \mathsf{loopEnd}$$

where e must not have side effects. In our running example this rule application results in Fig. 3(9).

There is one problem with the loopEnd rule: Unlike the other rules introduced in this section, it is not sound, as you can easily see. Applying it is sound if the formula φ is an invariant for the loop. In situations like the one in the example, we have good reason to believe that this is the case: φ is a fixed point of accumulated symbolic executions of the loop body, so it should hold at the loop entry in all possible concrete executions. This is however not quite guaranteed; for example, non-symbolic-execution rules might have been applied in between, disrupting the inference process. Formally prohibiting the application of the loopEnd rule in such situations is conceivable, but complicated. In our context, this is not a necessity: Our implementation prevents unsound applications in a heuristic manner, and if in very rare cases these heuristics should fail and an inferred "invariant" not really be an invariant, this error would be caught when trying to use the false invariant for a regular proof with the invariant rule.

What is a good loop invariant? After all the logical constant *true* is always an invariant. In our scenario where invariant inference is just one part of an overall program verification effort the answer is easy: a loop invariant is good if it allows to successfully complete the overall proof goal. This is the case in our example, since Fig. 3(9) can easily be seen to be universally valid.

In summary, our approach is as follows. The analysed program is symbolically executed with the program rules of the JAVA CARD DL calculus, but without intertwining this process with applications of other rules such as andRight. The symbolic execution is coordinated such that it follows the structure of the control flow graph; in particular, at confluences in the graph, the conjuncts describing the predecessors are combined using the merge and loopMerge rules. Each application of loopMerge is followed by an application of the abstraction rule. After applying abstraction, it is checked whether the resulting abstracted loop invariant candidate is a fixed point, i.e., whether the previous such candidate consisted of the same predicates. If so, it is taken as the inferred loop invariant, and loopEnd is applied. Otherwise, the loop is symbolically executed once again. This process works completely automatically, except that the user may choose to help it find better invariants by specifying predicates for each loop once in the beginning.

5 Implementation

We have implemented our method as an extension of the KeY system. The core element of this implementation are the rules introduced in Sect. 4. The bottleneck of the approach clearly lies in the abstraction rule: it requires checking for each predicate $p \in P$ whether p is implied by the invariant candidate. These checks are implemented as calls to an external automated first-order theorem prover such as Simplify [8]. Decision procedures which support input in the SMT-LIB format [20] can also be used. Of course, such theorem prover calls are neither always successful nor computationally cheap. The lack of completeness is miti-

gated by the fact that the predicates tend to be simple and thus manageable by the theorem prover. Acceptable performance can only be achieved by employing optimisations which reduce the number of necessary calls. Our implementation features several such optimisations. For example, it exploits implication relationships between predicates: if $p_1 \rightarrow p_2$ is known to be valid, and the theorem prover has not been able to prove $\varphi \rightarrow p_2$, then there is no point in checking the validity of $\varphi \rightarrow p_1$. Possibly, performance could be improved further by using an existing predicate abstraction algorithm such as the one from [11] at this place.

Besides the rules themselves, the implementation also comprises a heuristic predicate generator, which automatically creates many of the "usual suspect" invariant elements, such as ordering comparisons between integer program variables, or equality and inequality between variables of a reference type. The predicate generator is complemented by the possibility to manually enter predicates. No quantified formulas are generated automatically, as the number of predicates would get too large to be manageable. However, manually entered predicates are allowed to contain free variables; such predicates are then universally closed by the predicate generator, using various guards. For example, if the user specifies the predicate $\texttt{a[}x\texttt{]} \leq \texttt{max}$, the predicate generator adds $\forall x.(0 \leq x < i \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max})$, together with many other similar predicates.

The final element of the implementation is a proof search strategy, which controls the automatic application of the rules as it is necessary for invariant inference. In particular, the strategy prohibits the application of non-symbolic-execution rules, and it coordinates symbolic execution of several modalities such that it follows the control flow graph: if, e.g., the current proof goal is $(\varphi_1 \rightarrow [\texttt{max = a[i]; i++;}]\psi) \wedge (\varphi_2 \rightarrow [\texttt{i++;}]\psi)$, symbolic execution of the second conjunct is stopped until `max = a[i];` has been symbolically executed in the first conjunct and the merge rule can be applied.

6 Experiments

We tested our implementation on selection sort, a well-known and comparatively simple sorting algorithm with quadratic time complexity. The basic idea of the algorithm is to find the smallest element of the array to be sorted, swap it with the first element, and iteratively repeat this process on the subarray starting at the second position. The exact proof obligation supplied to the KeY system is shown in Fig. 4. It states that, after invoking the program contained in the box modality on an integer array `a` which is not `null` and which has a positive length, the array is sorted. This specification is not strong enough to ensure that the program actually sorts the array; for example, a program could satisfy it by simply setting all array elements to zero. It is however sufficient for our purposes. The program itself is a straightforward JAVA CARD rendering of selection sort. The temporary boolean variables `condOuter` and `condInner` are used

to buffer the values of the loop guard expressions, which is necessary because our implementation of the invariant inference currently requires that the guard expressions must be simple program variables.

```
a ≠̇ null ∧ a.length > 0 → [{ i = 0;
                             condOuter = i < a.length;
                             while(condOuter) {
                               minIndex = i;
                               j = i + 1;
                               condInner = j < a.length;
                               while(condInner) {
                                 if(a[j] < a[minIndex]) minIndex = j;
                                 j++;
                                 condInnder = j < a.length;
                               }
                               temp = a[i];
                               a[i] = a[minIndex];
                               a[minIndex] = temp;
                               i++;
                               condOuter = i < a.length;
                             }
                           }]∀x.(0 < x ∧ x < a.length → a[x − 1] ≤ a[x])
```

Fig. 4. Java Card DL proof obligation for selection sort.

We manually entered the predicates (`condOuter` $\doteq$ `true` $\leftrightarrow$ `i` $<$ `a.length`), (`condInner` $\doteq$ `true` $\leftrightarrow$ `j` $<$ `a.length`), (`a[`x`]` $\leq$ `a[`y`]`), and (`a[minIndex]` $\leq$ `a[`x`]`), where x and y are free variables. The first two of these are necessary only because of the introduction of `condInner` and `condOuter`, and their generation could easily be automated. The other two require more ingenuity, but are still significantly easier to guess than the loop invariants themselves. Together with the automatically generated predicates, this lead to 8794 predicates for the inner loop and 16950 predicates for the outer loop.

Using Simplify as the external first-order theorem prover, the invariant inference process terminated after 3 iterations for the outer loop, containing 4, 4 and 2 iterations for the inner loop, respectively. 652 rules were applied in total. The overall running time on a 1.5 GHz Pentium M machine was about 8.5 minutes. Approximately 65% of this time was spent running Simplify, which was called exactly 800 times. The resulting loop invariants for the inner and the outer loop are shown in Fig. 5 and Fig. 6.

In addition to loop invariants, the invariant rule of the Java Card DL calculus [4] makes use of modifier sets for loops, i.e., information about which memory locations may be modified by a loop. In the case of selection sort, appropriate modifier sets are {`minIndex`, `j`, `condInner`} for the inner loop, and {`minIndex`, `j`, `condInner`, `temp`, `a[*]`, `i`, `condOuter`} for the outer loop. When supplied with these modifier sets (which are quite obvious from the program

$$
\begin{aligned}
&\forall x.\forall y.(0 \le x \land x < y \land y \le i \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ \forall x.(\texttt{i} \le x \land x < \texttt{minIndex} \rightarrow \texttt{a}[\texttt{minIndex}] \le \texttt{a}[x])\\
&\land\ \forall x.(\texttt{i} < x \land x < \texttt{j} \rightarrow \texttt{a}[\texttt{minIndex}] \le \texttt{a}[x])\\
&\land\ \forall x.(\texttt{minIndex} < x \land x < \texttt{j} \rightarrow \texttt{a}[\texttt{minIndex}] \le \texttt{a}[x])\\
&\land\ \texttt{a}[0] \le \texttt{a}[\texttt{i}]\\
&\land\ \texttt{a}[\texttt{minIndex}] \le \texttt{a}[\texttt{i}]\\
&\land\ 0 < \texttt{a.length}\\
&\land\ \texttt{j} \le \texttt{a.length}\\
&\land\ 0 < \texttt{j}\\
&\land\ \texttt{minIndex} < \texttt{a.length}\\
&\land\ 0 \le \texttt{minIndex}\\
&\land\ \texttt{minIndex} < \texttt{j}\\
&\land\ \texttt{i} < \texttt{a.length}\\
&\land\ 0 \le \texttt{i}\\
&\land\ \texttt{i} < \texttt{j}\\
&\land\ \texttt{i} \le \texttt{minIndex}\\
&\land\ \forall x.\forall y.(0 \le x \land x < \texttt{i} \land \texttt{i} \le y \land y < \texttt{a.length} \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ \forall x.\forall y.(0 \le x \land x < \texttt{i} \land \texttt{i} \le y \land y < \texttt{j} \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ \forall x.\forall y.(0 \le x \land x < \texttt{i} \land \texttt{i} \le y \land y < \texttt{minIndex} \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ \texttt{a} \neq \texttt{null}\\
&\land\ \texttt{condOuter} \doteq \texttt{true}\\
&\land\ (\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length})\\
&\land\ (\texttt{condInner} \doteq \texttt{true} \leftrightarrow \texttt{j} < \texttt{a.length})
\end{aligned}
$$

Fig. 5. Inferred invariant for the inner loop of selection sort.

$$
\begin{aligned}
&\forall x.\forall y.(0 \le x \land x < y \land y < \texttt{i} \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ 0 < \texttt{a.length}\\
&\land\ \texttt{i} \le \texttt{a.length}\\
&\land\ 0 \le \texttt{i}\\
&\land\ \forall x.\forall y.(0 \le x \land x < \texttt{i} \land \texttt{i} \le y \land y < \texttt{a.length} \rightarrow \texttt{a}[x] \le \texttt{a}[y])\\
&\land\ (\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length})\\
&\land\ \texttt{a} \neq \texttt{null}
\end{aligned}
$$

Fig. 6. Inferred invariant for the outer loop of selection sort.

code), and, crucially, the loop invariants from Fig. 5 and Fig. 6, the KeY system (in normal mode) was able to automatically prove the validity of the formula from Fig. 4 in about 2 minutes.

7 Conclusions

We have presented a method for inferring invariants for while loops in JAVA programs that can seamlessly be integrated in program verification based on the symbolic execution paradigm. To do so this paradigm had to be adapted in two aspects. First, when using symbolic execution for program verification, intermediate proof goals that involve a case distinction are split into subgoals that are then proved separately. For invariant inference we have to avoid this splitting. Second, the ideas of fixed-point iteration and approximation are not present in the symbolic execution paradigm for program verification. So, we had to introduce them.

The approach has been implemented as an addition to the KeY verification system. The results of first experiments are very encouraging. But, since the success to a great deal depends on the heuristic choice of the set P of abstraction predicates, much more experience is needed to arrive at a dependable evaluation.

Approximation in static analysis typically takes the form of "erring on the safe side", i.e., precision may be lost, but not correctness. In principle, our invariant inference is no exception: the inferred invariants may sometimes not be useful, but they should always indeed be invariants. However, since we introduced a rule which is not strictly sound, this is not guaranteed with the same high degree of confidence that is carried by the axioms from the KeY calculus. Remedying this imperfectness is one direction for future work. Nevertheless, the success rate of suggesting true invariants is already a lot higher than in dynamic analysis methods such as Daikon.

Another line of future work, which is more speculative, concerns the generation of the abstraction predicates. One could investigate the use of CEGAR (counterexample-guided abstraction refinement) techniques [6, 5] to arrive at useful predicates in a less heuristic, more systematic manner.

References

1. B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. The KeY System 1.0 (deduction component). In F. Pfenning, editor, *Proceedings, 21st International Conference on Automated Deduction (CADE)*, LNCS. Springer, 2007. To appear.
2. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007. To appear.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

4. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In K.-K. Lau, editor, *Proceedings, 7th International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *Proceedings, 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *LNCS*, pages 2–18. Springer, 2005.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings, 12th International Conference on Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings, 4th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
9. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007. To appear.
10. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
11. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings, 29th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 191–202. ACM Press, 2002.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings, 9th International Conference on Computer Aided Verification (CAV)*, pages 72–83. Springer, 1997.
13. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
14. J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In M. Rochard-Foy and A. Wellings, editors, *Proceedings, 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM Press, 2006.
15. Java Card platform specification 2.2.1. Sun Microsystems, Inc., October 2003.
16. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
17. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In K. Yi, editor, *Proceedings, 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 119–134. Springer, 2005.
18. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
19. W. Mostowski. Fully verified Java Card API reference implementation. In *Proceedings, 4th International Verification Workshop (VERIFY'07), Workshop at CADE-21, Bremen, Germany*. CEUR Workshop Proceedings, 2007. To appear.
20. S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, University of Iowa, 2006.
21. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
22. P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In M. Hinchey and T. Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Press, 2007. To appear.
23. S. Stepney, D. Cooper, and J. Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.
24. B. Weiß. Inferring invariants by static analysis in KeY. Diplomarbeit, University of Karlsruhe, March 2007.

Author Index