WING 2009

Workshop on Invariant Generation

Working Proceedings

March 29, 2009, University of York, UK

Preface

Whether working at the level of abstract design models or concrete loop-based code, inductive invariants play a pivotal role in specifying and reasoning about software systems. Inductive invariants capture design intuitions, so not surprisingly their generation represents a significant research challenge. In general, no single technique or approach will adequately address this challenge. Progress will most likely require novel integrations of complementary techniques and approaches. By promoting the exchange of ideas and experiences, the WING workshop series aims to support the development of such integrations. Engagement from across a number of fields is therefore a very important ingredient for a successful workshop. So we are delighted that this, the second WING workshop, has attracted a strong grouping of researchers, each bringing a different perspective on the challenge of invariant generation.

This diversity is reflected in the submissions, which include both algorithmic and heuristic approaches to invariant discovery and verification. The use of novel formalisms, new logics and reasoning techniques are also well represented, while a diverse range of applications is good to see.

We are delighted to have two excellent keynote speakers, Leonardo de Moura (Microsoft Research) and Andrey Rybalchenko (Max Planck Institute for Software Systems). Both have significant research expertise and insights into the challenges and opportunities for the growing WING community.

We would like to thank the program committee members and reviewers for all their efforts. Thanks also go to the steering committee members for their valuable advice and guidance.

Finally, we would like to thank Andrei Voronkov for his help on EasyChair, and the ETAPS-09 local organizers for their support and guidance.

Andrew Ireland & Laura Kovács
Program Chairs

Workshop Organization

WING 2009 is organized as a satellite workshop of ETAPS 2009.

Program Chairs

Andrew Ireland (Heriot-Watt University, UK)
Laura Kovács (EPFL, Switzerland)

Program Committee

Nikolaj Bjørner (Microsoft Research, US)
Martin Giese (University of Oslo, Norway)
Reiner Hähnle (Chalmers University of Technology, Sweden)
Paul Jackson (University of Edinburgh, UK)
Jens Knoop (Technical University of Vienna, Austria)
Francesco Logozzo (Microsoft Research, US)
Wolfgang Schreiner (RISC-Linz, Austria)
Helmut Veith (Technical University of Darmstadt, Germany)
Andrei Voronkov (Manchester University, UK)
Thomas Wies (EPFL, Switzerland)

External Reviewers

Richard Bubel (Chalmers University of Technology, Sweden)
Florian Zuleger (Technical University of Darmstadt, Germany)

Table of Contents

Applications and Challenges in Satisfiability Modulo Theories

Leonardo de Moura
Microsoft Research,
One Microsoft Way, Redmond, WA, 98074, USA
leonardo@microsoft.com

Abstract

Satisfiability Modulo Theories (SMT) solvers decide logical satisfiability (or dually, validity) of formulas in classical multi-sorted first-order logic with equality, with respect to a background theory. SMT solvers have proven highly scalable, efficient and suitable for integrating theory reasoning. The success of SMT for program analysis and verification is largely due to the suitability of supported background theories. These theories include: arithmetic, uninterpreted functions, recursive datatypes; and theories of program structures such as machine arithmetic (bit-vectors), arrays, and heaps. SMT solvers enable applications such as extended static checking, automatic invariant generation, predicate abstraction, test case generation, model-based testing, and bounded model checking over infinite domains, to cite a few. In this talk, we give a short overview of the theory and techniques used in SMT. We also present theoretical and practical challenges related to SMT and their applications. Then, we describe how SMT solvers, with emphasis on Microsoft's Z3 solver, are used in program analysis and verification. Several concrete applications from program analysis at Microsoft will be presented.

Improving Scalability of Invariant Generation for Assertion Checking

Andrey Rybalchenko
MPI-SWS, Germany
rybal@mpi-sws.mpg.de

Abstract

Invariant generation holds great potential for proving program assertions. It can automatically discover invariants that imply given assertions and fulfill the inductiveness conditions imposed by the proof rules for assertion checking. Finding such invariants is a difficult task, which requires solving complex constraints that determine the desired invariants. Improving the efficiency of solving techniques is one of the major steps towards achieving practical applicability of invariant generation tools. In this talk, we will review how invariant generation is applied for assertion checking and present a variety of techniques that can dramatically improving its scalability. In particular, we will present a recent method of leveraging dynamic program analysis to statically simplify constraints used for invariant generation.

Automated Invariant Generation for the Verification of Real-Time Systems

Bahareh Badban
Department of Computer and Information Science
University of Konstanz, Germany

Stefan Leue
Department of Computer and Information Science
University of Konstanz, Germany

Jan-Georg Smaus[*]
Institut für Informatik
University of Freiburg, Germany

Abstract

We present an approach to automatically generating invariants for timed automata models. The CIPM algorithm that we propose first computes new invariants for timed automata control locations taking their originally defined invariants as well as the constrains on clock variables imposed by incoming state transitions into account. In doing so the CIPM algorithm also prunes idle transitions, which are transitions that can never be taken, from the automaton. We discsuss a prototype implementation of the CIPM algorithm as well as some initial experimental results.

1 Introduction

Predicate abstraction is an instance of the general theory of abstract interpretation [6]. It is a technique for generating finite abstract models of large or infinite state systems. This technique involves abstracting a concrete transition system using a set of formulas called *predicates*. Predicates usually denote some state properties of the concrete system. The predicate abstraction is conservative in the sense that if a property holds on the abstract system, there will be a concretization of the property that holds on the concrete system as well. Abstraction is defined by the value (true or false) of the predicates in any concrete state of the system [18]. This technique was first introduced by Graf and Saïdi [10] as a method for automatically determining invariant properties of infinite-state systems. As mentioned above, the idea of predicate abstraction is to generate a *finite state* abstraction of the system with respect to a finite set of predicates, which are mostly provided by the users themselves. Such an abstraction will have at most $2^{\|P\|}$ distinct states for a total number $\|P\|$ of predicates [16]. There are two obstacles to the practical use of the predicate abstraction approach:

1. The initial abstraction is *not fine enough*, and hence it is too abstract to be able to verify any property of the concrete model. In such a case the method relies on counter examples or proofs in the overall verification process to refine the abstraction [5], and as it is stated by Lahiri, et.al. in [15]:

   > "It is not clear if it is always preferable to compute the abstraction incrementally. But, we have observed that the refinement loop can often become the main bottleneck in these techniques (e.g. SLAM), and limits the scalability of the overall system."

2. The other obstacle is that the abstraction works with the predicates that are *provided by the user*. Hence, this method relies on the user's understanding of the system and on a trial-and-error process [15]. This phenomenon has been pointed out by Das and Dill in [7]:

[*]The work of this author was partially funded by the German Research Council (DFG) through the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB TR14/AVACS).

"Another problem is how to discover the appropriate set of predicates. In much of the work on predicate abstraction, the predicates were assumed to be given by the user, or they were extracted syntactically from the system description. It is obviously difficult for the user to find the right set of predicates (indeed, it is trial-and-error process involving inspecting failed proofs), and the predicates appearing in the system description are rarely sufficient. There has been less work and less progress, on solving the problem of finding the right set of predicates. In addition [...] there is a challenge of avoiding irrelevant predicates [...]"

The purpose of our work is to provide support for an automated predicate abstraction technique for dense real-time models according to the timed automaton model of [1] by generating a more useful set of predicates than a manual, ad-hoc process would be able to provide. We analyze the behaviour of the system under verification to discover its local state invariants. During this analysis we remove idle transitions which are transitions that can never be traversed. We plan to incorporate the generated invariants into the abstraction phase of a counterexample guided abstraction refinement method for timed automata by using them as the *initial set of predicates* that is used to define an initial abstraction of the concrete model.

Related Work. How to discover predicates for use in the abstraction of real time system models has been widely discussed in the literature. Colón and Uribe [5] introduce an interactive method for predicate abstraction of real-time systems where a set of predicates called *basis* is provided by the user. As mentioned in the paper itself, this way the choice of abstraction basis is based on the user's understanding of the system. Therefore, generation of a suitable abstraction basis relies on trial-and-error. Möller et. al. in [21] introduce a method which is based on identifying a set of predicates that is fine enough to distinguish between any two clock regions and which creates a strongly preserving abstraction of the system. Refinement of the abstraction is accomplished using an analysis of the spuriousness of counter examples. Also in [7] Das and Dill use the spurious trace in discovering predicates for the predicate abstraction. Das et. al. in [8] introduce $Mor\phi^{--}$ which is a prototype for the verification of invariants in predicate abstraction. McMillan et. al. in [14, 19] and Henzinger, et. al. in [11] use interpolation to detect feasibility of the abstract trace and also to extract predicates from the proof for use in the abstraction. McMillan and Amla [20] introduce a proof-based automatic abstraction. The slicing approach of [13, 23] is a recent method with the same intention of reducing the state space of timed automata models. However, it is based on static analysis and not on a semantic interpretation of the automaton structure.

Another direction of research in the field of predicate abstraction addresses symbolic techniques. In this method a decision procedure takes a set of predicates P and symbolically executes a decision procedure on all the subsets over P. This results in a directed graph which represents the answer to a predicate abstraction query. The method aims at reducing the number of decision procedure calls since this number often tends to be extremely large [15]. In the first step, the first-order formula is encoded into some equi-satisfiable Boolean formula, and in the next step it is verified using a SAT solver. Examples of different symbolic methods are in [18, 15, 17, 16].

In [12] Hoffmann et. al. use the technique of predicate abstraction in order to obtain search heuristics to be used in directed model checking of safety properties. In [2] Ball et. al. introduce an abstraction method based on oracle-guided widening. In most cases the widening is such that it simply drops a variable from the rest of the computations of the pre-states.

Structure of the Paper. The paper is organized as follows. Section 2 provides some preliminary definitions on real-time automata. The semantics of timed automata and their possible transitions are discussed in section 2.1. Section 3 introduces our method of creating new invariants. The respective algorithm, called CIPM, is explained in Section 3.1. In Section 3.2 we illustrate the algorithm by an example which is used by Möller et. al. in [21]. Section 4 describes an initial implementation of our approach. Finally, Section 5 presents concluding remarks and discusses future work.

2 Preliminary Definitions

In this section we reiterate the classical definition of timed automata according to [4, 1]. Additional concepts and notations which will be used throughout the paper are also introduced in this section.

A *timed automaton* consists of a finite state automaton together with a finite set of *clocks*. Clocks are non-negative real valued variables which keep track of the *time* elapsed since the last reset operation performed on the respective clock. The finite state automaton describes the system *control* states and its transitions. Initially, all clocks are set to 0. All clocks evolve at the same speed. A *configurations* of the system is given by the current control location of the automaton and the value of each clock, denoted $\langle l, u\rangle$, where l is the control location and u is the valuation function which assigns to each clock its current value. $u+d$, for $d \in \mathbb{R}^{+1}$, is a valuation which assigns to each clock x the value $u(x)+d$, i.e., it increases the value of all clocks by d. $\mathsf{G}(X)$ denotes the set of *clock constraints* g for a set X of clock variables. Each g is of the form

$$g := x \leq t \mid t \leq x \mid \neg g \mid g_1 \wedge g_2$$

where $x \in X$, and t, called *term*, is either a variable in X or a constant in $\mathbb{R}^+$. By $var(g)$ we denote the set of all clock variables appearing in g. A timed automaton is then formally defined as follows:

Definition 1. *A timed automaton $\mathscr{A}$ is a tuple $\langle L, l_0, \Sigma, X, \mathscr{I}, E\rangle$ where*

- *L is a finite set of locations (or states), called* control locations*,*
- *$l_0 \in L$ is the initial location,*
- *Σ is a finite set of labels, called* events*,*
- *X is a finite set of clocks,*
- *$\mathscr{I} : L \mapsto \mathsf{G}(X)$ assigns to each location in L some clock constraint in $\mathsf{G}(X)$,*
- *$E \subset L \times \Sigma \times 2^X \times \mathsf{G}(X) \times L$ represents* discrete *transitions.*

The clock constraint associated with each location $l \in L$ is called its *invariant*, denoted $\mathscr{I}(l)$. We later refer to these invariants as the *original* invariants. It requires that time can pass in a control location only as long as its corresponding invariant remains true. In other words, $\mathscr{I}(l)$ must hold whenever the current state is l.

We call a constraint g *atomic* if there exist two terms s and t such that g is equivalent to $s \leq t$, $t \leq s$ or their negation[2]. With each clock constraint g, we associate a set of its atomic sub-formulas, $\mathsf{atom}(g)$, defined as:

- $\mathsf{atom}(g) := \{g\}$ when g is atomic,
- $\mathsf{atom}(g_1 \wedge g_2) := \mathsf{atom}(g_1) \cup \mathsf{atom}(g_2)$.

[1] $\mathbb{R}^+$ is the set of all non-negative reals, including 0.
[2] The negation of $t \leq s$ is $s < t$.

Reset constraints are conjunctions of (one or more) formulas of the form $x := c$ where c is a constant in $\mathbb{R}^+$. We similarly define the application of atom over reset constraints, e.g. $\mathsf{atom}(x := 3 \wedge y := 0) \equiv \{x := 3,\ y := 0\}$. An atomic constraint is called *bounded* if it is of the form $x \prec c$, where c is a constant value and $\prec \in \{=, <, \leq\}$. g is *unbounded* if it is not bounded. For example, $x \leq y$ and $x > 2$ are unbounded. We define a set of unbounded constraints in a set A of constraints as: $^{\mathsf{un}}(A) := \{a \in A \mid a \text{ is unbounded}\}$.

We say a valuation u *satisfies* g, denoted $u \models g$, if the assigned value to all variables in g by u satisfies g. For example if $g := x+1 > y \wedge x < 5$, and $u(x) = 4.1$ and $u(y) = 3$, then $u \models g$. We consider true as a valid proposition which is satisfied by each valuation, i.e. $u \models \mathsf{true}$ for each u. For a set A, $u \models A$ if $u \models a$ for each $a \in A$. Given two constraints g and g', g entails g', denoted $g \Rightarrow g'$, if for any valuation u, $u \models g'$ if $u \models g$. For instance, $x \geq 3 \Rightarrow x > 2$. Based on this, we define a function join[3] as:

$$\mathsf{join}(g, g') := \begin{cases} g' & \text{if } g \Rightarrow g' \\ g & \text{if } g' \Rightarrow g \\ \mathsf{true} & \text{if } \neg g \Rightarrow g' \text{ or equivalently, } \neg g' \Rightarrow g \\ g \vee g' & \text{otherwise} \end{cases} \tag{1}$$

Intuitively, given any two constraints g and g', $\mathsf{join}(g, g')$ is equivalent to the weaker one. We further extended this definition over sets:

$$\mathsf{join}(A, B) := \bigvee_{(a,b) \in A|B} \mathsf{join}(a, b)$$

where $A|B := \{(a,b) \mid a \in A,\ b \in B \text{ and } var(a) = var(b)\}$. For example, if $A = \{x < y, x > 2\}$ and $B = \{x < 3\}$ then $A|B = \{(x > 2, x < 3)\}$ and hence, $\mathsf{join}(\{x < y, x > 2\}, \{x < 3\}) = \mathsf{join}(x > 2, x < 3) = \mathsf{true}$, since $x \geq 3 \Rightarrow x > 2$.

In a timed automaton the values of all clocks evolve at the same speed. Therefore, if at time t_0, x has the value of x_0 then after Δt time units the value of x will be $x_0 + \Delta t$. This fact leads us to the next property:

Note 1. In a timed automaton $\mathscr{A}$, if at some point of time (e.g. t_0) a relation like $x \prec y$ where $\prec \in \{=, <, \leq\}$ holds between two clock variables x and y then this relation will be preserved until one of the variables is reset. This is because $x(t) = x + \Delta t \prec y + \Delta t = y(t)$, where Δt computes the time elapsed as of t_0.

Lemma 1. *If $u \models g$ for a valuation u and some* unbounded *atomic constraint g, then $u + d \models g$ for any $d \in \mathbb{R}^+$. This can be extended to any set of unbounded atomic constraints, too.*

Proof. If g is of the form $x \geq c$ or $x > c$, then proof is obvious. For other constraints g, by Note 1, $u + d \models g$ (d is the Δt). By definition this property also extends to sets of unbounded atomic constraints. □

Lemma 2. *For each two atomic constraints g and g', and sets A and B of atomic constraints, we have:*

1. $g \Rightarrow \mathsf{join}(g, g')$ *and* $g' \Rightarrow \mathsf{join}(g, g')$.
2. *if $u \models g$ for some valuation u, then $u \models \mathsf{join}(g, g')$.*
3. *if $u \models A$ (or $u \models B$) for some valuation u, then $u \models \mathsf{join}(A, B)$.*

Proof. The proofs of these properties can be sketched as follows:

[3]This operation is called *strong join* in [22].

1. This is obvious according to the definition of join.
2. By the definition of join the statement 2 is equivalent to statement 1.
3. $u \models A$, hence by definition, $u \models a$ for all $a \in A$. Therefore, according to the second item, for all $a \in A$ and $b \in B$, $u \models \mathsf{join}(a,b)$. Hence, by definition, $u \models \mathsf{join}(A,B)$. This holds, analogously, for when $u \models B$.

□

2.1 Semantics

We associate a transition system $\mathscr{S}_{\mathscr{A}}$ with each timed automaton $\mathscr{A}$[4]. States of $\mathscr{S}_{\mathscr{A}}$ are pairs $\langle l,u\rangle$, where $l \in L$ is a control location of $\mathscr{A}$ and u is a valuation over X which satisfies $\mathscr{I}(l)$, i.e. $u \models \mathscr{I}(l)$. $\langle l_0,u\rangle$ is an *initial* state of $\mathscr{S}_{\mathscr{A}}$ if l_0 is the initial location of $\mathscr{A}$ and for all $x \in X$: $u(x) = 0$.

Transitions. For each transition system $\mathscr{S}_{\mathscr{A}}$ the system configuration changes by two kinds of transitions:

- *Delay transitions* allow time $d \in \mathbb{R}^+$ to elapse. The value of all clocks is increased by d leading to the transition $\langle l,u\rangle \overset{d}{\mapsto} \langle l,u+d\rangle$. This transition can take place only when the invariant of location l is satisfied along the transition.
- *Discrete transition* enabling a transition (c.f. Definition 1)[5]. In this case all clocks, except those which are reset, remain unchanged. This results in the transition $\tau := \langle l,u\rangle \overset{a,g,r}{\mapsto} \langle l',u'\rangle$ where a is an event, g is a clock constraint and r is a reset constraint (c.f. Definition 1).

An *execution* of a system is a possibly infinite sequence of configurations $\langle l,u\rangle$ where each pair of two consecutive configurations corresponds to either a discrete or a delay transition.

In the sequel, τ and d denote discrete and delay transitions, respectively. We may denote a discrete transition τ as $\langle l,u\rangle \overset{\tau}{\mapsto} \langle l',u'\rangle$ when a,g,r do not need to be clarified. For each $\tau := \langle l,u\rangle \overset{a,g,r}{\mapsto} \langle l',u'\rangle$, we define $\mathsf{G}_\tau := \mathsf{atom}(g)$ and $\mathsf{R}_\tau := \mathsf{atom}(r)$. For this transition, l and l' are called source of τ, denoted $\mathsf{sorc}(\tau)$, and target of τ, denoted $\mathsf{tar}(\tau)$, respectively. $\mathsf{G}_{\tau/\mathsf{R}_\tau}$ (respectively $\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}$) represents the set of all atomic constraints in G_τ (respectively $\mathscr{I}(\mathsf{sorc}(\tau))$) which do not have a variable occurring in R_τ. For instance, if $\tau := \langle l,u\rangle \overset{x\leq y \wedge z<x+1,\ z:=0}{\mapsto} \langle l',u'\rangle$, then $\mathsf{G}_\tau = \{x \leq y,\ z < x+1\}$, $\mathsf{R}_\tau = \{z := 0\}$ and $\mathsf{G}_{\tau/\mathsf{R}_\tau} = \{x \leq y\}$. In this example, z occurs in R_τ. For this transition if $\mathscr{I}(l) = \{y < 2, z > 4\}$ then $\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau} = \{y < 2\}$.

For each discrete transition τ we define:

$$\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \cup {}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \cup \overline{\mathsf{atom}}(\mathsf{R}_\tau)$$

where $\overline{\mathsf{atom}}(\mathsf{R}_\tau) = \bigcup_{r \in \mathsf{atom}(\mathsf{R}_\tau)} \overline{r}$, and

$$\overline{x := c} = \begin{cases} \{x \leq y \mid y \in X - \{x\}\} & \text{if } c = 0 \\ \{x \geq c\} \cup \{x \leq y + c \mid y \in X - \{x\}\} & \text{if } c > 0 \end{cases}$$

We will later show in Theorem 3 that $\mathsf{inv}(\tau)$ is a set of constraints that are preserved in $\mathsf{tar}(\tau)$. Below, in the next lemma, we prove this for when we have just entered a state and before any time elapses. For the example above, $\tau := \langle l,u\rangle \overset{x\leq y \wedge z<x+1,\ z:=0}{\mapsto} \langle l',u'\rangle$, $\mathsf{inv}(\tau) = {}^{\mathsf{un}}(\{x \leq y\}) \cup {}^{\mathsf{un}}(\{y < 2\}) \cup \overline{z := 0} = \{x \leq y\} \cup \emptyset \cup \{z \leq x, z \leq y\} = \{x \leq y, z \leq x, z \leq y\}$. Here, $X = \{x,y,z\}$.

[4]We work with nondeterministic timed automata.

[5]A transitions is *enabled* if it can be traversed from the source control location.

Lemma 3. *For each discrete transition $\langle l,u\rangle \overset{\tau}{\mapsto} \langle l',u'\rangle$, we have $u' \models \mathsf{inv}(\tau)$.*

Proof. Clock variables are reset iff they occur in R_τ. Hence, those variables which do not occur in R_τ retain their value when the transition τ is occurring. These are variables which belong to $\mathsf{G}_{\tau/\mathsf{R}_\tau}$. Therefore, $u' \models \mathsf{G}_{\tau/\mathsf{R}_\tau}$, and since $\mathsf{G}_{\tau/\mathsf{R}_\tau} \subseteq {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau})$, $u' \models {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau})$. The same reasoning applies to ${}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau})$. Since u' has some new values for the reset variables in r it holds that $u' \models \bar{r}$ right at the time of the occurrence of the transition (notice that no time has elapsed yet). Therefore, $u' \models \overline{\mathsf{atom}}(\mathsf{R}_\tau)$. Summing these up results in $u' \models \mathsf{inv}(\tau)$. □

Definition 2. *For each control location l, we define a set of incoming discrete transitions, ${}^{\mathsf{in}}\mathsf{trans}(l,\mathscr{A})$, and a set of outgoing discrete transitions, ${}^{\mathsf{out}}\mathsf{trans}(l,\mathscr{A})$ as:*

$$\begin{aligned} {}^{\mathsf{in}}\mathsf{trans}(l,\mathscr{A}) &:= \{\tau \mid \exists l_i,u_i,u:\ \langle l_i,u_i\rangle \overset{\tau}{\mapsto} \langle l,u\rangle\} \\ {}^{\mathsf{out}}\mathsf{trans}(l,\mathscr{A}) &:= \{\tau \mid \exists l',u',u:\ \langle l,u\rangle \overset{\tau}{\mapsto} \langle l',u'\rangle\} \end{aligned}$$

Notice that this definition relies exclusively on discrete transitions. Since only a finite number of such transitions exists these two sets are well-defined.

We now define a *reduction system*. This system simplifies the (disjunction of) clock constraints. The intuition is that any valuation function u satisfies the left hand-side of the reduction step (denoted by $\longrightarrow$) if and only if it also satisfies the right-hand side.

Definition 3 (Reduction System)**.** *We apply the following reduction rules on disjunction of constraints, ϕ. Here, s and t are terms.*

1. $s < t \vee s = t \longrightarrow s \le t$

2. $s < t \vee s > t \longrightarrow s \neq t$

ϕ is called *simplified* if none of the rules above are applicable on it. We apply the *reduction rules* of Definition 3 on ϕ as long as they can be applied, which means that the result of the application is not identical to the constraint itself. When the reduction process terminates, we call the result $\mathsf{simp}(\phi)$. Obviously, $\mathsf{simp}(\phi)$ is simplified.

Lemma 4. *If $u \models \phi$ then $u \models \mathsf{simp}(\phi)$, for each valuation u.*

Proof. If $\phi \longrightarrow \psi$ with any of the reduction rules, then obviously $u \models \psi$. Therefore, $u \models \mathsf{simp}(\phi)$ as well. □

In the next section we introduce an automatic approach to creating new invariants and to reducing the size of the model by pruning away those transitions which can never be traversed and which hence have no impact on a reachability analysis of the model.

3 Creating New Invariants

In this section we present the CIPM algorithm which strengthens the given original invariants in each control location by analysing the incoming discrete transitions to that specific control location.

3.1 The Algorithm

The input of Algorithm 1 is a timed automaton. Without loss of generality we assume that each location is assigned a separate index between 0 and $\|\mathscr{A}\|-1$, e.g. l_1, where $\|\mathscr{A}\|$ is the number of control locations in $\mathscr{A}$.

Before explaining the algorithm we introduce the notion of idleness of a transition which expresses that a transition will never be enabled.

Definition 4. *A discrete transition $\tau : \langle l,u\rangle \mapsto \langle l',u'\rangle$ is called* idle *if it can never be enabled.*

Amongst other reasons, a transition can be idle when the constraint over the transition is never being satisfied or the valuation function obtained from the transition does not satisfy the invariant of the target location (i.e., $u' \neg \models \mathscr{I}(l')$). For instance, if τ is the discrete transition $\langle l,u\rangle \overset{x\leq y}{\mapsto} \langle l',u'\rangle$, where $x > y+3$ is invariant in location l, i.e. $\mathscr{I}(l) = \{x > y+3\}$, then this transition is idle since the constraint $x \leq y$ is never fulfilled as long as we are in l.

The CIPM algorithm first collects the set $\mathscr{I}(l_i)$ of all the original invariants in each location l_i. It then selects each location l_i and collects its incoming transitions in $^{\mathsf{in}}\mathsf{trans}(l_i,\mathscr{A})$. The idle transitions within that set are detected using Lemma 5 and are deleted from the model. For each non-idle τ in $^{\mathsf{in}}\mathsf{trans}(l_i,\mathscr{A})$ the algorithm next computes $\mathsf{inv}(\tau)$. It thereby extracts all constraints that are propagated to location l_i when executing transition τ. Applying join to all propagated constraints yields $\mathsf{inv}(l_i)$ which defines the full set of constraints that are imposed on l_i by all of the transitions in $^{\mathsf{in}}\mathsf{trans}(l_i,\mathscr{A})$. Since l_i may also have some original invariant, $\mathscr{I}(l_i)$ is the conjunction of the original invariant and all of the previously computed imposed constraints on l_i. This is expressed in the algorithm by $\mathscr{I}_{\mathscr{A}}(l_i) := \mathscr{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$. Computing $\mathscr{I}_{\mathscr{A}}(l_i)$ may render some of the outgoing transitions of l_i idle. Therefore, the algorithm next checks all outgoing transitions of l_i for idleness again using Lemma 5 (1st item). It then removes all transitions detected as being idle. The set seen stores the traversed transitions. It is used to ensure that all transitions are checked for being idle only once.

In Algorithm 1, we use conjunction of sets, which is defined as: $A \wedge B := \bigwedge_{1\leq i\leq n} a_i \wedge \bigwedge_{1\leq j\leq m} b_j$ for $A = \{a_1,\ldots a_n\}$ and $B = \{b_1,\ldots b_m\}$ [6].

Lemma 5. *A discrete transition τ is idle when either of the conditions below, holds:*

- $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ *is a contradiction,*
- $\mathsf{inv}(\tau) \wedge \mathscr{I}(\mathsf{tar}(\tau))$ *is a contradiction.*

Proof.
- $\mathscr{I}(\mathsf{sorc}(\tau))$ holds as long as the current location is $\mathsf{sorc}(\tau)$. At this location, τ is enabled only when G_τ holds. If this occurs then $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ holds. By assumption, this can never happen.
- By Lemma 3, if $\langle l,u\rangle \overset{\tau}{\mapsto} \langle l_i,u_i\rangle$ is enabled then $u_i \models \mathsf{inv}(\tau)$. By definition, $u_i \models \mathscr{I}(\mathsf{tar}(\tau))$ too. Therefore, $u_i \models \mathsf{inv}(\tau) \wedge \mathscr{I}(\mathsf{tar}(\tau))$. This contradicts the assumption. So, τ is never enabled.

□

We say two timed automata $\mathscr{A}$ and $\mathscr{A}_1$ are *equivalent*, denoted $\mathscr{A} \doteq \mathscr{A}_1$, if they differ only on some idle transitions.

Theorem 1. *The* CIPM *algorithm has the following properties:*

- *it is terminating,*

[6] $A \wedge \emptyset$ and $\emptyset \wedge A$ are equivalent to A.

Algorithm 1 Creating Invariants and Pruning the Model (CIPM)

```
REQUIRES: a timed automaton $\mathscr{A}$
  $n := \|\mathscr{A}\|$                    %% the number of control locations in $\mathscr{A}$
  $X :=$ the set of all clock variables occurred in $\mathscr{A}$
  $i := 0,\ j := 0,\ \mathsf{seen} := \emptyset$
  repeat
    $\mathscr{I}(l_j) :=$ the given (original) invariant of $l_j$
    $j := j+1$
  until $j < n$
  repeat
    if $i = 0$ then
      $\mathsf{inv}(l_i) := \{x = y \mid x, y \in X$ where $x$ and $y$ are not identical$\}$
    else
      $\mathsf{inv}(l_i) := \emptyset$
    $k := 0,\ \mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_i, \mathscr{A})$
    if $\mathsf{In} = \emptyset \wedge i > 0$ then
      $\mathscr{A} := \mathscr{A} \setminus {}^{\mathsf{out}}\mathsf{trans}(l_i, \mathscr{A})$
    else
      while $\mathsf{In} \neq \emptyset \ \wedge\ (k = 0 \vee \mathsf{inv}(l_i) \neq \emptyset)$ do
        choose $\tau \in \mathsf{In}$
        $\mathsf{In} := \mathsf{In} \setminus \{\tau\}$
        if $\tau \notin \mathsf{seen}$ then
          if $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ is a contradiction then
            $\mathscr{A} := \mathscr{A} \setminus \{\tau\}$          %% the idle transition
          else
            $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
        $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup\ {}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup\ \overline{\mathsf{atom}}(\mathsf{R}_\tau)$
        if $\mathsf{inv}(\tau) \wedge \mathscr{I}(l_i)$ is a contradiction then
          $\mathscr{A} := \mathscr{A} \setminus \{\tau\}$            %% the idle transition
        else
          $k := k+1$
          if $k = 1$ then
            $\mathsf{inv}(l_i) := \mathsf{inv}(\tau)$
          else
            $\mathsf{inv}(l_i) := \mathsf{join}(\mathsf{inv}(l_i), \mathsf{inv}(\tau))$
      $\mathscr{I}(l_i) := \mathscr{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$
      $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_i)$
      while $\mathsf{Out} \neq \emptyset$ do
        choose $\tau \in \mathsf{Out}$
        $\mathsf{Out} := \mathsf{Out} \setminus \{\tau\}$
        if $\tau \notin \mathsf{seen}$ then
          if $\mathscr{I}(l_i) \wedge \mathsf{G}_\tau$ is a contradiction then
            $\mathscr{A} := \mathscr{A} \setminus \{\tau\}$            %% the idle transition
          else
            $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
    $i := i+1$
  until $i < n$
  $\mathscr{I}_{\mathscr{A}} := \{l_i \mapsto \mathscr{I}(l_i)\}$
  return $(\mathscr{A}, \mathscr{I}_{\mathscr{A}})$
```

- *if* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A}, \mathscr{I}_{\mathscr{A}})$ *then* $\mathscr{A} \doteq \mathscr{A}_1$*, and*
- *for each control location* l*,* $\mathsf{inv}(l)$ *consists of only unbounded constraints.*

Proof.
- In timed automata, the number of control locations ($\|\mathscr{A}\|$) is a finite number, say n. Hence, the first two **repeat** loops halt after n steps. Besides, each timed automaton consists of a finite automaton which describes the system control states and its transitions. Hence, each control location has only a finite number of incoming and outgoing discrete transitions. Thus, the other two **while** loops will also stop after a finite number of steps.
- According to Lemma 5 and the algorithm $\mathscr{A}$ is updated only by removing some idle transitions. Therefore, the output automaton will be either the exact same automaton, or it will be an automaton with a smaller number of idle transitions.
- The above argument can be easily derived from the definition of $inv(\tau)$ and the definition of $\mathsf{inv}(l)$ in the algorithm.

□

Note 2. Since $\mathscr{A} \doteq \mathscr{A}_1$ according to the previous theorem, in the sequel we may use $\mathscr{A}$ and $\mathscr{A}_1$ interchangeably.

The new constraint $\mathscr{I}_{\mathscr{A}}(l)$ implies the original invariant $\mathscr{I}(l)$ and moreover it extracts a stronger clock constraint which should hold as long as we stay in l. We prove this in the next two theorems.

Theorem 2. *If* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A}, \mathscr{I}_{\mathscr{A}})$*, then* $\mathscr{I}_{\mathscr{A}}(l) \Rightarrow \mathscr{I}(l)$ *for each control location* l *in* $\mathscr{A}$*, where* $\mathscr{I}(l)$ *is the original invariant of* l *in* $\mathscr{A}_1$*.*

Proof. $\mathscr{I}_{\mathscr{A}}(l) = \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$. According to the definition of $\Rightarrow$, we need to show that if $u \models \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ then $u \models \mathscr{I}(l)$, for any valuation function u. Let $\mathscr{I}(l) = \{a_1, ...a_n\}$ and $\mathsf{simp}(\mathsf{inv}(l)) = \{b_1, ...b_m\}$. Therefore, $u \models \bigwedge_{1 \leq i \leq n} a_i \wedge \bigwedge_{1 \leq j \leq m} b_j$, and hence, for each $1 \leq i \leq n$, $u \models a_i$. By definition, this means that $u \models \mathscr{I}(l)$. □

The next theorem shows that $\mathscr{I}_{\mathscr{A}}$ associates with each control location l a set of new invariants.

Theorem 3. *If* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A}, \mathscr{I}_{\mathscr{A}})$*, then* $u \models \mathscr{I}_{\mathscr{A}}(l)$*, for each reachable configuration* $\langle l, u \rangle$ *in* $\mathscr{S}_{\mathscr{A}1}$*. In other words,* $\mathscr{I}_{\mathscr{A}}(l)$ *is invariant in* l*.*

Proof. Since $\mathscr{I}_{\mathscr{A}}(l) = \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$, we need to prove that $u \models \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ where $\mathscr{I}(l)$ is the original given invariant of location l (cf. Definition 1). To this end, we show that $u \models \mathscr{I}(l)$ and $u \models \mathsf{simp}(\mathsf{inv}(l))$. The first part holds by definition. For the second part, by Lemma 4 we only need to prove that $u \models \mathsf{inv}(l)$. We split the proof into two steps according to whether $\langle l, u \rangle$ is reached by a discrete transition τ or whether it is reached by a delay transition d.

- Assume that $\langle l, u \rangle$ is reached by a discrete transition τ (i.e. $... \overset{\tau}{\mapsto} \langle l, u \rangle$). Then $u \models \mathsf{inv}(\tau)$ by Lemma 3. Hence, $u \models \mathsf{join}(\mathsf{inv}(l), \mathsf{inv}(\tau))$ by Lemma 2 (2nd item). This, according to the algorithm, is the updated value of $\mathsf{inv}(l)$. Therefore, $u \models \mathsf{inv}(l)$.
- Assume that $\langle l, u \rangle$ is reached by a delay transition d. Then, there exist a discrete transition τ, a valuation u_1 and a delay value $d_1 \in \mathbb{R}^+$ such that $... \overset{\tau}{\mapsto} \langle l, u_1 \rangle \overset{d_1}{\mapsto} \langle l, u \rangle$, i.e. $u := u_1 + d_1$ and $\langle l, u_1 \rangle$ is reached by the discrete transition τ. Therefore, according to the previous part we get $u_1 \models \mathsf{inv}(l)$. According to Theorem 1 (last item), all elements of $\mathsf{inv}(l)$ are unbounded, hence, $u_1 + d_1 \models \mathsf{inv}(l)$ by Lemma 1. This completes the proof.

□

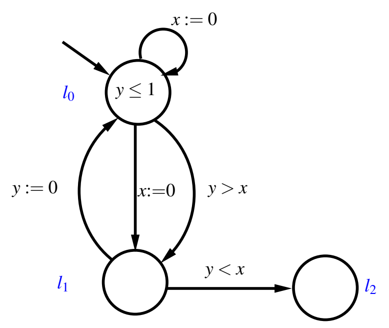


Figure 1: $\mathscr{A}$. In this timed automaton, x and y are clock variables.

3.2 Example

We illustrate the CIPM algorithm using an example taken from [21], cf. the timed automaton in Figure 1. In this model, x and y are clock variables. l_0, l_1 and l_2 are control locations. Our intention is to prune this automaton and to find a new set of invariants for each control location according to the CIPM algorithm.

The model starts with the initial value of $x = y = 0$ in location l_0. $y \leq 1$ is the original invariant in l_0, i.e. $\mathscr{I}(l_0) = \{y \leq 1\}$. In other words, we can stay in l_0 only as long as the value of y does not exceed 1. Once this value has passed 1 then one of the outgoing transitions must be taken out of this state (e.g. $l_0 \overset{x:=0}{\mapsto} l_1$). For the other locations we have: $\mathscr{I}(l_1) = \mathscr{I}(l_2) = \emptyset$.

Initially, $n := 3$, $X := \{x,y\}$, $i := 0$, $j := 0$ and $\mathsf{seen} := \emptyset$. In the first **repeat** loop, for $0 \leq j < 3$ the algorithm collects the original invariants at l_j. So, $\mathscr{I}(l_0) = \{y \leq 1\}$ and $\mathscr{I}(l_1) = \mathscr{I}(l_2) = \emptyset$. In the second **repeat** loop, we get: $\mathsf{inv}(l_0) := \{x = y\}$, $k := 0$, and $\mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_0, \mathscr{A}) = \{l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. Since the condition of the first **if** loop does not hold, the **while** loop must be activated. Here, $\mathsf{In} \neq \emptyset \wedge (k = 0 \vee \mathsf{inv}(l_0) \neq \emptyset)$ holds, we choose $\tau := l_0 \overset{x:=0}{\mapsto} l_0$ from In, and let $\mathsf{In} := \{l_1 \overset{y:=0}{\mapsto} l_0\}$. $\tau \notin \mathsf{seen}$ (which is $\emptyset$) and $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \{y \leq 1\} \wedge \emptyset$. By definition this is $y \leq 1$, which is not a contradiction. Hence, $\mathsf{seen} := \mathsf{seen} \cup \{\tau\} = \{l_0 \overset{x:=0}{\mapsto} l_0\}$. Now, since for this τ we have: ${}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) = \emptyset$, ${}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) = {}^{\mathsf{un}}(\{y \leq 1\}) = \emptyset$ and $\overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$, we derive: $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \cup {}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \cup \overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$. Then, $\mathsf{inv}(\tau) \wedge \mathscr{I}(l_0) = \{x \leq y\} \wedge \{y \leq 1\} = x \leq y \wedge y \leq 1$ which is not a contradiction. Hence, $k := k + 1 = 1$ and $\mathsf{inv}(l_0) := \mathsf{inv}(\tau) = \{x \leq y\}$.

Once more, since $\mathsf{In} = \{l_1 \overset{y:=0}{\mapsto} l_0\} \neq \emptyset$, we go through the **while** loop of $\mathsf{In} \neq \emptyset \wedge (k = 0 \vee \mathsf{inv}(l_0) \neq \emptyset)$. Here, we bring the result of computations briefly. We choose $\tau := l_1 \overset{y:=0}{\mapsto} l_0$, we get $\mathsf{In} := \emptyset$, and since $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \emptyset \wedge \emptyset = \mathsf{true}$, $\mathsf{seen} := \{l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. $\mathsf{inv}(\tau) := \emptyset \wedge \emptyset \wedge \{y \leq x\} = \{y \leq x\}$ and $k := k + 1 = 2$, hence, $\mathsf{inv}(l_0) := \mathsf{join}(\{x \leq y\}, \{y \leq x\}) = \mathsf{join}(x \leq y, y \leq x) = \mathsf{true}$. At this point, since $\mathsf{In} := \emptyset$, we leave the **while** loop, and put $\mathscr{I}(l_0) := \mathscr{I}(l_0) \wedge \mathsf{simp}(\mathsf{inv}(l_0)) = \{y \leq 1\} \wedge \mathsf{simp}(\mathsf{true}) = \{y \leq 1\} \wedge \mathsf{true}$, which is equivalent to $y \leq 1$. So, we gain no new invariant for l_0.

Next, $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_0) = \{l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{y>x}{\mapsto} l_1\}$. We choose $\tau := l_0 \overset{x:=0}{\mapsto} l_1$. $\mathsf{Out} := \mathsf{Out} \backslash \{\tau\} = \{l_0 \overset{y>x}{\mapsto} l_1\}$. $\tau \notin \mathsf{seen}$, and $\mathscr{I}(l_0) \wedge \mathsf{G}_\tau = y \leq 1$ which is not a contradiction. Hence, $\mathsf{seen} := \{l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. Then, we choose $\tau := l_0 \overset{y>x}{\mapsto} l_1$, and for the same reason, $\mathsf{seen} := \{l_0 \overset{y>x}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. $\mathsf{Out} := \mathsf{Out} \backslash \{\tau\} = \emptyset$, hence we leave this **while** loop, and put $i := i + 1 = 1$. $1 < 3$, so the **repeat** loop must be gone through once more.

The same process should be repeated again. The interesting part in this second round is that for $\tau := l_0 \overset{y>x}{\mapsto} l_1 \in {}^{\mathsf{in}}\mathsf{trans}(l_1, \mathscr{A})$, we get $\mathsf{inv}(\tau) := \{y > x\} \wedge \emptyset \wedge \emptyset = \{y > x\}$. For the other transition $\tau := l_0 \overset{x:=0}{\mapsto} l_1 \in {}^{\mathsf{in}}\mathsf{trans}(l_1, \mathscr{A})$, $\mathsf{inv}(\tau) := \{x \leq y\}$, and $\mathsf{inv}(l_1) := \mathsf{join}(\{x \leq y\}, \{y > x\}) = \{x \leq y\}$. In the

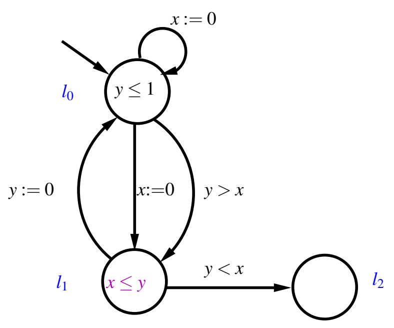


Figure 2: New invariants.

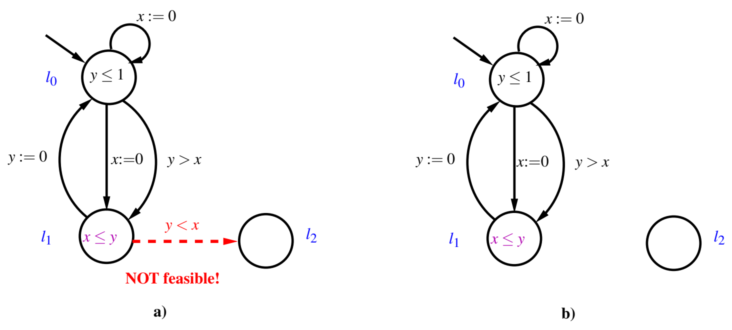


Figure 3:

end, $\mathscr{I}(l_1) := \emptyset \wedge \{x \leq y\} = x \leq y$. This means that for this location we actually obtain a new invariant which is $x \leq y$. See Figure 2.

Then, $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_1) = \{l_1 \overset{y<x}{\mapsto} l_2,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. We choose $\tau := l_1 \overset{y<x}{\mapsto} l_2$, then $\mathsf{Out} := \{l_1 \overset{y:=0}{\mapsto} l_0\}$. $\tau \notin \mathsf{seen}$, and $\mathscr{I}(l_1) \wedge \mathsf{G}_\tau = x \leq y \wedge y < x$, which is a contradiction! This is shown in Figure 3(a). Therefore, the automaton is updated to $\mathscr{A} := \mathscr{A} \backslash \{l_1 \overset{y<x}{\mapsto} l_2\}$. We continue the algorithm with this new automaton. For this, see Figure 3(b). The other transition $l_1 \overset{y:=0}{\mapsto} l_0$ is already in seen. So this loop terminates here.

With this new automaton, for location l_2 we obtain: $\mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_2, \mathscr{A}) = \emptyset$. Hence, the **while** loop of $\mathsf{In} \neq \emptyset \ \wedge \ (k = 0 \vee \mathsf{inv}(l_2) \neq \emptyset)$ can not be entered. There is also no transition out of this location.

In the end we obtain $\mathscr{A} := \mathscr{A} \backslash \{l_1 \overset{y<x}{\mapsto} l_2\}$, and $\mathscr{I}_{\mathscr{A}} := \{l_0 \mapsto \{y \leq 1\},\ l_1 \mapsto \{x \leq y\},\ l_2 \mapsto \emptyset\}$. Figure 3(b).

4 Implementation and Experimental results

We have developed a prototypical implementation of the CIPM algorithm in C++. The code takes a timed automaton in UPPAAL [3] syntax as input and computes the new invariants for each location, as well as removing the spurious transitions as shown in the pseudo-code of Alg. 1[7]. The actual implementation of CIPM thus consists of code for operating on the abovementioned datastructure, to remove transitions and modify invariants of the automaton.

[7]For parsing the UPPAAL input file into a suitable C++ datastructure, we used code that was provided to the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB TR14/AVACS) gproject by Gerd Behrmann.

The implementation consists of about 1000 lines of code. For checking whether some invariants togther constitute a contradiction (such a condition occurs in three places in the pseudo-code of Alg. 1), we use ICS [9], which is a solver for linear arithmetic. Likewise, we use ICS to check the implications occurring in the definition of join (see Equation 1).

Timed automata in UPPAAL are in some respects more general than the timed automata that we have defined in this paper. In particular, UPPAAL automata may have integer variables in addition to the real-valued clock variables. Integer variables only change their value when there is an explicit assignment. The presence of integer variables has repercussions on the clocks since clock assignments, guards and invariants may involve linear expressions of those variables. E.g., there may be a guard $x \leq 2i + j$ where x is a clock variable and i and j are integer variables. In our current code, integer variables are ignored, but we plan to extend the implementation to cater for them.

Another feature of UPPAAL is that of parallel composition of automata (called *processes* in UPPAAL parlance) into a *system*. In the current implementation, we assume a system that is composed of just one process. Again, we plan to extend the implementation so that it works for systems with more than one process.

There is however one aspect where the automata we consider here are more general than UPPAAL automata: In UPPAAL, there are no disjunctive invariants. The invariant of a location, as well as the guard of a transition, is a sequence of equations and disequalities, interpreted as a conjunction. Therefore, we cannot directly implement the invariant $g \vee g'$ occurring in the definition of join, and we approximate it as the trivial invariant true. In some cases, we might be able to do better than that. E.g., $(x \geq 2 \wedge y \geq 1) \vee (x \geq 1 \wedge y \geq 2)$ implies $x \geq 1 \wedge y \geq 1$, which is stronger than true. However what the best approximation we can express with UPPAAL syntax is and how we can compute it is a nonobvious question left for future work as well. Another idea would be to split a location requiring a disjunctive invariant into two locations.

We have tested the implementation on the example of Fig. 1 and some other hand-designed examples. Our prototype tool transformed these examples in the expected way which increases our confidence that the proposed pseudo-code for CIPM accomplishes what we intend it to do.

5 Conclusion

Our work proposes the CIPM algorithm which accomplishes two goals. First, it automatically generates invariants for timed automata models. The algorithms computes new invariants in each control location of a timed automaton taking logical conditions on the original state invariants imposed by incoming transitions into account. Second, we defined the notion of idle transitions which helps in reducing the size of a timed automaton by eliminating transitions that can never be traversed.

We presented a preliminary implementation of the CIPM algorithms. At the current stage it is too early to talk about the performance of the implementation since the runtime for the examples including the one discussed above is, of course, negligible. However, the algorithm looks at each location and each transition at most once and thus its complexity should be low. How this would change if we ran the algorithm repeatedly on the same automaton is a different matter.

Future work includes the definition of a counterexample guided abstraction refinement technique using our proposed invariant generation approach extended by suitable predicate abstractions. Currently, we are incorporating the invariants computed by CIPM into an abstraction framework for timed automata. The idea is to couple each control location with its corresponding invariant and to use these invariants to determine a predicate abstraction for the respective pair of states. To illustrate the approach we considered an example from [21].

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *TACAS'02*, 2002.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems. Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, volume 3185 of *LNCS*, pages 200–236, Bertinoro, Italy, Sep 2004. Springer–Verlag.

[4] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer–Verlag, 2001.

[5] Michael Colón and Tomás E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *CAV'98*, pages 293–304, 1998.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. *POPL'77*, pages 238–252, 1977.

[7] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design(FMCAD)*. Springer-Verlag, November 2002.

[8] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification (CAV'99)*. Springer-Verlag, 1999.

[9] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.

[10] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244, 2004.

[12] Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in uppaal. In *Model Checking and Artificial Intelligence, MoChArt'06*, pages 51–66, 2006.

[13] Agata Janowska and Pawel Janowski. Slicing of Timed Automata with Discrete Data. *Fundamenta Informaticae*, 72(1-3):181–195, 2006.

[14] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *CAV'05*, pages 39–51, 2005.

[15] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate Abstraction via Symbolic Decision Procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[16] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[17] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A Symbolic Approach to Predicate Abstraction. In *CAV'03*, pages 141–153, 2003.

[18] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Proc. of Computer Aided Verification, CAV*, pages 424–437, 2006.

[19] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proc. of Computer Aided Verification, CAV*, pages 123–136, 2006.

[20] Kenneth L. McMillan and Nina Amla. Automatic Abstraction without Counterexamples. In *TACAS*, pages 2–17, 2003.

[21] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time System. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[22] Sriram Sankaranarayanan, Michael Colón, Henny B. Sipma, and Zohar Manna. Efficient strongly relational polyhedral analysis. In *VMCAI*, 2006.

[23] Uffe Sørensen and Claus Thrane. Slicing for Uppaal. Master's thesis, Aalborg university, 2008.

Invariants and Robustness of BIP Models

(Report on Ongoing Work)

Jan Olaf Blech, Thanh-Hung Nguyen, Michaël Périn
Verimag Laboratory, Université de Grenoble, France

Abstract

Verification techniques have become popular in software and hardware development. They increase confidence and potentially provide rich feedback. However, with increasing complexity verification techniques are more likely to contain errors themselves. Many verification tools use invariants of the considered systems for their analysis. These invariants are often generated by the verification tools in a first step. The correctness of these invariants is crucial for the analysis results.

In this paper we present on-going work addressing the problem of automatically generating realistic and guaranteed correct invariants. Since invariant generation mechanisms are error-prone, after the computation of invariants by a verification tool, we formally prove that the generated invariants are indeed invariants of the considered systems using a higher-order theorem prover and automated techniques. We regard invariants for BIP models. BIP (behavior, interaction, priority) is a language for specifying asynchronous component based systems. Proving that an invariant holds often requires an induction on possible system execution traces. For this reason, apart from generating invariants that precisely capture a system's behavior, inductiveness of invariants is an important goal.

We establish a notion of robust BIP models. These can be automatically constructed from our original non-robust BIP models and over-approximate their behavior. We motivate that invariants of robust BIP models capture the behavior of systems in a more natural way than invariants of corresponding non-robust BIP models. Robust BIP models take imprecision due to values delivered by sensors into account. Invariants of robust BIP models tend to be inductive and are also invariants of the original non-robust BIP model. Therefore they may be used by our verification tools and it is easy to show their correctness in a higher-order theorem prover.

The presented work is developed to verify the results of a deadlock-checking tool for embedded systems after their computations. Therewith, we gain confidence in the provided analysis results.

1 Introduction

Verification tools to ensure properties of complex systems have become popular in many application areas. One major goal is to guarantee safety and security properties of the considered systems. These can be computed by generating invariants of the considered systems in a first step and analyzing them. However, as verification tools become more and more complex it is not always easy to see if they are themselves working correctly. An incorrect verification tool might state a wrong property about a system.

Guided by first experiments on automatically verifying – thereby establishing a correctness certificate – the results of a verification tool after they have been computed in a higher-order theorem prover (Coq), we verify invariants of given systems (BIP models) within a theorem prover, we introduce a notion of robustness for these systems. The invariants that are subject to this paper are computed and used by the D-Finder [BBSN08] tool that decides deadlock-freedom of systems modeled in the BIP language [BBS06]. The BIP language features the descriptive power to model asynchronous systems and is designed for building real-time embedded systems consisting of heterogeneous components. Invariants that are both inductive and capture the behavior of our systems in an adequate way are highly desirable for our analysis and verification tools.

In our case study, we require invariants to be inductive in order to be verified automatically by deductive methods. We motivate a technique that is likely to produce inductive invariants: we establish a notion of robust BIP models. These models take imprecision of values due to physical measurements into account. Invariants of robust systems are aimed at describing a system's behavior in a realistic way taking sensor sensitivity and imprecision into account while preserving the necessary precision to be used as basis for analysis results. Our invariants are suitable for automated verification using a discrete

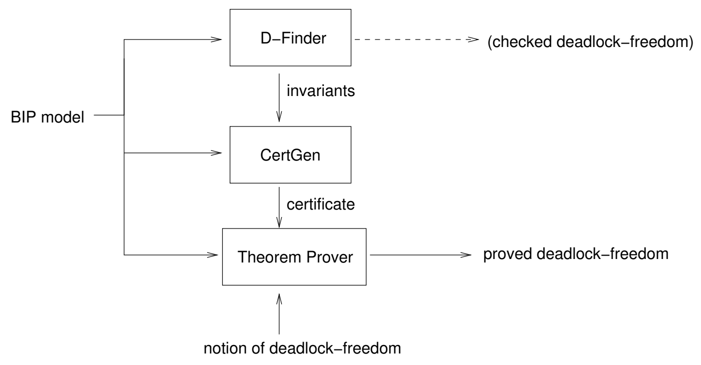


Figure 1: Our Methodology

semantics. We present a mapping from non-robust to robust systems and prove that invariants of robust systems are also invariants of the original non-robust systems. This allows us to reuse invariant based analysis results for these systems.

1.1 Our Case Study: Guaranteeing Correctness of the Results of a Verification Tool

Robust BIP models are used to make the process (called certification) of automatically proving the results of a deadlock-detection verification tool easier thereby guaranteeing the correctness of its verdict. The overall approach of this tool and the verification process guaranteeing that its results are correct is described in the following two paragraphs.

1.1.1 The Deadlock-detection Tool D-Finder and the Certification of its Results

The deadlock-detection tool D-Finder takes BIP models as inputs and decides whether they are deadlock free. In order to do this, in a first step invariants of these are computed. These are analyzed for potential deadlocks. D-Finder is aimed to be safe in a way that it might detect false positives, i.e., potential deadlocks that do not exist. The computation of invariants is the most sophisticated step within D-Finder. In addition to D-Finder's algorithms an external tool Omega [Ome00] is used in the invariant generation process to perform quantifier elimination. In a second step these invariants are checked to be deadlock-free by using the external SMT solver Yices [DM06] and a definition of deadlock-states.

Verifying that invariants hold is used for guaranteeing the absence of deadlocks in our guiding case study [BP08, BP08a]. The methodology underlying this case study is depicted in Figure 1. BIP models are passed to D-Finder, the deadlock-detection tool. In this paper, we do not trust D-Finder in a first place, but want to establish proofs, that it has indeed worked correctly for each run of this tool. Apart from detecting deadlocks, a certificate is generated by some part of the tool (denoted CertGen). This certificate comprises a proof of deadlock-freedom and is passed to a theorem prover. The D-Finder tool computes invariants and uses them to decide whether a system is deadlock-free or not. Most important to this paper is the fact, that the certificates contain these invariants as well as a proof script that is generated by the certificate generator proving that the invariants do indeed hold. The theorem prover uses this proof script to prove that a BIP model is indeed deadlock-free.

1.1.2 Proving Deadlock-freedom

To verify that a system is indeed deadlock-free in the theorem prover, we have to check the certificates. We break this task of verifying deadlock-freedom for a given BIP model *BM* down into different subtasks

$$
\begin{array}{ll}
1. & \forall s.\mathit{ReachableStates}_{BM}(s) \longrightarrow \mathit{Enabled}_{BM}(s) \\
 & \qquad\qquad \uparrow \quad \textit{transitivity} \\
2. & \forall s.\mathit{ReachableStates}_{BM}(s) \longrightarrow \neg \mathit{DIS}_{BM}(s) \quad \textit{and} \quad [\text{PO1}]\ \forall s.\neg \mathit{DIS}_{BM}(s) \longrightarrow \mathit{Enabled}_{BM}(s) \\
 & \qquad\qquad \uparrow \quad \textit{transitivity} \\
3. & [\text{PO2}]\ \forall s.\mathit{ReachableStates}_{BM}(s) \longrightarrow \Psi_{BM}(s) \quad \textit{and} \quad [\text{PO3}]\ \forall s.\Psi_{BM}(s) \longrightarrow \neg \mathit{DIS}_{BM}(s)
\end{array}
$$

Figure 2: Verifying Deadlock-freedom: The Meta-Proof

as shown in Figure 2. The proofs for these subtasks are composed to prove the top line. In the figure, we use the following definition of enabled states capturing BIP states from which a state transition to a succeeding state is possible:

$$\mathit{Enabled}_{BM}(s) \equiv \exists s'.(s,s') \in [\![BM]\!]_{BIP} \wedge s \neq s'$$

The $[\![BM]\!]_{BIP}$ denotes the set of possible state transitions of the BIP model *BM* thereby defining its semantics. Furthermore, we use a definition of reachable states $\mathit{ReachableStates}_{BM}$ for a BIP model *BM* which is defined inductively in a way that the initial state is reachable and all succeeding states of a reachable state are reachable.

The task of verifying deadlock-freedom is performed by using the refinement shown in Figure 2:

1. The top line in the figure shows our notion of deadlock-freedom for a BIP model. We ultimately want to prove this line. We demand that all reachable states have at least one succeeding state. Thus, there is no reachable state where no transition is possible.

2. Instead of a direct proof, we follow the architecture of D-Finder and take advantage of the invariants discovered by D-Finder (Ψ_{BM},$\neg \mathit{DIS}_{BM}$): we conduct the proof shown in the second line consisting of two proof goals. The first goal reformulates the notion of enabled states and puts a predicate $\neg \mathit{DIS}_{BM}$ instead. Thus, we may verify that this goal holds for a BIP model *BM*. The second proof goal [PO1] (Proof Obligation 1) states that whenever one proves the first goal correct, the correctness property of the first line is implied – thereby guaranteeing the more human readable notion of correctness.

3. The third line splits the first proof goal of the second line into two proof goals [PO2] and [PO3]. This line introduces an invariant $\Psi_{BM}(s)$ as a transitive step. This invariant is part of the certificate. To use this line in our proofs we have to show that it also implies the first line.

The $\neg \mathit{DIS}_{BM}$, and Ψ_{BM} are provided by D-Finder.

Most tasks in this proof scheme are relatively easy from a technical point of view. It is sufficient to prove the three proof obligations and construct the proof for the first line via transitivity of the implication rule. This, as well as proving [PO1] and [PO3] can easily be done automatically. However, the generation of the invariant $\Psi_{BM}(s)$ can be error prone. Therefore the automatic verification that

$$[\text{PO2}]\ \forall s.\mathit{ReachableStates}_{BM}(s) \longrightarrow \Psi_{BM}(s)$$

does hold is a challenging tasks of our methodology. It captures the correctness of the main task of the D-Finder tool: finding invariants. The work presented in the rest of this paper concentrates on generating realistic invariants, reports on experiences with case studies and suggests ways to improve the computation of the invariants thereby making the verification task easier.

1.2 Overview

We introduce the BIP semantics for modeling our systems in Section 2 and present a small example. A discussion of invariants of BIP models and their properties is given in Section 3. Section 4 introduces robust BIP models and a motivation for and proofs of their properties. The benefits of robust BIP models in verifying invariants for our example application scenario are presented in Section 5. Related work is discussed in Section 6. In Section 7 we draw a conclusion and present our goals for future work.

2 BIP Models and their Semantics

In this section we describe the semantics of BIP models. BIP is a software framework designed for building embedded systems consisting of heterogeneous components. It is characterized by three modeling layers: behavior of components encoded as transition systems extended with variables, interactions between components realized via communication ports and priority rules which reduce non-determinism between interactions (BIP stands for Behaviors + Interactions + Priorities). Apart from code generation the BIP tool chain comprises static analyses tools for checking properties like deadlock-freedom.

BIP models are composed of atomic components [BBS06, BBSN08] that can be composed into larger components. Components are state transition systems. They communicate via ports with each other.

Definition 2.1 (Atomic BIP component)**.** *An atomic component B_i can be represented by a tuple (L_i, P_i, T_i, V_i) such that*

- *V_i is a set of variables,*
- *$L_i = \{l_i^0, l_i^1, l_i^2, ..., l_i^k\}$ is a set of control locations,*
- *P_i is a set of ports,*
- *$T_i \subseteq L_i \times (X_i \rightarrow bool) \times (X_i \rightarrow X_i) \times P_i \times L_i$ is a set of transitions, each one comprising a location, a guard function $g : X_i \rightarrow bool$, an update function $f : X_i \rightarrow X_i$, a port, and a succeeding location. The X_i denote valuation functions: mappings from variables V_i to their values D_i.*

The guard functions are predicates and are formulated on the variables appearing in an atomic component. The following definition describes a language for these predicates which we use for the work presented in this paper:

Definition 2.2 (Guard language)**.** *A predicate ϕ belongs to the guard language iff it is constructed using the following rules:*

$$\begin{aligned}
&\phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid e\\
&e ::= e' < e' \mid e' \leq e' \mid e' = e' \mid e' \neq e' \mid e' \geq e' \mid e' > e'\\
&e' := op \mid e' + e' \mid e' - e' \mid val \cdot e'\\
&op ::= var \mid val
\end{aligned}$$

Assuming the guard function appears in the ith component, the $var \in V_i$ are variables appearing in it. $val \in D_i$ denotes some numerical type. The variables $var \in V_i$ are mapped to the same type. Typical types are reals and integers.

The semantic interpretation of the guard language follows the rules of predicate logic and arithmetic. Note, that the expressibility of the guard language corresponds to Presburger arithmetic when D_i denotes integer values.
The atomic components of a BIP model are connected via ports. They communicate via interactions. Thus, a composed component is defined as a tuple $((B_1,...,B_n), \mathit{Interactions})$ comprising the atomic components and their interactions.
An interaction is a tuple $(p_1,\ldots,p_n)$ where p_i is a port of the atomic component B_i or $\perp$ if B_i is not involved in this interaction.
The state of an atomic component B_i is a tuple (l_i,x_i) comprising a location and a variable valuation function. The state of a BIP model is the product of the state of its atomic components: $(L_1 \times X_1) \times \ldots \times (L_n \times X_n)$.
A transition relation for BIP models is defined via the following predicate.

Definition 2.3 (Transition Relation)**.** *A transition relation for a* BIP *model BM (denoted $[\![BM]\!]_{BIP}$) for* BIP *models is defined via the following rule:*

$$\frac{(p_1,\ldots,p_n) \in \mathit{Interactions} \qquad \forall i \in \{1..n\}.\ (l_i,g_i,f_i,p_i,l_i') \in B_i \wedge (g_i(x_i) \wedge x_i' = f_i(x_i)) \vee (l_i = l_i' \wedge p_i = \perp \wedge x_i' = x_i)}{(((l_1,x_1),...,(l_n,x_n))\ ,\ ((l_1',x_1'),...,(l_n',x_n'))) \in [\![BM]\!]_{BIP}}$$

A state transition from a given reachable state is possible if there is an interaction such that there is in each component either a possible state transition labeled with the port or the component is not involved in the interaction. Furthermore, in order to do a transition of an atomic component the appropriate guard functions must evaluate to true. To derive the succeeding states the update functions are performed on the valuation functions of the involved atomic components.
Using the transition relation, reachable states of a BIP model are defined in the following definition.

Definition 2.4 (Reachable States)**.** *The predicate $\mathit{ReachableStates}_{BM}$ indicating reachable states of a* BIP *model BM with an initial state $s_0 \in (L_1 \times X_1) \times \ldots \times (L_n \times X_n)$ is defined via the following inductive rules:*

$$\frac{}{\mathit{ReachableStates}_{BM}(s_0)}$$

$$\frac{\mathit{ReachableStates}_{BM}(s) \quad (s,s') \in [\![BM]\!]_{BIP}}{\in \mathit{ReachableStates}_{BM}(s')}$$

The first rule says that the initial state is reachable. The second inference rule captures the transition behavior of BIP using the transition relation.

An Example Figure 3 shows a temperature control system [BBSN08, ACH+95] modeled in BIP. It controls the cooling of a reactor by moving two independent control rods. It is a simple example to illustrate D-Finder and its invariants. The goal is to keep the temperature between $\theta = 100$ and $\theta = 1000$. When the temperature reaches the maximum value one of the rods has to be used for cooling. The BIP model comprises three atomic components one for each rod and one for the controller. Each contains a state transition system. Transitions can be labeled with guard conditions, valuation function updates, and a port. The components interact via ports thereby realizing cooling, heating, and time elapsing interactions. Initially the system starts in locations l_1, l_3, l_5 with values of $t_1 = 3600$, $t_2 = 3600$, $\theta = 100$. Note, that the system does indeed contain a deadlock. Since we are interested in verifying that invariants hold, this, however, is not important in the context of this paper.

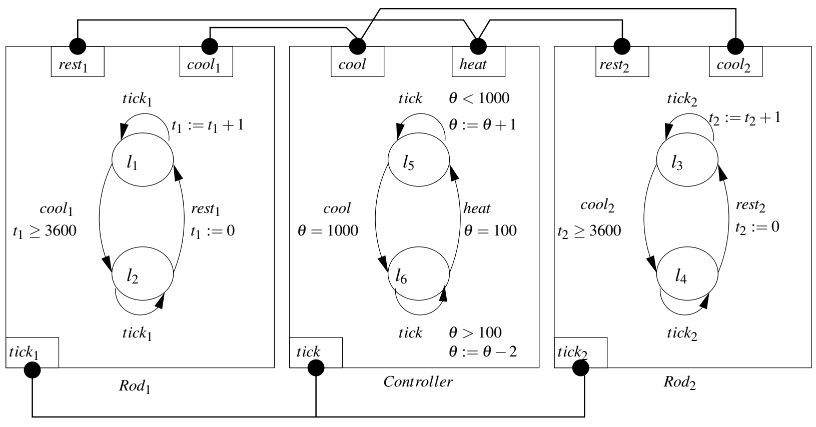


Figure 3: Temperature Control System

3 Invariants of BIP Models

In this section we discuss invariants for BIP models and motivate desired properties.

Definition 3.1 (Invariant). *A predicate I over the states of a* BIP *model BM is an invariant of BM iff* $\forall s.ReachableStates_{BM}(s) \longrightarrow I(s)$

The generated invariants Ψ_{BM} that we are verifying in our theorem prover are composed of component invariants (*CI*) and interaction invariants (*II*):

$$\Psi_{BM} \overset{def}{=} CI_1 \wedge \ldots \wedge CI_n \wedge II_1 \wedge \ldots \wedge II_m$$

The following invariants are computed by D-Finder to approximate the behavior of the components (*component invariants*) in the example from Figure 3:

- $CI_1 = (at_{l1} \wedge t_1 \geq 0) \vee (at_{l2} \wedge t_1 \geq 3600)$
- $CI_2 = (at_{l3} \wedge t_2 \geq 0) \vee (at_{l4} \wedge t_2 \geq 3600)$
- $CI_3 = (at_{l5} \wedge 100 \leq \theta \leq 1000) \vee (at_{l6} \wedge 100 \leq \theta \leq 1000)$

at_i is a predicate denoting the fact that we are at location i in a component. In addition to component invariants D-Finder computes interaction invariants capturing the behavior induced by interactions between the atomic components. In addition to component invariants D-Finder generates interaction invariants which capture the behavior of components interacting with each other. An example for an interaction invariant for the given BIP model is shown below:

$$II_1 = (at_{l1} \wedge t_1 = 0) \vee (at_{l3} \wedge t_2 = 0) \vee (at_{l5} \wedge 101 \leq \theta \leq 1000) \vee (at_{l6} \wedge (\theta = 1000 \vee 100 \leq \theta \leq 998))$$

Definition 3.2 (Inductive Invariants). *An invariant I is inductive for a* BIP *model BM iff*

1. *It holds for the initial state* s_0 *of BM:* $I(s_0)$
2. $\forall s\, s'.I(s) \wedge (s,s') \in [\![BM]\!]_{BIP} \longrightarrow I(s')$

By using the definition of reachable states we can prove the following theorem which is independent of concrete BIP models:

Theorem 3.1. *Every inductive invariant of a* BIP *model BM is also an invariant of BM.*

Both CI_1 and CI_2 are inductive. CI_3 is not inductive because it evaluates to true for a state where we are at control location $l6$ (at_{l6}) and $\theta = 101$. But, it does not hold for the succeeding state at_{l6} and $\theta = 99$. Note, that since a state where at_{l6} and $\theta = 101$ is never reached within a real system run, CI_3 is still an invariant.

For verifying invariants during certificate checking we perform an induction on the set of reachable states. For this reason, it is highly desirable if invariants are inductive.

Making Invariants Inductive by Strengthening We can make invariants inductive by strengthening them. Given an invariant I we can add some strengthening constraints C to create a stronger invariant I'. The proof that I' implies that I holds for a given state s is a trivial application of the conjunction elimination rule:

$$I'(s) \overset{def}{=} I(s) \wedge C(s) \longrightarrow I(s)$$

For example CI_3 can be made inductive by adding a constraint that at location $l6$, θ is always divisible by two $(2|\theta)$. We can now verify the inductive invariant and show that it implies the weaker non-inductive one. Thus, the non-inductive one, is proven to be an invariant, too.

We have experimented with techniques to strengthen invariants automatically. In many cases it is possible to “guess” the C constraints that make invariants inductive. The additional constraint could be constructed following the general method presented in [BM08] that refines the invariant to reach an inductive one.

However, sometimes strengthening invariants seems artificial. Adding the divisibility constraint mentioned above to a variable that represents a physical measure could also indicate some design flaw of the original system. We should not base the verification of a system on the fact that the temperature measured by some sensors is an even number.

So, instead of strengthening the constraints on physical measures, we introduce in this paper a way to model the uncertainty of measurement. We concentrate on finding slightly weaker invariants of systems that represent the nature of variables depending on physical measurements in a more natural way.

For each invariant there is always a weaker invariant that is inductive: $I \equiv true$ is the weakest invariant for all BIP models and is inductive.

4 Robust BIP Models

In this section we introduce robust BIP models. Robust BIP models and their invariants are aimed at describing systems in a more natural way. Specifically, the target of our approach are systems whose values represent physical measurements. These are performed by sensors, which are usually not exact but have some tolerance range of imprecision associated with them. The measured value can vary within this range differing from the actual value. To capture the behavior in BIP models that are based on this imprecise information, we introduce special sets of *measurement* variables. Guard functions depending on an exact *measurement* variable in a BIP model are changed in a way that they evaluate to true – i.e. may perform a transition – within a range of unpreciseness to achieve robust BIP models.

Robust BIP models are realized by exchanging guards by robust guards described by a robust guard language:

$$\begin{array}{ll}
\langle(\phi_1 \wedge \phi_2)\rangle_2 = \langle\phi_1\rangle_2 \wedge \langle\phi_2\rangle_2 & \langle(\phi_1 \vee \phi_2)\rangle_2 = \langle\phi_1\rangle_2 \vee \langle\phi_2\rangle_2 \\
\langle(e'_1 < e'_2)\rangle_2 = \langle e'_1\rangle_2 < \langle e'_2\rangle_2 & \langle(e'_1 \leq e'_2)\rangle_2 = \langle e'_1\rangle_2 \leq \langle e'_2\rangle_2 \\
\langle(e'_1 \geq e'_2)\rangle_2 = \langle e'_1\rangle_2 \geq \langle e'_2\rangle_2 & \langle(e'_1 > e'_2)\rangle_2 = \langle e'_1\rangle_2 > \langle e'_2\rangle_2 \\
\langle(e'_1 = e'_2)\rangle_2 = \langle e'_1\rangle_2 = \langle e'_2\rangle_2 & \langle(e'_1 \neq e'_2)\rangle_2 = \langle e'_1\rangle_2 \neq \langle e'_2\rangle_2 \\
\langle(e'_1 + e'_2)\rangle_2 = \langle e'_1\rangle_2 + \langle e'_2\rangle_2 & \langle(e'_1 - e'_2)\rangle_2 = \langle e'_1\rangle_2 - \langle e'_2\rangle_2 \\
\langle(val \cdot e')\rangle_2 = val \cdot \langle e'\rangle_2 & \langle val\rangle_2 = val
\end{array}$$

$$\begin{array}{ll}
\langle m\rangle_2 = m + \delta_m & \text{if } m \in V_R \\
\langle var\rangle_2 = var & \text{otherwise}
\end{array}$$

Figure 4: Function for Introducing the δ_m

Definition 4.1 (Robust guard language)**.** *A robust guard language describing predicates ϕ is defined for a set of measurement variables $V_R \subseteq V_i$ if the guard appears in the ith component in the following way:*

$$\begin{array}{l}
\phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid e \\
e ::= e' < e' \mid e' \leq e' \mid e' = e' \mid e' \neq e' \mid e' \geq e' \mid e' > e' \\
e' ::= op \mid e' + e' \mid e' - e' \mid val \cdot e' \\
op ::= var \mid val \mid m + \delta_m
\end{array}$$

$var \in (V_i \backslash V_R)$, $m \in V_R$, val denotes some numerical type var like reals or integers.

Each reference to a measurement variable within a robust guard function is accompanied by some unknown constant δ_m which captures its imprecision. Compared to the original guard language, the interpretation of the δ_m in the robust guard language requires some non-trivial definitions.

The semantical interpretation of guard functions is adapted in a way that a robust system is an abstraction of a non-robust system. This means that due to non-determinism it allows more possibilities of transition but preserves the transition possibilities of the non-robust system. Given a set of measurement variables V_R, for each $m \in V_R$ there is some unknown δ_m within some fixed range $-\Delta_m \leq \delta_m \leq \Delta_m$ ($\Delta_m \geq 0$). This range captures the level of imprecision which a value that represents a physical measurement can have.

A robust guard is constructed from a non-robust guard in two steps:

- In the first step, negations are eliminated by putting them to the lowest level thereby changing (in)equalities using a function $\langle ... \rangle_1$. This function is defined inductively on the term structure of the guards and performs e.g. the following transformations:
  - $\langle\neg(\phi_1 \wedge \phi_2)\rangle_1 = \langle\neg\phi_1\rangle_1 \vee \langle\neg\phi_2\rangle_1$
  - $\langle\neg(e'_1 > e'_2)\rangle_1 = (e'_1 \leq e'_2)$
  - $\langle\neg(e'_1 = e'_2)\rangle_1 = (e'_1 \neq e'_2)$

  Thus, it eliminates all cases of $\neg\phi$.

- The second step introduces the δ_m for the measurement variables and is performed by a function $\langle ... \rangle_2$ which is shown in Figure 4.

Consider as an example the guard $\neg\theta = 1000$. It is transformed into $\langle\langle\neg\theta = 1000\rangle_1\rangle_2 = \langle\theta \neq 1000\rangle_2 = \theta + \delta_\theta \neq 1000$ for a measurement variable θ.
The semantic interpretation of such a guard ϕ is done in the following way for measurement variables $m_1 ... m_n$:

$\exists \delta_{m_1} ... \delta_{m_n} . \phi$ with $-\Delta_{m_1} \leq \delta_{m_1} \leq \Delta_{m_1}$ ... $-\Delta_{m_n} \leq \delta_{m_n} \leq \Delta_{m_n}$.

Thus, in the above example, we have $\exists \delta_\theta . \theta + \delta_\theta \neq 1000$ with $-\Delta_\theta \leq \delta_\theta \leq \Delta_\theta$.

The existential quantification of the δ_m ensures that the robust guard function over approximates the corresponding non-robust guard function. We need the first transformation step eliminating the negations, because our methodology only works, if there are no negations in our guard: an existential quantification over a negated δ_m would result in an under approximation.

Note, that robust guards can be transformed back into the non-robust guard language while preserving their semantics. $\theta + \delta_\theta \neq 1000$ can be equivalently written as $1000 - \Delta_\theta > \theta \vee 1000 + \Delta_\theta < \theta$ by only using the constant Δ_θ. For practical applications, the computations performed in D-Finder can be done, using either the robust guards having been translated back into the non-robust guard language, or with slight modifications in D-Finder, by using the robust guard language directly.

A non-robust guard function can be constructed from a robust guard function by setting each δ_m to zero.

Definition 4.2 (Robust BIP Models)**.** *A BIP model is considered robust for a set of variables V_R iff all guard functions depending on a variable $m \in V_R$ are formalized in the robust guard language.*

Each guard can be substituted by a robust guard by replacing each occurrence of $m \in V_R$ by $m + \delta_m$ as shown above. We define a function $Robust_{V_R}$ for a set of measurement variables V_R to map BIP models to robust BIP models by replacing the guard functions by robust ones.

Theorem 4.1 (Reachable State Inclusion)**.** *For a robust BIP model BM_Δ and a BIP model BM with $BM_\Delta = Robust_{V_R}(BM)$, given a set of measurement variables V_R the following property holds :*
$ReachableStates_{BM} \subseteq ReachableStates_{BM_\Delta}$

Proof:
We have to show that the robust guard functions allow at least all state transitions that the non-robust guard functions allow. We do an induction on the term structure of ϕ to show:
$\langle\langle ... \rangle_1\rangle_2$ evaluates at least in all cases to true where $\delta_{m_1} ... \delta_{m_n}$ fixed to 0 would evaluate to true.

An example robust BIP model constructed from our temperature controller example (cp. Figure 3) for a measurement variable θ is shown in Figure 5.

Theorem 4.2 (Invariant Preservation)**.** *Each Invariant I of a robust BIP model BM_Δ is an invariant of a BIP model BM for a given set of measurement variables V_R if $BM_\Delta = Robust_{V_R}(BM)$*

Proof:
$\forall s \in ReachableStates_{BM_\Delta} . I(s)$ (since I is an invariant of BM_Δ) and
$ReachableStates_{BM} \subseteq ReachableStates_{BM_\Delta}$ (reachable states inclusion) implies
$\forall s \in ReachableStates_{BM} . I(s)$ $\square$

5 Application of Robust BIP Models

In this section we discuss the deadlock checking of robust BIP models with D-Finder and summarize and extend our comparison of invariants used in the process of certifying the deadlock freedom of a BIP model for usage with a higher-order theorem prover after D-Finder has provided its verdict.

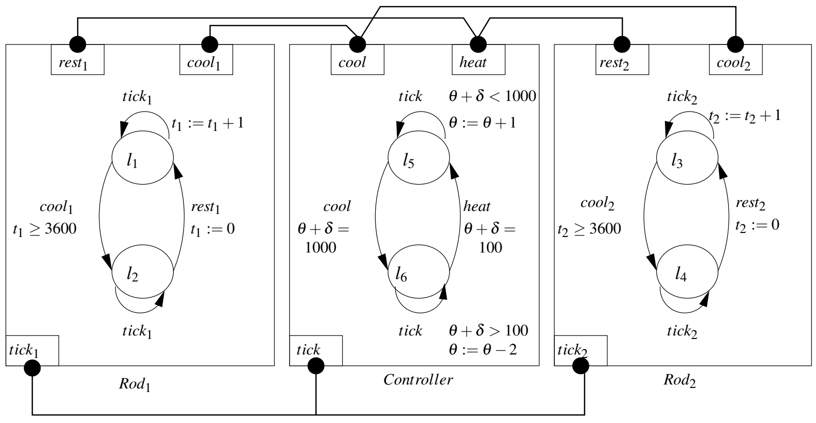


Figure 5: Robust Temperature Control System with Measurement Variable θ

$$\begin{tikzcd}
 & Enabled_{BM} \quad (deadlock\text{-}freedom) & \\
(\text{D-Finder})\ \Psi_{BM_\Delta} \arrow[ur, Rightarrow, "SMT\ solving"] & (\text{D-Finder})\ \Psi_{BM} \arrow[u, Rightarrow, "SMT\ solving"'] & \\
 & & \Psi_{BM} \wedge Guess \arrow[ul, Rightarrow, "trivial"'] \\
ReachableStates_{BM_\Delta} \arrow[uu, Rightarrow, "ind.\ proof(new\ proof\ strategy)"'] & & \\
 & & ReachableStates_{BM} \arrow[uu, Rightarrow, "ind.\ proof"'] \arrow[uuul, Rightarrow, dashed, "original\ proof\ strategy"] \arrow[ull, Rightarrow, "trivial", "BM \equiv (BM_\Delta \wedge \Delta{=}0)"']
\end{tikzcd}$$

weaker & likely inductive **non-inductive invariant** **stronger & inductive**

Figure 6: Relation Between Invariants

Invariants of Robust and Non-robust BIP models and D-Finder The relations between different invariants of BIP models *BM* and its robust counter-part BM_Δ is shown in Figure 6

Starting from the weakest inductive invariant (*true*), D-Finder computes a sequence of stronger invariants using the initial conditions and the provided BIP model *BM*. Due to abstractions performed by D-Finder this process can provide an invariant Ψ_{BM} that is not inductive. Using non-robust BIP models in our certification process the proof of the (dashed) implication can not be established for non-inductive invariants and the certificate generation fails. The critical abstraction used in D-Finder is in elimination of existential quantifiers in the logical formula that defines the successors of a state which becomes part of the provided invariant. D-Finder uses the Omega library for that step [Ome00]. A closer look at the original formula and its abstraction reveals the constraints that were lost during abstraction. The loss of precision is due to the fact that (1) some facts are implicit (*e.g.*, for a variable x, the fact $x \geq 0$ is eliminated if x is of type `nat`), or (2) facts cannot be represented in the logic (this is the case for divisibility constraints in the Omega library). The addition of these lost constraints leads to an inductive invariant. The divisibility constraints can be useful for variables of the program that range over discrete domains (*e.g.*, counters). However it produces inductive but unrealistic invariants for variables that represent physical measurements provided by sensors.

Furthermore, Figure 6 shows the goals sketched in the introduction and in Section 3 on invariants. In order to conduct the proof we seek for an inductive invariant $\Psi_{Inductive}$ that contains the (non-computable) sets of reachable states and that entails the deadlock-freedom property ($Enabled_{BM}$), such that the following implications holds:

$$ReachableStates_{BM} \Rightarrow \Psi_{Inductive} \qquad\qquad \Psi_{Inductive} \Rightarrow Enabled_{BM}$$

To build $\Psi_{Inductive}$ we can try to strengthen the invariant provided by D-Finder by guessing a suitable constraint such that $\Psi_{BM} \wedge Guess$ is inductive. The certificate then encapsulates the proof of the right-most chain of implications. Otherwise we can try the approach promoted in this paper for building an inductive invariant weaker than Ψ_{BM} but still strong enough to entail deadlock-freedom. This approach comprises the Ψ_{BM_Δ} invariant of the robust BIP model and corresponds to the left-most chain of implications of Figure 6.

Each approach can lead to a certificate based on a proof by induction which can be automatically generated and then provided to a higher-order theorem prover for checking.

Evaluation In contrast to non-robust BIP models, D-Finder produces inductive invariants of robust BIP models in the case studies that we examined so far. The model of Figure 5 is the robust version of our running example of Figure 3. In all guards the uncertainty δ is added to the θ variable that corresponds to the measure of the temperature. This transformation prevents the generation of over constrained invariants. Obviously, the generated invariants are less precise than those obtained for the original model. Hopefully, in our experimentations this did not introduce new deadlock possibilities. Actually, it is highly desirable that the deadlock-freedom property of a system does not depend on the sensitivity of sensors.

Figure 7 summarizes our approach using the running example. It shows the invariant CI_3 generated by D-Finder on the original BIP model BM (in the middle) compared to the corresponding stronger invariant obtained by strengthening (on the left) and the weaker but robust invariant generated by D-Finder from the robust model BM_Δ (on the right).

The CI_3 invariant relates the value of θ to the location of the controller state machine. The original invariant is not inductive. It can be either strengthened by adding the additional constraint, $2|\theta$. This leads to an inductive but unrealistic invariant. A more realistic invariant is obtained by running D-Finder on the robust model. The resulting invariant is inductive and strong enough to conduct the last step of the deadlock analysis. The final result is as precise as the ones obtained with the two other invariants; no false deadlocks are generated by robust invariants.

Not all invariants of robust BIP models have to be necessarily inductive. It may be possible that we need to strengthen them sometimes e.g. by using the technique discussed in [BM08]. These techniques, however, must not add constraints bearing unnatural facts on the measuring process of the measurement variables.

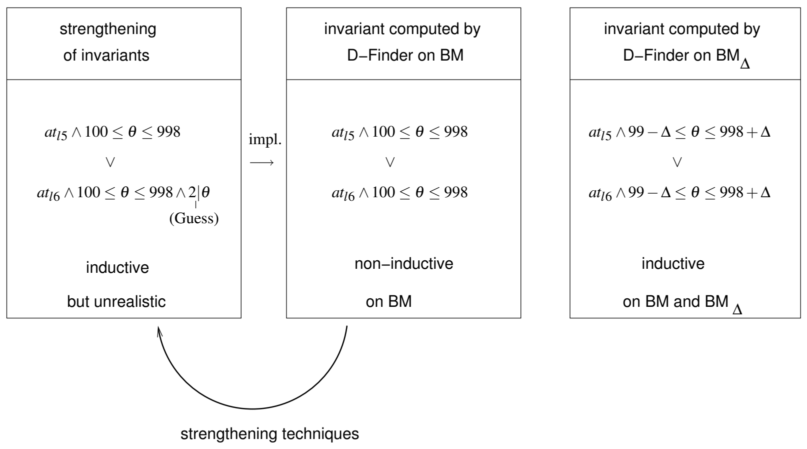

strengthening of invariants
$at_{l5} \wedge 100 \leq \theta \leq 998 \vee at_{l6} \wedge 100 \leq \theta \leq 998 \wedge 2|\theta$
(Guess)
inductive but unrealistic
impl.
invariant computed by D−Finder on BM
$at_{l5} \wedge 100 \leq \theta \leq 998 \vee at_{l6} \wedge 100 \leq \theta \leq 998$
non−inductive on BM
invariant computed by D−Finder on BM$_\Delta$
$at_{l5} \wedge 99-\Delta \leq \theta \leq 998+\Delta \vee at_{l6} \wedge 99-\Delta \leq \theta \leq 998+\Delta$
inductive on BM and BM$_\Delta$
strengthening techniques

Figure 7: Handling the Example Invariant

6 Related Work

This paper describes the modification of systems in order to generate realistic inductive invariants for a verification tool. These invariants are verified to hold by a higher-order theorem prover for certifying the results of the verification tool. Inductiveness is crucial for deductive proofs in the certification process. To our knowledge this particular subject has not yet been studied before.

Certification The certification of analysis results is an important aspect of this work. We generate certificates in the form of proof-scripts for a higher-order theorem prover. The generation of proofs to certify the verdict of a model-checker was first introduced in [Nam01]. Other important work for certifying the results of verification tools comprise the use of support sets for a model checker [TC02], keeping track of justifications for the BLAST model-checker [HJM+02], and a SAT solver that generates certificates [ZM03]. Further related is Proof-Carrying Code (PCC) [Nec97], a method to guarantee that executable code fulfills a policy on access and resource management and Foundational PCC [WAS03] characterized by a small set of axioms and a simpler proof-checker. In this paper we concentrate on certifying invariants via a formal proof done by induction.

Generation of Inductive Invariants Automatic generation of invariants has been studied for a long time (see e.g. [MP95]). For the invariants considered in this paper the papers [BLS96] and [SDB96] where most influential. [BLS96] describes many principles that have been implemented in our deadlock-verification tool D-Finder.

[BM08] focuses on techniques to incrementally generate inductive invariants. The main idea is to use counter-examples to refine an invariant until it becomes provable by induction. The paper features in addition to the description of general techniques, further valuable techniques to refine invariants by splitting large conjunctions into subparts and refine selected subparts independently. This method has been successfully used in the refinement of boolean invariants. The presented techniques require a deep knowledge of the class of systems in consideration, in order to cleverly take advantage of the presented counter-example guided refinement approach of invariants.

In this paper we regard strengthening of non-inductive invariants by taking advantage of our knowledge of the verification tool D-Finder – we know when the invariant can lose their inductiveness quality – as one possibility to achieve inductive invariants.

Robustness Another feature of this paper is robustness. Robustness of timed automata is described in [GHJ97]. This notion is similar to our notion and has been introduced for real-time systems in order to cope with properties that e.g. occur when transforming continuous signals to discrete values or problems due to imprecision of sensors. The theory of verification of systems is enriched with a so called tube-semantics to capture uncertainty. Similarly, [AM95] introduces the notion of finite variability for properties of continuous systems to capture semantics intervals of continuous time. The presented method allows proving properties of a continuous semantics by reasoning on a discrete semantics which is more appropriate for automated verification and deductive reasoning. A constrained solving based method for generating inductive invariants for hybrid systems is presented in [SSM04].

In contrast to these work, we do not consider real-time system with continuous semantics. The BIP system has a discrete semantics and acquires information about the physical quantities through sensors. One of our our goals is introducing uncertainty about the sensors measurements while reusing D-Finder invariant generation techniques based on a discrete semantics.

7 Conclusion and Future Work

In this paper we have introduced the notion of robust BIP models for systems containing values representing the results of physical measurements. Robust BIP models can be obtained from non-robust BIP models and over approximate their behavior. We proved that each invariant of the robust BIP model is also an invariant of the corresponding non-robust BIP model. We motivated that invariants that have been automatically generated by the deadlock detection tool D-Finder for robust BIP models tend to be inductive while those of non-robust systems are likely to be non-inductive. Inductivity of invariants allows for easy formal verification that they do indeed hold in a higher-order theorem prover. Our technique is developed for certifying the results of D-Finder at runtime. It allows to keep a discrete representation instead of a continuous representation of values occurring in a system, thus enabling the use of tools like D-Finder which work on discrete representations.

Future work continuing this on-going work involves the analysis of further case studies to discover more benefits and potential limitations of our approach. We have achieved first results in producing realistic and inductive invariants by running D-Finder on robust BIP models. It seems that instead of using strengthening techniques, we can use robust (and therefore weaker) constraints to achieve inductive invariants. Once D-Finder provides analyzes for BIP models with value exchange during component interaction, we will model sensors and provided values as distinct components. Future work includes further reasoning on robustness to ensure that inductiveness of D-Finder invariants is guaranteed by the construction process of robust BIP models. More technical challenges comprise the addition of ranges of unpreciseness that can deal with non-linear errors such as a certain error percentage of a measured value. This can be modeled as functions depending on measured values and have to be integrated in our guard language. These functions are difficult to handle since the Yices SMT-solver can only deal with invariants generated from guard functions formalized in Presburger arithmetic. Another task reserved to future work is fixing invariants by adding constraints in order to make them inductive. Such constraints might be generated by using hints from unresolved theorem prover goals created during the process of trying to prove an invariant correct.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AM95] L. De Alfaro, Z. Manna. Verification in Continuous Time by Discrete Reasoning. Proceedings of the 4-th International Conference on Algebraic Methodology and Software Technology AMAST'95, LNCS, Springer-Verlag, 1995.

[BBSN08] S. Bensalem, M. Bozga, J. Sifakis, and T-H. Nguyen. Compositional Verification for Component-based Systems and Application. ATVA 2008 6th International Symposium on Automated Technology for Verification and Analysis, October 20-23, 2008, Seoul, South Korea.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[BLS96] S. Bensalem, Y. Lakhnech, H. Saidi. Powerful techniques for the automatic generation of invariants. CAV'96, Springer-Verlag, 1996.

[BM08] A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. Formal Aspects of Computing, Springer-Verlag 2008.

[BP08] J. O. Blech and M. Périn. Towards certifying deadlock-freedom for BIP models. Technical Report TR-2008-1, Verimag, September 2008.

[BP08a] J. O. Blech and M. Périn. On Certificate Generation and Checking for Deadlock-freedom of BIP Models. Technical Report TR-2008-19, Verimag, December 2008.

[DM06] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). CAV'06. Volume 4144 of LNCS, Springer-Verlag 2006.

[GHJ97] V. Gupta, T. A. Henzinger, R. Jagadeesan. Robust Timed Automata Proceedings of the International Workshop on Hybrid and Real-Time Systems, LNCS, Springer-Verlag, 1997.

[HJM+02] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. *Proc of CAV '02*, 2002. Springer-Verlag.

[Nam01] K. S. Namjoshi. Certifying model checkers. *Proc of CAV '01*, 2001. Springer-Verlag.

[Nec97] G. C. Necula. Proof-carrying code. In *POPL '97*, pages 106–119, New York, NY, USA, 1997. ACM.

[MP95] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, 1995.

[Ome00] The Omega library. Version 1.2 (August 2000)

[SDB96] J. X. Su, D. L. Dill, C. W. Barrett. Automatic Generation of Invariants in Processor Verification. FM-CAD'96, 1996.

[SSM04] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constructing Invariants for Hybrid Systems. In Hybrid Systems: Computation and Control (HSCC 2004), LNCS, Springer-Verlag 2004.

[TC02] L. Tan and R. Cleaveland. Evidence-based model checking. *Proc of CAV '02*, London, UK, 2002. Springer-Verlag.

[WAS03] D. Wu, A.W. Appel, and A. Stump. Foundational proof checkers with small witnesses. *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, 2003.

[ZM03] L. Zhang and S. Malik. Validating SAT Solvers Using Independent Resolution-Based Checker: Practical Implementations and Other Applications DATE 2003.

Discovering Specifications for Unknown Procedures - Work in Progress -

Florin Craciun[1] Chenguang Luo[1] Guanhua He[1] Shengchao Qin[1] Wei-Ngan Chin[2]
[1] Department of Computer Science, Durham University, U.K.
{florin.craciun, chenguang.luo, guanhua.he, shengchao.qin}@durham.ac.uk
[2] School of Computing, National University of Singapore
chinwn@comp.nus.edu.sg

Abstract

We study automated verification of pointer safety for heap-manipulating imperative programs with unknown procedure calls or code pointers. Given the specification of a procedure whose body contains calls to an unknown procedure, we try to infer the possible specifications for the unknown procedure from its calling contexts. We employ a forward shape analysis with separation logic and an abductive inference mechanism to synthesize both pre- and postconditions for the unknown procedure. The inferred specification is a partial specification of the unknown procedure. Therefore it is subject to a later verification when the code or the complete specification for the unknown procedure are available. Our inferred specifications can also be used for program understanding.

1 Introduction

A program verification system takes as input a given program and a specification of its desired properties and attempts to determine whether or not the program meets the specified properties. The verifier usually requires to have access to the entire given program which, in practice, may not be completely available for various reasons. In general the unknown program fragments can correspond to unknown procedure calls which might be either calls to library procedures without source codes or code pointers with their corresponding meta-programming features like run-time generation/loading of code.

To deal with the verification of programs with unknown procedure calls, current program verifiers either

- stop at the first unknown procedure call and provide an incomplete verification, or
- simply ignore the unknown procedure calls (e.g. replace them by skip) or treat them as nondeterministic assignments without side effects (both methods can be unsound in general) ([4]), or
- assume the variables that are modified by the unknown procedure call and its caller do not overlap (so they have different so-called *footprints* and the hypothetical frame rule [15] can be applied; however this assumption does not hold in most cases), or
- use specification mining [1] to discover possible specifications for the program (which is performed dynamically and is just a check instead of a proof for program correctness), or
- ask for a description of the unknown procedure properties, which might be obtained by analyzing their binary code ([2, 8, 12, 10]). In the case of code pointers it is necessary to have an analysis that can estimate all possible procedures to which the pointers may refer at run-time. In general this estimation may return too many candidates, making the verification almost impossible at compile time.

In this paper we propose a different approach to the verification of programs with unknown procedure calls. Based on the desired properties specified for the entire program, our solution is to verify the known program fragments, and to postpone the verification of the unknown procedures by inferring partial specifications of their desired properties. The inferred specifications are subject to a later verification when the code or the complete specification of the unknown procedure become known (e.g. at loading time in Java). Our inferred specifications can also be used for program understanding to build partial specifications of the generic procedures (e.g. the methods of Java interfaces). The inferred specifications are derived from the calling contexts. Therefore, in the case of multiple specifications for the same unknown procedure they have to be combined together in order to get a more general (stronger) specification. Even though our approach can not discover the complete specification of the unknown procedure, what we can discover can be used to completely describe the unknown procedure behaviour inside the analyzed module (e.g. the behaviour of the implementation of method `compareTo` of interface `Comparable` inside the current analyzed Java package, `JavaCup`).

We use our approach to verifying the pointer safety of heap-manipulating programs with unknown procedure calls. Our framework is built on the work on shape analysis with separation logic [16, 7, 13]. Program specifications are given as pre and post conditions for each known procedure. We perform a modular verification on a per procedure basis as in the HIP system [13]. Starting with the given precondition, a forward verification is run until the unknown procedure call is met. A precondition for the unknown procedure is derived from the current heap state ([11]). Thus the current heap is split into two disjoint heaps: (1) the unknown procedure's precondition consisting of all of the current heap nodes reachable from the unknown procedure's actual parameters, and (2) a frame. The frame is used as initial state for the forward verification of any program code that follows the unknown procedure call. The verification might find situations where there is not enough information to perform an operation such as a (known) procedure call or a dereferencing. If the missing information refers to the actual arguments or to the result of the unknown procedure call then we perform a similar abductive inference as that in Calcagno et al. [3, 4] to infer what is missing. The inferred information will be part of the unknown procedure postcondition. If the missing information does not refer to the unknown procedure then there is an error in the code.

Our proposal is mainly inspired by the recent work [4] that uses bi-abduction to infer the pre/post conditions of a given procedure by analysing its body. It uses a bottom up approach such that the procedures called in the body of the given procedure are known and their pre/post conditions were inferred prior to the current procedure. The problem we solve in this paper is dual to that in Calcagno et al. [4]. Given the pre/post conditions of a procedure whose body invokes an unknown procedure, we attempt to infer the pre/post conditions of the unknown procedure. A similar problem to ours has been solved in the context of logic programs [9], where, based on the specification and the implementation of a logic programming module, an abductive method infers constraints on undefined literals of that module.

Our proposal can also be regarded as a top-down inference system rather than a bottom-up one [4] that most traditional approaches rely on. In our case the user provides the specification for the top-level procedure and our approach can infer the specifications for the called procedures. This process may be repeated at several levels down. However the inferred specifications may not be complete, therefore they are subjected to a later verification.

A verification system uses an entailment procedure to check whether the program heap state $\Delta_{\mathtt{i}}$ computed at the program instruction $\mathtt{i}$ entails the precondition $\mathsf{Pre}_{\mathtt{i}+1}$ (namely the requirements for a correct execution) of the next program instruction $\mathtt{i}+1$ as follows:

$$\Delta_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i}+1}$$

In general not the entire current program state $\Delta_{\mathtt{i}}$ is required to establish $\mathsf{Pre}_{\mathtt{i}+1}$. Therefore a more general form of the entailment is used:

$$\Delta_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i}+1} * \mathsf{R}_{\mathtt{i}}$$

where $*$ stands for the separation conjunction [16], and $\mathsf{R}_{\mathtt{i}}$ is the frame, namely the part of the heap state $\Delta_{\mathtt{i}}$ which is not accessed by the program instruction $\mathtt{i}+1$. When the entailment fails

$$\Delta_{\mathtt{i}} \nvdash \mathsf{Pre}_{\mathtt{i}+1}$$

we can use a reasoning with abduction

$$\Delta_{\mathtt{i}} * [\mathsf{M}_{\mathtt{i}}] \triangleright \mathsf{Pre}_{\mathtt{i+1}}$$

to find the missing part $\mathsf{M}_{\mathtt{i}}$ of the heap state $\Delta_{\mathtt{i}}$ such that

$$\Delta_{\mathtt{i}} * \mathsf{M}_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i+1}} * \mathsf{R}_{\mathtt{i}}$$

In principle the abduction strengthens the current heap state $\Delta_{\mathtt{i}}$. Our approach mainly uses the abduction to discover the postconditions of the unknown procedures. However a naive application of the abduction may generate too strong postconditions for the unknown procedures, in which cases acceptable implementations of the unknown procedures can be rejected as being incorrect. Therefore, in Section 2 we define the property that has to be satisfied by an abductive system in order to be useful for our approach.

The remainder of the paper is organized as follows. Section 2 introduces the key features of our approach by a simple example that consists of one unknown procedure call. Section 3 extends our approach to multiple sequential calls of the same unknown procedure. A brief conclusion is then given at the end.

2 Discovering a Specification for an Unknown Procedure

In this section we illustrate how our analysis computes a precondition and a postcondition for an unknown procedure. Before that, we introduce the abstract domain that we will use, which is similar to the one exploited in Calcagno et al. [4].

We have two disjoint sets of variables. One is a finite set of program variables *Var* (denoted by $\mathtt{x}, \mathtt{y}, \mathtt{z}, ...$) and a countable set of logical variables *LVar* (expressed by $\mathtt{x}', \mathtt{y}', ...$). The logical variables are never used in programs; they are for recording some historical values of program variables. Hence they may be existentially quantified at any point. *Loc* is a countably infinite set of locations, and *Val* is a set of values that includes *Loc*. Our *storage model* is a traditional separation logic one:

$$\begin{array}{lll} Heap & =_{df} & Loc \rightharpoonup Val \\ Stack & =_{df} & (Var \cup LVar) \to Val \\ State & =_{df} & Stack \times Heap \end{array}$$

We regard *symbolic heaps* as special separation logic formulae to represent the abstraction of a set of concrete heaps. They are defined as follows:

$$\begin{array}{llll} E & ::= & \mathtt{x} \mid \mathtt{x}' & \textit{Expressions} \\ \Pi & ::= & E = E \mid E \neq E \mid \mathsf{true} \mid \Pi \wedge \Pi & \textit{Pure formulae} \\ \mathsf{B} & ::= & E \mapsto E \mid \mathtt{list}(E, E) & \textit{Basic separation predicates} \\ \Sigma & ::= & \mathsf{B} \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Separation formulae} \\ \Delta & ::= & \Pi \wedge \Sigma & \textit{Symbolic heaps} \end{array}$$

Expressions can be both kinds of variables. Pure formulae are composed by the conjunction of (dis) equalities between expressions to show (non) alias relationship between references. The basic separation predicates describe simple points-to or list segments, which will be introduced later. Separation formulae stand for basic shape predicate, or empty heap, or a separation conjunction of them. A symbolic heap consists of both pure part and separation (shape) part. We overload the separation conjunction over Δ_1 and Δ_2 to conjoin their shape and pure parts, respectively.

Based on the abstract domain defined, consider the procedure `findLast` whose specification and implementation are shown in Figure 1. The procedure searches for the last (non-null) element of a singly

```
// Given Specification:
       // $\mathsf{Pre}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
       // $\mathsf{Post}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null}$
node findLast(node x) {
0    node w, y, z;
0a    // $\Delta_0 := \mathsf{Pre}_{\mathtt{findLast}}$
0b    // $\Delta_0 \vdash \mathtt{x}{\neq}\mathtt{null}$
1    w := x.next;
1a    // $\Delta_1 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
2    if (w == null) return x;
2a    // $\Delta_2 := \mathtt{w}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{x}{\mapsto}\mathtt{null} \wedge \mathtt{x}{\neq}\mathtt{null}$
2b    // $\exists \mathtt{w},\mathtt{y},\mathtt{z} \cdot \Delta_2 \vdash \mathsf{Post}_{\mathtt{findLast}}$
3    else {
3a    // $\Delta_3 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3b    // $\mathsf{Local}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3c    // $\mathsf{Frame}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3d    // $\mathsf{Pre}_{\mathtt{unkProc}} := \mathsf{Local}(\Delta_3,\{\mathtt{x}\})$
4    y := unkProc(x);
4a    // $\Delta_4 := (\exists \mathtt{fv}(\mathsf{Pre}_{\mathtt{unkProc}}) \cdot \mathsf{Frame}(\Delta_3,\{\mathtt{x}\})) * (\mathsf{emp} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{=}\mathtt{x})$
4b    // $\mathsf{M} := (\mathtt{res}_{\mathtt{unkProc}}{=}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{=}\mathtt{x})$
4c    // $\Delta_4 \nvdash [\mathtt{y}/\mathtt{x}]\ \mathsf{Pre}_{\mathtt{findLast}}$
4d    // $\Delta_4 * [\mathsf{M}_1] \triangleright [\mathtt{y}/\mathtt{x}]\ \mathsf{Pre}_{\mathtt{findLast}}$       $\mathsf{M}_1 := \mathsf{list}(\mathtt{y},\mathtt{null}) \wedge \mathtt{y}{\neq}\mathtt{null}$
4e    // $\mathtt{fv}(\mathsf{M}_1) \subseteq \mathtt{ReachVar}(\Delta_4 * \mathsf{M}_1, \{\mathtt{x}',\mathtt{y}'\})$       $\mathsf{M} := \mathsf{M} * \mathsf{M}_1$
4f    // $\Delta_4 * \mathsf{M}_1 \vdash [\mathtt{y}/\mathtt{x}]\ \mathsf{Pre}_{\mathtt{findLast}} * \mathsf{R}_1$       $\mathsf{R}_1 := \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{=}\mathtt{x}$
5    z := findLast(y);
5a    // $\Delta_5 := \mathsf{R}_1 * [\mathtt{y}/\mathtt{x},\mathtt{z}/\mathtt{res}]\ \mathsf{Post}_{\mathtt{findLast}}$
6    return z;
6a    // $\Delta_6 := \Delta_5 \wedge \mathtt{res}{=}\mathtt{z}$
6b    // $(\exists \mathtt{z},\mathtt{y},\mathtt{y}',\mathtt{x}' \cdot \Delta_6) \nvdash \mathsf{Post}_{\mathtt{findLast}}$
6c    // $(\exists \mathtt{z},\mathtt{y},\mathtt{y}',\mathtt{x}' \cdot \Delta_6) * [\mathsf{M}_2] \triangleright \mathsf{Post}_{\mathtt{findLast}}$       $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{y})$
6d    // $\mathtt{fv}(\mathsf{M}_2) \subseteq \mathtt{ReachVar}(\Delta_6 * \mathsf{M}_2, \{\mathtt{x}',\mathtt{y}'\})$       $\mathsf{M} := \mathsf{M} * \mathsf{M}_2$
7    } }
8 // $\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{fv}(\mathsf{M}) \setminus (\mathtt{fv}(\mathsf{Pre}_{\mathtt{unkProc}}) \cup \{\mathtt{res}_{\mathtt{unkProc}}\}) \cdot \mathsf{M}$
```

Figure 1: Procedure `findLast` calling an unknown procedure `unkProc`.

linked list. At line 4 it calls the unknown procedure unkProc. We use an imperative language with references. The data structure `node { int val; node next }` defines a list element.

We run a standard forward shape analysis based on separation logic. The results of our analysis are marked as comments in the code. A list segment is described inductively by the following separation predicate:

$$\mathsf{list}(\mathtt{x},\mathtt{y}) \ =_{df}\ (\mathtt{x}{=}\mathtt{y}\wedge \mathsf{emp})\vee(\mathtt{x}{\mapsto}\mathtt{y})\vee(\exists \mathtt{z}\cdot \mathtt{x}{\mapsto}\mathtt{z} * \mathsf{list}(\mathtt{z},\mathtt{y})\wedge \mathtt{z}\neq \mathtt{y})$$

where the points-to predicate $\mathtt{x}{\mapsto}\mathtt{y}$ denotes a heap with a single allocated node at address x with its next field pointing to the address y.

The analysis starts with the procedure findLast precondition (line 0a). The verification is straightforward until just before the unknown procedure call. At line 3, the current heap Δ_3 is split into two disjoint heaps: the local heap $\mathsf{Local}(\Delta_3,\{\mathtt{x}\})$ to be sent to the unknown procedure and the frame heap $\mathsf{Frame}(\Delta_3,\{\mathtt{x}\})$ that is not accessed by the unknown procedure. The local heap is obtained by taking the part of the current heap Δ_3 reachable in the formula from the actual parameter x of the unknown procedure.

In general, at a call site $\mathtt{f}(\mathtt{x}_1,..,\mathtt{x}_n)$ the current heap state Δ can be expressed as $\Delta=\Pi\wedge\Sigma$, where Π is a pure formula insensitive to heap (mainly consisting of aliasing information) and Σ expresses heap shape (the shape of linked data structures located on the heap). The part of the heap to be sent to the procedure is denoted by $\mathsf{Local}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$ and is defined as follows:

$$\mathsf{Local}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})=\exists \mathtt{fv}(\Pi\wedge\Sigma)\setminus\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})\cdot\Pi*\mathtt{ReachHeap}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$$

where $\mathtt{fv}(\Delta)$ stands for all the free (program and logical) variables occurring in Δ. And the frame, namely the part of the heap that is not accessed by the procedure call is defined as follows:

$$\mathsf{Frame}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})=\Pi\wedge\mathtt{UnreachHeap}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$$

where $\mathtt{UnreachHeap}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$ is the formula consisting of all $*$-conjuncts from Σ which are not in $\mathtt{ReachHeap}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$. The formula $\mathtt{ReachHeap}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$ denotes the part of Σ reachable from $\{\mathtt{x}_1,..,\mathtt{x}_n\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\Sigma_1\mid \exists \mathtt{z}_1,\mathtt{z}_2,\Sigma_2\cdot \mathtt{z}_1\in\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})\wedge\Sigma=\Sigma_1*\Sigma_2\wedge\Sigma_1=\mathsf{B}(\mathtt{z}_1,\mathtt{z}_2)\}$$

$\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})$ is the minimal set of variables which are aliases or are reachable through the heap such that:

$$\begin{array}{l}\{\mathtt{x}_1,..,\mathtt{x}_n\}\cup\{\mathtt{z}_2\mid\exists \mathtt{z}_1,\Pi_1\cdot \mathtt{z}_1\in\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})\wedge\Pi=(\mathtt{z}_1=\mathtt{z}_2)\wedge\Pi_1\}\cup\\ \{\mathtt{z}_2\mid\exists \mathtt{z}_1,\Sigma_1\cdot \mathtt{z}_1\in\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})\wedge\Sigma=\mathsf{B}(\mathtt{z}_1,\mathtt{z}_2)*\Sigma_1\}\subseteq\mathtt{ReachVar}(\Pi\wedge\Sigma,\{\mathtt{x}_1,..,\mathtt{x}_n\})\end{array}$$

Note that $\mathsf{B}(\mathtt{z}_1,\mathtt{z}_2)$ stands for either $\mathtt{z}_1{\mapsto}\mathtt{z}_2$ or $\mathsf{list}(\mathtt{z}_1,\mathtt{z}_2)$.

Thus the precondition of the unknown procedure unkProc is set to the local heap $\mathsf{Local}(\Delta_3,\{\mathtt{x}\})$ as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}}:=\mathtt{x}{\mapsto}\mathtt{w}*\mathsf{list}(\mathtt{w},\mathtt{null})\wedge\mathtt{x}{\neq}\mathtt{null}\wedge\mathtt{w}{\neq}\mathtt{null}$$

The above precondition is a safe estimation of that part of the calling context heap that may be accessed by the unknown procedure unkProc. Therefore it is not one of the weakest possible preconditions. Our computed precondition may also contain cutpoints. In our example the cutpoint is w. As can be seen below, this cutpoint does not occur in the postcondition (II) of the procedure unkProc. Therefore it can be existentially quantified as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}}:=\exists\mathtt{w}\cdot(\mathtt{x}{\mapsto}\mathtt{w}*\mathsf{list}(\mathtt{w},\mathtt{null})\wedge\mathtt{x}{\neq}\mathtt{null}\wedge\mathtt{w}{\neq}\mathtt{null}) \qquad (\mathtt{I}')$$

Next we perform a weakening over the above formula $(\mathtt{I}')$. Generally it is unsound to weaken a computed precondition in verification; however, we can abstract $(\mathtt{I}')$ to a simpler one for two reasons. First, we inferred this precondition from the calling context (rather than the procedure body). Second, for a possible later verification against the procedure body, this weakening does not allow any incorrect implementation to pass. Therefore we get the weakened result as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \qquad (\mathtt{I})$$

However the precondition $(\mathtt{I}')$ requires a list with at least two nodes while the precondition $(\mathtt{I})$ requires a list with at least one node. Thus the precondition $(\mathtt{I})$ may reject some possible correct implementations of the unknown procedure unkProc. Therefore, to maintain necessary precision, it is not always desired to weaken the precondition inferred from the calling context.

The unknown procedure call is at line 4. In general the heap state after a procedure call consists of two disjoint parts: the frame (the heap part that is not changed by the procedure call) and the procedure postcondition (the heap part that is changed by the procedure call). In our case the frame is known (the first part of state Δ_4 at line 4a), but the procedure postcondition is unknown. What we know is that the unknown procedure unkProc may change the heap reachable from the program variables x and y. Therefore initially we set the unknown procedure postcondition to $\mathsf{emp} \wedge \mathtt{y}{=}\mathtt{y}' \wedge \mathtt{x}{=}\mathtt{x}'$ (the second part of state Δ_4 at line 4a). We use the logical variables $\mathtt{x}'$ and $\mathtt{y}'$ to denote respectively the values of the program variables x and y when the unknown procedure call returns. Our goal is to discover the postcondition, namely the heap that can be reached from $\mathtt{x}'$ and $\mathtt{y}'$ by analysing the usage of $\mathtt{x}'$ and $\mathtt{y}'$ in the remaining code after the unknown procedure call. Therefore after the unknown procedure call our analysis keeps track of a pair $\langle \Delta_\mathtt{i};\ \mathsf{M}\rangle$ where $\Delta_\mathtt{i}$ is the current heap state, while M denotes the postcondition discovered so far for the unknown procedure. The notations $\mathsf{M}_\mathtt{i}$ are also used to denote parts of the discovered postcondition. Thus the first discovered information M is that just after the unknown call the value of the program variable y and the unknown procedure result are equal (line 4b). M also contains the information that the logical variables $\mathtt{x}'$ and $\mathtt{y}'$ denote the values of the program variables x and y respectively, just after the unknown procedure call.

At line 5 the procedure findLast is called recursively. Since the current heap state Δ_4 does not entail the precondition of the procedure findLast (line 4c) there is an error. However this error may not be a program error, it may be caused by the incomplete heap denoted by $\mathtt{x}'$ and/or $\mathtt{y}'$. Therefore our analysis performs an abductive reasoning (line 4d) to infer the missing part M_1 of Δ_4 such that $\mathsf{M}_1 * \Delta_4$ entails the precondition of the procedure findLast. At line 4e our analysis checks whether the inferred M_1 is part of the unknown procedure postcondition, namely whether M_1 refers either to $\mathtt{x}'$ and/or $\mathtt{y}'$ or to aliases of $\mathtt{x}'$ and/or $\mathtt{y}'$ or to a heap reachable from $\mathtt{x}'$ and/or $\mathtt{y}'$. This check succeeds and therefore M_1 is added to the current discovered postcondition M.

The heap state Δ_4 combined with the inferred M_1 entails the precondition of the procedure findLast and also generates a residual frame heap $\mathtt{R}_1$ (line 4f). The heap state Δ_5 after the recursive call consists of the procedure findLast postcondition and the frame heap $\mathtt{R}_1$ (line 5a).

At the end of the procedure body the current heap state Δ_6 (computed at line 6a) must entail the postcondition of the procedure findLast. This entailment fails (line 6b). We perform another abductive reasoning (line 6c) to infer the missing M_2 as follows:

$$(\exists \mathsf{z},\mathsf{y},\mathsf{y}',\mathsf{x}' \cdot \mathsf{list}(\mathsf{y},\mathsf{z}) * \mathsf{z}{\mapsto}\mathtt{null} \wedge \mathtt{res}{=}\mathsf{z} \wedge \mathsf{y}{=}\mathsf{y}' \wedge \mathsf{x}{=}\mathsf{x}') * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null}$$

where Δ_6 and $\mathsf{Post}_{\mathtt{findLast}}$ are replaced by their formulae. The above judgment can be further simplified as follows:

$$\mathsf{list}(\mathtt{y},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null} * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null}$$

and then to the following:

$$\mathsf{list}(\mathtt{y},\mathtt{res}) * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x},\mathtt{res})$$

Using the abductive inference rules from Calcagno et al. [4] we can obtain the following result $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{res})$ that passes the check from line 6d. Using formula from line 8 we obtain the following postcondition for the unknown procedure (where $\mathtt{u}$ stands for α-renaming of $\mathtt{res}$):

$$\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x},\mathtt{u}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}},\mathtt{null}) \wedge \mathtt{res}_{\mathtt{unkProc}} \neq \mathtt{null} \quad (\mathtt{II}')$$

The above postcondition $(\mathtt{II}')$ and the precondition $(\mathtt{I})$ might form a candidate specification for the unknown procedure unkProc. However, the postcondition $(\mathtt{II}')$ is not strong enough to establish the postcondition of the outer procedure findLast. This problem is due to incompleteness of the abductive inference system from Calcagno et al. [4].

It is very difficult to impose the completeness for an abductive inference system. Therefore we define the following weaker property that has to be satisfied by the abductive system. In general, given an abductive judgment:

$$\Delta * [\mathsf{M}] \triangleright \mathsf{H}$$

there are many possible solutions for M. In order to be able to compute the most precise postcondition we are interested in the weakest solution M (namely minimal solution w.r.t. the order $\preceq$ defined in Calcagno et al. [4]) satisfying the following entailment

$$\Delta * [\mathsf{M}] \vdash \mathsf{H} * \mathsf{R} \quad (\mathtt{III})$$

such that R does not refer to the arguments and result of the unknown call, or to the aliases of the arguments and result of the unknown call, or to the heap reachable from the arguments and result of the unknown call. Using the notations of our example with an unknown call having the argument $\mathtt{x}'$ and the result $\mathtt{y}'$ the property of R can be expressed as follows:

$$\mathtt{fv}(\mathsf{R}) \not\subseteq \mathtt{ReachVar}(\Delta * \mathsf{M}, \{\mathtt{x}',\mathtt{y}'\}) \quad (\mathtt{IV})$$

The computation of a solution that satisfies the properties $(\mathtt{III})$ and $(\mathtt{IV})$ depends on the abductive inference rules which implement the abductive judgment. However the existence of this solution does not imply the completeness of the abductive inference system.

In our case for the following abductive judgment:

$$\mathsf{list}(\mathtt{y},\mathtt{res}) * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x},\mathtt{res})$$

there are three possible solutions $\mathsf{M}_2 := \mathtt{x}{=}\mathtt{y}$, $\mathsf{M}_2 := \mathtt{x} {\mapsto} \mathtt{y}$, and $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{y})$ which satisfy the above properties $(\mathtt{III})$ and $(\mathtt{IV})$. The best solution is $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{y})$ which is weaker than the other two solutions. However the current abductive inference rules from Calcagno et al. [4] are not able to infer these solutions. Therefore those rules have to be refined such that they can also support matching on the right sides of two separation predicates when their left sides are equal. Using the best solution $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{y})$ we can obtain a more precise postcondition for the unknown procedure as follows:

$$\mathsf{Post}_{\mathtt{unkProc}} := \mathsf{list}(\mathtt{x},\mathtt{res}_{\mathtt{unkProc}}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}},\mathtt{null}) \wedge \mathtt{res}_{\mathtt{unkProc}} \neq \mathtt{null} \quad (\mathtt{II})$$

where the best solution $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathtt{y})$ also satisfies the check from line 6d.

Note that the accumulation of the inferred $\mathsf{M}_\mathtt{i}$ into M does not generate any inconsistency as long as the effect of each $\mathsf{M}_\mathtt{i}$ is reflected in the next state $\Delta_{\mathtt{i}+1}$ through the entailment residual frame (e.g. line 4f).

After each abduction our analysis checks whether the inferred formula can be part of the unknown procedure postcondition (e.g. checks from lines 4e and 6d). A failed check denotes a program error. In Figure 2 we consider the same example from Figure 1 but after line 4 we added a new instruction which dereferences a local variable. The new instruction is a program error. Our analysis discovers this error at line 4e when the check on the new abducted formula M_1 fails.

```
// Given Specification:
       // $\mathsf{Pre}_{\texttt{findLast}} := \mathsf{list}(\texttt{x},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null}$
       // $\mathsf{Post}_{\texttt{findLast}} := \mathsf{list}(\texttt{x},\texttt{res}) * \texttt{res}{\mapsto}\texttt{null}$
node findLast(node x) {
0   node w, y, z, a, b;
0a   // $\Delta_0 := \mathsf{Pre}_{\texttt{findLast}}$
0b   // $\Delta_0 \vdash \texttt{x}{\neq}\texttt{null}$
1   w := x.next;
1a   // $\Delta_1 := \texttt{x}{\mapsto}\texttt{w} * \mathsf{list}(\texttt{w},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null}$
2   if (w == null) return x;
2a   // $\Delta_2 := \texttt{w}{=}\texttt{null} \wedge \texttt{res}{=}\texttt{x} \wedge \texttt{x}{\mapsto}\texttt{null} \wedge \texttt{x}{\neq}\texttt{null}$
2b   // $\exists \texttt{w},\texttt{y},\texttt{z} \cdot \Delta_2 \vdash \mathsf{Post}_{\texttt{findLast}}$
3   else {
3a   // $\Delta_3 := \texttt{x}{\mapsto}\texttt{w} * \mathsf{list}(\texttt{w},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null} \wedge \texttt{w}{\neq}\texttt{null}$
3b   // $\mathsf{Local}(\Delta_3,\{\texttt{x}\}) := \texttt{x}{\mapsto}\texttt{w} * \mathsf{list}(\texttt{w},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null} \wedge \texttt{w}{\neq}\texttt{null}$
3c   // $\mathsf{Frame}(\Delta_3,\{\texttt{x}\}) := \texttt{x}{\neq}\texttt{null} \wedge \texttt{w}{\neq}\texttt{null}$
3d   // $\mathsf{Pre}_{\texttt{unkProc}} := \mathsf{Local}(\Delta_3,\{\texttt{x}\})$
4   y := unkProc(x);
4a   // $\Delta_4 := (\exists \texttt{fv}(\mathsf{Pre}_{\texttt{unkProc}}) \cdot \mathsf{Frame}(\Delta_3,\{\texttt{x}\})) * (\mathsf{emp} \wedge \texttt{y}{=}\texttt{y}' \wedge \texttt{x}{=}\texttt{x}')$
4b   // $\mathsf{M} := (\texttt{res}_{\texttt{unkProc}}{=}\texttt{y} \wedge \texttt{y}{=}\texttt{y}' \wedge \texttt{x}{=}\texttt{x}')$
4c   // $\Delta_4 \nvdash \texttt{b}{\neq}\texttt{null}$
4d   // $\Delta_4 * [\mathsf{M}_1] \triangleright \texttt{b}{\neq}\texttt{null}$
4e   // $\mathsf{M}_1 := \texttt{b}{\neq}\texttt{null}$
4f   // $\texttt{fv}(\mathsf{M}_1) \not\subseteq \texttt{ReachVar}(\Delta_4 * \mathsf{M}_1, \{\texttt{x}',\texttt{y}'\})$      Error!!!
5   a := b.next;
6   z := findLast(y);
7   return z;
8   } }
```

Figure 2: An error in the procedure `findLast`.

3 Multiple Invocations of an Unknown Procedure

In this section we illustrate how our analysis computes a precondition and a postcondition for an unknown procedure, which is invoked more than once. There are two cases: (1) the two unknown calls are in sequence and are executed one after another; (2) the two unknown calls are on different program execution paths. For the second case (e.g. two unknown calls are in two different branches of a conditional statement) it is not difficult to use the approach described in the previous section. First we infer the preconditions for both unknown calls by localizing the program state immediately before the calls. Then we do abductions till the end to gain their postconditions, respectively. Since the two specifications are inferred from different calling contexts, they are combined by conjunction [6]. In a later verification of the procedure's body they must be verified to ensure the implementation satisfies requirements from each calling context. Therefore in this section we concentrate on the first case where some source code invokes an unknown procedure many times in sequence.

We start from a simplest case. Figure 3 describes the general scenario consisting of a known procedure, known which calls twice the same unknown procedure unknown. Note that the code blocks b_1, b_2 and b_3 are composed by known program instructions. The specification of the known procedure known is given as the precondition $\mathsf{Pre}_{\mathtt{known}}$ and the postcondition $\mathsf{Post}_{\mathtt{known}}$. We denote by P_1, Q_1, and R_1 (P_2, Q_2, and R_2) the precondition, the postcondition and the frame, respectively for the first (the second) unknown procedure unknown call. Our goal is to infer P_1, Q_1, P_2 and Q_2 from procedure known's codes and specification.

```
$\mathsf{Pre}_{\mathtt{known}}$
procedure known()
  $b_1$;   // Code block one
  //  $P_1 * R_1$
  unknown($\vec{x_1}$);
  //  $Q_1 * R_1$
  $b_2$;   // Code block two
  //  $P_2 * R_2$
  unknown($\vec{x_2}$);
  //  $Q_2 * R_2$
  $b_3$;   // Code block three
end
$\mathsf{Post}_{\mathtt{known}}$
```

Figure 3: Two sequential unknown procedure calls.

In order to achieve our goal we assume there exists a most general specification $\{\mathsf{Pre}_{\mathtt{unknown}}\}$ unknown $\{\mathsf{Post}_{\mathtt{unknown}}\}$, such that $P_1 \vdash \mathsf{Pre}_{\mathtt{unknown}}$, $P_2 \vdash \mathsf{Pre}_{\mathtt{unknown}}$, $\mathsf{Post}_{\mathtt{unknown}} \vdash Q_1$ and $\mathsf{Post}_{\mathtt{unknown}} \vdash Q_2$. This assumption may not be useful for all possible unknown procedures unknown, since in the worst case the most general specification corresponds to the trivial specification $\{\mathsf{true}\}$ unknown $\{\mathsf{false}\}$. However our goal is to avoid the trivial specification and to get more precise result.

The general idea of our abduction-based approach is informally described in Figure 4. First our approach infers the precondition P_1 of the first unknown call and the postcondition Q_2 of the second unknown call in the same way as in the previous section. Afterwards we regard them as a raw specification for the unknown procedure (with necessary substitutions), and attempt to refine them with the codes in between the two unknown calls. Therefore we have $Q_1 := \sigma\ Q_2$ and $P_2 := \sigma^{-1}\ P_1$, and try to verify b_2

```
Algorithm InferTwoSpec
  Do forward analysis from $\mathsf{Pre}_{\texttt{known}}$ over $b_1$ to get $P_1 * R_1$;
  Distinguish $P_1$ as the local state of $\vec{\mathtt{x}}_1$;
  Do forward analysis with abduction from $\langle \mathsf{emp};\ \mathsf{emp} \rangle$ to
      $\langle \mathsf{Post}_{\texttt{known}};\ \mathsf{M}_1 \rangle$ over $b_3$ to get $\mathsf{M}_1$ as $Q_2$;
  Assume $Q_1 := \sigma\ Q_2$, and $P_2 := \sigma^{-1}\ P_1$;
  Do forward analysis with abduction from $\langle Q_1 * R_1;\ \mathsf{emp} \rangle$ to
      $\langle P_2 * R_2;\ \mathsf{M}_2 \rangle$ over $b_2$ to get $\mathsf{M}_2$;
  return $\{P_1\}$ unknown $\{Q_1 * \mathsf{M}_2\}$;
```

Figure 4: Analysis algorithm for two sequential unknown calls.

with the specification $\{Q_1 * R_1\}\ b_2\ \{P_2 * R_2\}$.

Here is an example to illustrate our general approach. We consider the procedure appendThree whose specification and implementation are shown in Figure 5. The procedure appends three singly linked lists into one, by updating one's tail to point to another's head. It has two invocations of the same unknown procedure unkProc, one followed by the other.

To focus on our algorithm itself, this example has no other codes except for the unknown calls, without losing generality. From the beginning we are provided with $\mathsf{Pre}_{\texttt{appendThree}}$ and $\mathsf{Post}_{\texttt{appendThree}}$, and our aim is to find the unknown procedure unkProc(x,y)'s specification.

As shown in line 0a, the first step is to infer the initial precondition of the unknown procedure call, P_1. This is accomplished by localizing the state before the first unknown call against its parameters x and y. We get $P_1 := \mathsf{list}(\mathtt{x},\mathtt{null}) * \mathsf{list}(\mathtt{y},\mathtt{null})$, with the frame part $\mathsf{list}(\mathtt{z},\mathtt{null})$ left unchanged and to be carried over to the second unknown call.

The second step beginning from line 2a is analogous to the inference of the unknown procedure call's postcondition in the last section. It starts with $\mathsf{emp} \wedge \mathtt{x} = \mathtt{x}' \wedge \mathtt{z} = \mathtt{z}'$ to assume the unknown call's effect as emp with logical variables to record the parameters' current values, and utilizes abduction when necessary to get the postcondition Q_2 for the unknown procedure call. In this case it directly gains the abduction result from appendThree's postcondition as $\mathsf{list}(\mathtt{x},\mathtt{y}) * \mathsf{list}(\mathtt{y},\mathtt{z}) * \mathsf{list}(\mathtt{z},\mathtt{null})$. As postprocessing, it then existentially quantifies y as it does not occur in the second unknown call's parameter list, and changes its name to u. So the postcondition is $\exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x},\mathtt{u}) * \mathsf{list}(\mathtt{u},\mathtt{z}) * \mathsf{list}(\mathtt{z},\mathtt{null})$.

The last step from line 1a performs a final check for $\{Q_1 * R_1\}\ b_2\ \{P_2 * R_2\}$. Here from context we know $R_1 := \mathsf{list}(\mathtt{z},\mathtt{null})$ and $R_2 := \mathsf{emp}$. According to our algorithm, Q_1 is assumed the same as Q_2 and P_2 the same as P_1, under some substitutions. Hence we have $Q_1 * R_1 := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x},\mathtt{u}) * \mathsf{list}(\mathtt{u},\mathtt{y}) * \mathsf{list}(\mathtt{y},\mathtt{null}) * \mathsf{list}(\mathtt{z},\mathtt{null})$, and $P_2 * R_2 := \mathsf{list}(\mathtt{x},\mathtt{null}) * \mathsf{list}(\mathtt{z},\mathtt{null})$. Then the entailment $Q_1 * R_1 \vdash P_2 * R_2$ is checked to ensure the correctness of the second invocation. As it succeeds, the specification for the unknown procedure $\mathtt{unkProc}(\mathtt{a}_1,\mathtt{a}_2)$ is as follows:

$$\begin{aligned}\mathsf{Pre}_{\texttt{unkProc}} &:= \mathsf{list}(\mathtt{a}_1,\mathtt{null}) * \mathsf{list}(\mathtt{a}_2,\mathtt{null})\\ \mathsf{Post}_{\texttt{unkProc}} &:= \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{a}_1,\mathtt{u}) * \mathsf{list}(\mathtt{u},\mathtt{a}_2) * \mathsf{list}(\mathtt{a}_2,\mathtt{null})\end{aligned}$$

However, it is still possible that $Q_1 * R_1$ is not sufficiently strong in the verification for b_2 to establish $P_2 * R_2$, especially when the specification for the known procedure is imprecise (either the precondition is excessively strong or the postcondition is too weak). For this sake we will use abduction in the verification to collect the heap states (M_2) that Q_1 lacks, and strengthen $\mathsf{Post}_{\texttt{unknown}}$ to be $Q_1 * \mathsf{M}_2$.

We have yet another example to illustrate this postcondition strengthening. Figure 6 describes a procedure towardsLast. According to its specification, the procedure takes the head of a linked list as

```
// Given Specification:
        // $\mathsf{Pre}_{\texttt{appendThree}} := \mathsf{list}(\texttt{x},\texttt{null}) * \mathsf{list}(\texttt{y},\texttt{null}) * \mathsf{list}(\texttt{z},\texttt{null})$
        // $\mathsf{Post}_{\texttt{appendThree}} := \mathsf{list}(\texttt{x},\texttt{y}) * \mathsf{list}(\texttt{y},\texttt{z}) * \mathsf{list}(\texttt{z},\texttt{null})$
// Goal:
        // To infer unkProc's specification
void appendThree(node x, node y, node z) {
0a    // Step 1:
0b    // $\Delta_0 := \mathsf{list}(\texttt{x},\texttt{null}) * \mathsf{list}(\texttt{y},\texttt{null}) * \mathsf{list}(\texttt{z},\texttt{null})$
0c    // $\mathsf{Local}(\Delta_0,\{\texttt{x},\texttt{y}\}) := \mathsf{list}(\texttt{x},\texttt{null}) * \mathsf{list}(\texttt{y},\texttt{null})$
0d    // $\mathsf{Frame}(\Delta_0,\{\texttt{x},\texttt{y}\}) := \mathsf{list}(\texttt{z},\texttt{null})$
0e    // $\mathsf{Pre}_{\texttt{unkProc}} := \mathsf{Local}(\Delta_0,\{\texttt{x},\texttt{y}\})$
1     unkProc(x, y);
1a    // Step 3:
1b    // Current state $\Delta_3 := \exists \texttt{u} \cdot \mathsf{list}(\texttt{x},\texttt{u}) * \mathsf{list}(\texttt{u},\texttt{y}) * \mathsf{list}(\texttt{y},\texttt{null}) * \mathsf{list}(\texttt{z},\texttt{null})$
1c    // Check entailment $\Delta_3 \vdash [\texttt{z}/\texttt{y}]\ \mathsf{Pre}_{\texttt{unkProc}}$
1d    // Entailment succeeds, and unkProc's specification is approved
2     unkProc(x, z);
2a    // Step 2:
2b    // $\Delta_1 := \mathsf{emp} \wedge \texttt{x}{=}\texttt{x}' \wedge \texttt{z}{=}\texttt{z}' \quad \mathsf{M} := \mathsf{emp} \wedge \texttt{x}{=}\texttt{x}' \wedge \texttt{z}{=}\texttt{z}'$
2c    // Do abduction to get $\Delta_1 * [\ \mathsf{M}_1\ ] \triangleright \mathsf{Post}_{\texttt{appendThree}}$
2d    // $\mathsf{M}_1 := \mathsf{list}(\texttt{x},\texttt{y}) * \mathsf{list}(\texttt{y},\texttt{z}) * \mathsf{list}(\texttt{z},\texttt{null})$
2e    // $\Delta_2 := \Delta_1 * \mathsf{M}_1 \quad \mathsf{M} := \mathsf{M} * \mathsf{M}_1$
2f    // $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{x},\texttt{z},\texttt{u} \cdot [\texttt{u}/\texttt{y}]\ \mathsf{M}$
2g    // $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{u} \cdot \mathsf{list}(\texttt{x}',\texttt{u}) * \mathsf{list}(\texttt{u},\texttt{z}') * \mathsf{list}(\texttt{z}',\texttt{null})$
3     }
```

Figure 5: Procedure appendThree calling an unknown procedure twice.

input, and returns any node in the list. It also calls twice the same unknown procedure in sequence, and our aim is to analyze the procedure's pre- and postconditions.

To start with, we still take `towardsLast`'s precondition to do forward analysis to get the precondition for the first unknown call. As shown in lines `0b` and `0e`, the program state immediately before that call is $\mathsf{list}(\texttt{y},\texttt{null}) \wedge \texttt{y}{\neq}\texttt{null} \wedge \texttt{y}{=}\texttt{x}$, and the localized precondition for the call is $\mathsf{list}(\texttt{y},\texttt{null}) \wedge \texttt{y}{\neq}\texttt{null}$.

In the second step, we use the same approach (forward analysis with abduction) to find out the postcondition of the second unknown procedure call, expressed from lines `3a` to `3g`. After that call we have no knowledge about the heap, and so the result of abduction will be the whole postcondition of `towardsLast`, $\mathsf{list}(\texttt{x},\texttt{z}) * \mathsf{list}(\texttt{z},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null}$. The current discovered postcondition is computed at line `3g`.

Last we also try to verify the codes between the two unknown calls, from the postcondition of the

```
// Given Specification:
        // $\mathsf{Pre}_{\mathtt{towardsLast}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x} \neq \mathtt{null}$
        // $\mathsf{Post}_{\mathtt{towardsLast}} := \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathsf{list}(\mathtt{res},\mathtt{null}) \wedge \mathtt{x} \neq \mathtt{null}$
node towardsLast(node x) {
0    node y := x, z;
0a   // Step 1:
0b   // $\Delta_0 := \mathsf{list}(\mathtt{y},\mathtt{null}) \wedge \mathtt{y}{\neq}\mathtt{null} \wedge \mathtt{y}{=}\mathtt{x}$
0c   // $\mathsf{Local}(\Delta_0,\{\mathtt{y}\}) := \mathsf{list}(\mathtt{y},\mathtt{null}) \wedge \mathtt{y}{\neq}\mathtt{null}$
0d   // $\mathsf{Frame}(\Delta_0,\{\mathtt{y}\}) := \mathsf{emp} \wedge \mathtt{y}{=}\mathtt{x}$
0e   // $\mathsf{Pre}_{\mathtt{unkProc}} := \mathsf{Local}(\Delta_0,\{\mathtt{y}\})$
1    y := unkProc(y);
1a   // Step 3:
1b   // Initialize $\mathsf{M} := \mathsf{emp} \wedge \mathtt{res}_{\mathtt{unkProc}}{=}\mathtt{y}$
1c   // Current state $\Delta_3 := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{u},\mathtt{y}) * \mathsf{list}(\mathtt{y},\mathtt{null}) \wedge \mathtt{u}{\neq}\mathtt{null} \wedge \mathtt{y}'{=}\mathtt{x}$
1d   // Do abduction to get $\Delta_3 * [\ \mathsf{M}_3\ ] \triangleright \mathsf{Pre}_{\mathtt{unkProc}}$
1e   // $\mathsf{M}_3 := \mathtt{y}{\neq}\mathtt{null}$     $\mathsf{M} := \mathsf{M} * \mathsf{M}_3$
1f   // So the unknown's postcondition is strengthened to be
1g   // $\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{x},\mathtt{y},\mathtt{z} \cdot \mathsf{Post}_{\mathtt{unkProc}} * \mathsf{M}$
1h   // $\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{u},\mathtt{res}_{\mathtt{unkProc}}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}},\mathtt{null}) \wedge \mathtt{u}{\neq}\mathtt{null} \wedge \mathtt{res}_{\mathtt{unkProc}}{\neq}\mathtt{null}$
2    z := unkProc(y);
3    return z;
3a   // Step 2:
3b   // $\Delta_1 := \mathsf{emp} \wedge \mathtt{y}{=}\mathtt{y}'' \wedge \mathtt{z}{=}\mathtt{z}' \wedge \mathtt{z}{=}\mathtt{res}$     $\mathsf{M} := \mathsf{emp} \wedge \mathtt{y}{=}\mathtt{y}'' \wedge \mathtt{z}{=}\mathtt{z}' \wedge \mathtt{z}'{=}\mathtt{res}_{\mathtt{unkProc}}$
3c   // Do abduction to get $\Delta_1 * [\ \mathsf{M}_1\ ] \triangleright \mathsf{Post}_{\mathtt{towardsLast}}$
3d   // $\mathsf{M}_1 := \mathsf{list}(\mathtt{x},\mathtt{z}) * \mathsf{list}(\mathtt{z},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
3e   // $\Delta_2 := \Delta_1 * \mathsf{M}_1$     $\mathsf{M} := \mathsf{M} * \mathsf{M}_1$
3f   // $\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{x},\mathtt{y},\mathtt{z} \cdot \mathsf{M}$
3g   // $\mathsf{Post}_{\mathtt{unkProc}} := \exists \mathtt{x} \cdot \mathsf{list}(\mathtt{x},\mathtt{res}_{\mathtt{unkProc}}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
4  }
```

Figure 6: Procedure `towardsLast` calling an unknown procedure twice.

first call (plus its frame part) to the precondition of the second. In this case there are no codes and an entailment checking is performed to guarantee that $[\mathtt{y}/\mathtt{res}_{\mathtt{unkProc}}]\ \mathsf{Post}_{\mathtt{unkProc}} \wedge \mathtt{y}'{=}\mathtt{x} \vdash \mathsf{Pre}_{\mathtt{unkProc}}$, where the logical variable $\mathtt{y}'$ stands for the previous value of the program variable $\mathtt{y}$. This check fails, because the first postcondition is not adequately strong. Therefore an abduction is necessary to enhance it, as accomplished in line `1d`. There we achieve an extra requirement for the postcondition such that the final specification for the unknown procedure `unkProc(a)` is as follows:

$$\begin{array}{lcl} \mathsf{Pre}_{\mathtt{unkProc}} & := & \mathsf{list}(\mathtt{a},\mathtt{null}) \wedge \mathtt{a}{\neq}\mathtt{null} \\ \mathsf{Post}_{\mathtt{unkProc}} & := & \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{u},\mathtt{res}_{\mathtt{unkProc}}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}},\mathtt{null}) \wedge \mathtt{u}{\neq}\mathtt{null} \wedge \mathtt{res}_{\mathtt{unkProc}}{\neq}\mathtt{null} \end{array}$$

In this case our inferred postcondition is strengthened to meet the request of the second unknown call's precondition. It is always safe to do this for two reasons. First, according to previous discussions, it is unsound to weaken a precondition or strengthen a postcondition in an analysis for some known codes' specifications [4]. However, since our aim is to find possible specifications of unknown procedure calls for a possibly later verification, when we weaken the precondition or strengthen the postcondition that we found, the range of "correct" programs is narrowed, which is always sound (safe), although at the cost of possible precision lost. Second, the strengthened postcondition is always capable to entail the known procedure's postcondition. This maintains the consistency of steps 2 and 3 in our analysis algorithm and stands for the other aspect of our approach's soundness.

```
Pre_known
procedure known()
   b_1        // Code block one;
   // P_1 * R_1
   unknown(x_1);
   // Q_1 * R_1
   b_2        // Code block two;
   // P_2 * R_2
   unknown(x_2);
   // Q_2 * R_2
   ...
   b_n        // Code block n;
   // P_n * R_n
   unknown(x_n);
   // Q_n * R_n
   b_{n+1}    // Code block n+1;
end
Post_known
```

Figure 7: n unknown procedure calls in sequence.

At last it is worth noting that, this approach can be extended without difficulty to cater for more invocations of the same unknown procedure. Suppose we have n calls as shown in Figure 7. In this case, first we still infer the precondition for the first unknown call and the postcondition for the last, to gain $\mathsf{Pre}_{\mathtt{unknown}} := P_1$ and $\mathsf{Post}_{\mathtt{unknown}} := Q_n$, respectively. After that we use abduction to verify $b_2, b_3, \ldots, b_n$ with the post- and preconditions. For b_i we take $\sigma_1\ \mathsf{Post}_{\mathtt{unknown}} * R_{i-1}$ as its precondition and $\sigma_2\ \mathsf{Pre}_{\mathtt{unknown}} * R_i$ as postcondition, to gather abductions that strengthen $\mathsf{Post}_{\mathtt{unknown}}$.

4 Conclusion

To verify the pointer safety for imperative programs with unknown procedure calls (or code pointers), we propose a novel approach to inferring the possible specifications for the unknown procedure from the calling contexts. We employ a forward shape analysis with separation logic and an abductive inference mechanism to synthesize both pre- and postconditions of the unknown procedure. We have defined the property that has to be satisfied by an abductive system in order to be useful for our approach. We have also discussed the solution for multiple calls of the same unknown procedure. The inferred specifications are partial specifications of the unknown procedure therefore they are subject to a later verification when the codes or the complete specifications of the unknown procedures become known.

Currently we are working on the formalization of our framework. We have also started to implement a prototype for an experimental validation of our approach.

Our approach is a general framework such that changing the abstract domain permits our framework to infer specifications within that new abstract domain. Two possible future works are to extend this method to broader shape domains and/or to other shape-related domains. The first adds more shape predicates to the domain to increase expressiveness, like doubly linked lists and trees ([13]). The second extension, for example the size domain ([5]), allows us to reason about properties such as length of lists, sortedness, and so forth. With the extended domains, the abstract semantics and analysis algorithm will remain conceivably the same, but the abduction will be redefined to discover the anti-frames for the newly introduced features. Since one possible application scenario for our approach consists of programs where the unknown procedures are only known at runtime, it might be interesting to combine our method with runtime verification of separation logic specifications [14].

Acknowledgement

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/E021948/1]. We would like to thank Hongseok Yang for helpful discussion on abduction and footprint analysis.

References

[1] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA, 2002. ACM.

[2] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, London, UK, 1995. Springer-Verlag.

[3] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis Symposium 2007 (SAS'07)*, Denmark, 2007.

[4] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, Georgia, USA, January 2009. ACM Press.

[5] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *Proc. 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 307–320, 2007.

[6] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Multiple pre/post specifications for heap-manipulating methods. In *Proc. 10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, Dallas, Texas, November 2007. IEEE CS Press.

[7] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*. Springer, 2006.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM.

[9] R. Giacobazzi. Abductive analysis of modular logic programs. In M. Bruynooghe, editor, *Proc. 1994 Int'l Symposium on Logic Programming (ILPS'94)*, pages 377–391. The MIT Press, 1994.

[10] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *Proceedings of International Conference on Computer Aided Verification 2007*, 2007.

[11] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium 2006 (SAS'06)*, 2006.

[12] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[13] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI 2007: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, 2007.

[14] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking of separation logic. In *VMCAI 2008: Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation*, LNCS. Springer, 2008.

[15] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Venice, Italy, January 2004. ACM Press.

[16] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

Towards Automated Property Discovery within Hume

Gudmund Grov & Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University, Riccarton, Scotland, EH14 4AS
`G.Grov@hw.ac.uk` `A.Ireland@hw.ac.uk`

Abstract

Hume is a Turing-complete programming language, designed to guarantee space and time bounds whilst still working on a high-level. Formal properties of Hume programs, such as invariants and transformations, have previously been verified using the temporal logic of actions (TLA). TLA properties are verified in an inductive way, which often requires lemma discovery or generalisations. Rippling was developed for guiding inductive proofs, and supports lemmas and generalisation discovery through proof critics. In this paper we show how rippling and proof critics can be used in the verification of Hume invariants represented in TLA. Our approach is based on existing work on the problem of verifying and discovering loop invariants for an imperative program. We then extend this work to Hume program transformations.

1 Introduction

In [32], Storey identifies *complexity of definitions*, *expressive power*, *bounded space and time*, *logical soundness*, *security* and *verifiability* as the key features for a programming language targeting safety-critical systems. No language supports all of them, and several of them are conflicting: for example, many time and space properties are undecidable for Turing-complete languages, and low-level languages, where this is decidable, often have a high complexity.

Hume [19] is a novel Turing-complete programming language designed to reduce the complexity of definition, whilst guaranteeing bounds on space and time usage, both key features in Storey's list. This is achieved by a layered architecture: a low-level, concurrent, finite state automata language, termed the *coordination layer*, is built on top of a high-level (Turing-complete) strict functional language, termed the *expression layer*. Programming then involves balancing between the two layers, and due to failed costing, this often requires transformations from the expression layer into the coordination layer. Hence, resource costing and program transformations are at the heart of the Hume development methodology.

The time and space analysis is well developed for Hume, as described in e.g. [17]. Correctness verification, which is also found on Storey's list, has previously been applied to Hume programs in the temporal logic of actions (TLA)[28].This work appears in [13], and can handle both invariants of the coordination layer, and verification of program transformations, which can be reduced to an invariant proof [16]. The proof of an invariant often requires the discovery of auxiliary invariants[1]. In [13], several Isabelle/HOL [31] tactics are given for reasoning about Hume programs within TLA. While a high degree of proof automation was achieved, a key missing ingredient is invariant discovery. Here, we build upon *rippling* [8], a search control technique designed for reasoning about inductive conjectures. In particular, we focus on *proof critics* [22] for rippling, which, can be used to guide the discovery of inductive invariants. Previously, these ideas have been applied to the verification and discovery of loop invariants for imperative programs [25, 24]. In this paper we will show how this work can be applied to Hume/TLA, and then extend it to program transformation verification. We also believe that this work is not limited to Hume, but applicable to generic Hume specification, which we will elaborate upon in §7.

The paper is structured as follows: we will first introduce the preliminaries in §2; this is followed by a discussion on the use of rippling to verify Hume invariants in §3; and proof critics to discover loop invariants in §4; this work is then extended to Hume program transformations in §5; before we discuss relevant work (§6); future work (§7); and conclude in §8.

[1]This is also the case for a generic TLA invariant, as discussed in [12].

2 Preliminaries

2.1 The temporal logic of actions (TLA)

The *temporal logic of actions* (TLA) [28] was developed to reason about concurrent systems, and combines temporal logic with actions. It is a uniform logic that can capture both safety and liveness requirements, however we will only discuss the safety aspect here. It is a three-tier logic where:

- in the *state level*, a *state function/predicate* is a function/predicate on one particular state, where a state is mapping from variables to values;
- in the *action level*, an *action* is a predicate on two states: a "before" and "result" state of the action;
- in the *temporal level*, a *formula* is a predicate on an infinite sequence of states.

All levels include a full predicate calculus. Additionally, the action level has a priming (') operator to separate variables in the "result" state (primed) from those of the "before" state. For example, $x' = x+1$ is the computation that increments x by 1. At this level, "before" variable v and its "result" counterpart v', are distinct. The temporal level has two additional operators: $\Box P$, which denotes P holds iff it holds for all following states of the sequence; and the $\boldsymbol{\exists}$ operator, for (temporal) existential quantification. $\boldsymbol{\exists}$ is used to hide variables internal for a specification. To show that a property holds for a program, we must show that the program *implements* the property. In TLA, both programs and properties are specified in the same logic, hence this is formalised as logical implication.

TLA allows specifications to be written at different levels of abstraction. The key to proving such refinements between abstract and concrete representations, is to allow *stuttering steps*, i.e. steps that leave the state unchanged. These steps are seen as internal steps within a specification. To define a (monolithic[2]) program we must specify an initial state I, and an action N, representing the transitions. N is a predicate which compares a "result" and a "before state". The action must hold throughout execution, i.e. $\Box N$. However, this does not support stuttering steps. Let $\langle v,i\rangle$ be the tuple of all visible variables v, and internal variables i of the program. We refine $\Box N$ to $\Box(N \vee \langle v,i\rangle' = \langle v,i\rangle)$, which asserts that in all transitions either N holds, or the state is left unchanged. This supports stuttering, and is abbreviated by $\Box[N]_{\langle v,i\rangle}$. We will use monolithic specifications of our programs. Such specifications, with the internal variables hidden, are written:

$$\boldsymbol{\exists}\, i : I \wedge \Box[N]_{\langle v,i\rangle}. \tag{1}$$

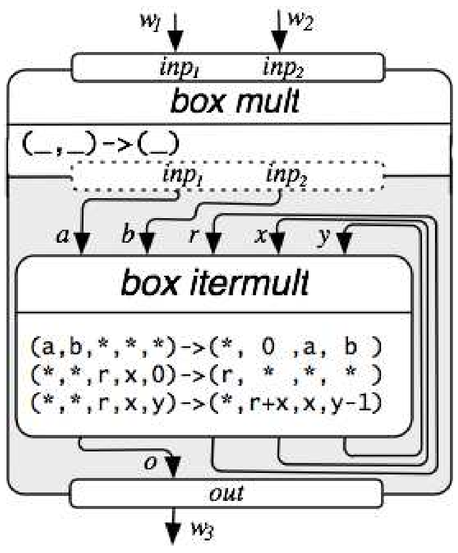
To ease reasoning, TLA requires that N must always specify the complete "result" state. Thus, unchanged variables must be explicitly stated. In a monolithic specification the subscript, i.e. $\langle v,i\rangle$, should therefore hold all the variables. Although $\boldsymbol{\exists}$ is semantically different from $\exists$, the proof rules are similar. For the work presented here for Hume, we can ignore $\boldsymbol{\exists}$ since our Hume invariants are independent of the $\boldsymbol{\exists}$-bound variables. Moreover, in a transformation proof the witness for i will always be the same. Thus, to ease the reading, we will a assume TLA specifications of the form:

$$I \wedge \Box[N]_v. \tag{2}$$

2.2 Hierarchical Hume

The *Hume coordination* layer describes a system as concurrent *boxes* linked by *wires*. Boxes are scheduled in a cyclical way, where each runnable box is executed in each step, and this process never terminates. In this paper we will use the *Hierarchical Hume* [14, 16] extension to Hume, which allows nesting

[2]A detailed explanation of a *monolithic* TLA specification can be found in [29].



```
(a,b,*,*,*)->(*, 0 ,a, b )
(*,*,r,x,0)->(r, * ,*, * )
(*,*,r,x,y)->(*,r+x,x,y-1)
```

$$\begin{array}{lrcl}
\text{(i)} & \Box(a*b & = & inp_1 * inp_2) \\
\text{(ii)} & \Box(\boldsymbol{r} + (\boldsymbol{x} * \boldsymbol{y}) & = & \boldsymbol{inp}_1 * \boldsymbol{inp}_2) \\
\text{(iii)} & \Box(o & = & inp_1 * inp_2) \\
\text{(iv)} & \Box(out & = & inp_1 * inp_2)
\end{array}$$

Figure 1: Hume multiplication as iteration.

```
mult:   $inp_1 := w_1;\ inp_2 := w_2;$
        { True }
          $a := inp_1;\ b := inp_2;$
          $r := 0;\ x := a;\ y := b;$
          while $(y \neq 0)$
          begin
            $r := r + x;\quad y := y - 1;$
          end
          $o := r;$
          $out := o;$
        {$out = inp_1 * inp_2$ }
```

Figure 2: Imperative multiplication as iteration.

of boxes inside another box. Now, a box consists of a set of *matches* of the form

pattern -> *expression*

where *pattern* is matched against the box's input wires. In a non-nesting box, a match will cause the *expression*, which belongs to the expression layer, to generate output to the output wires. In a nesting box, a match will copy the inputs into external input wires, and schedule the children boxes until the termination condition, defined by the *expression* is met. Then the internal output wires are copied to the output wires. If a pattern fails, the next match is attempted.

Hume boxes are scheduled in a two-phase lock step scheduling algorithm, where each step works as follows[3]:

[3]See for example [15] for details.

- each box is executed and output is produced in a result buffer (e.g. `out` of the `mult` box of Figure 1) in the execute phase;
- this is followed by a uniform super-step where outputs are asserted to the wires.

For a nesting box, this scheduler is nested, i.e. the children of the nesting box are scheduled similarly, until termination. To ease the reading, and enable focus on the key issues, we will abstract over this scheduling for the boxes that are nested. Moreover, these boxes are assumed to not be nesting, i.e. we assume a box hierarchy of two levels. Here, box execution and output assertion in one uniform step. By way of illustration, we present in Figure 1 an iterative implementation of a multiplier in Hume. Note that an equivalent imperative program is shown in Figure 2. Figure 1 graphically illustrates the uniform scheduling step. Here, the nested `itermult` box does not contain an output buffer, whilst `out` is the output buffer of the first level (nesting) `mult` box. Note that for our proofs below, the nesting `mult` box requires a lower abstraction level, containing both an input and an output buffer, as well as the two phase scheduling.

The `mult` box of Figure 1 performs multiplication by iteration. This is accomplished by the nested `itermult` box, which multiplies the inputs by iterative addition and is achieved by the feedback loop wires `r`,`x` and `y`. Note that the result is produced in one first-level step, even though many internal `mult` steps may be required.

Figure 2 describes the same program in an arbitrary imperative language, with the correctness condition annotated by Hoare triples [20]: $\{P\}c\{Q\}$ denotes that if P holds and statement c terminates, then Q holds. The Hume program then works as follows. If the wires $\mathtt{w}_1$ and $\mathtt{w}_2$ contains a value, then these are copied to the input buffer $\mathtt{inp}_1$ and $\mathtt{inp}_2$, and to the internal wires `a` and `b`; the internal `itermult` box is then scheduled until the `o` wire has got a value, which on termination is copied to the output buffer `out` (and output wire $\mathtt{w}_3$). The first match of the `itermult` box then succeeds, which copies the `a` and `b` wires to the `x` and `y` wires, while `r` is set to `0`. This is the "entry step" of the loop. In Hume, `*` means 'ignore' in a pattern, and 'do not write' in an expression. The third match is the "loop step" of the imperative program, and will fire when the `x`, `y` and `r` wires contain values and `y` $\neq$ `0`. It increments `r` by `y`, leaves `x` unchanged, and decrement `y` by `1`. The second match is the 'exit step' of the loop where the result of the tail-iteration `r` of the "loop steps" is copied to the `o` wire, and the termination condition of `mult` then holds, thus copying `o` to `out`.

2.3 Proof planning & rippling

Proof planning is a technique for automating proof search. Central to the technique is the notion of a *proof plan* [6], a high-level proof outline which encodes a common pattern of reasoning. We will focus here on a proof plan called *rippling*. Rippling is a rewriting technique based upon a difference reduction strategy. To illustrate, consider a conjecture where you are given a hypothesis of the form $(\forall b'.\ f(a,b'))$ while the goal takes the form $f(c_1(a),b)$. Note that the $c_1(\ldots)$ embedded within the goal prevents a match with the given hypothesis. In rippling, such embedded structures are called *wave-fronts*. The goal of rippling is to identify and reduce the number of wave-fronts such that a hypothesis can be applied. Wave-fronts can be represented using explicit annotations that are added to the goal. For example, using shading to denote wave-fronts, the goal given above becomes:

$$f(\boxed{c_1(\underline{a})}^{\uparrow}, \lfloor b \rfloor)$$

In addition to the shading, note that a wave-front is annotated with an arrow. The arrow indicates which direction the wave-front can be moved, *i.e.* upward or downward through the goal structure. There are two reasons for moving a wave-front downward. Firstly, if a wave-front can be moved to a position

Input sequent:

$$H \vdash G[f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))]$$

Method preconditions:

1. there exists a subterm T of G which contains wave-front(s), *e.g.*

$$f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))$$

2. there exists a wave-rule which matches T, *e.g.*

$$C \rightarrow f_1(\boxed{c_1(\underline{X})}^{\uparrow}, Y, Z) \Rightarrow \boxed{c_5(\underline{f_1(X, \boxed{c_3(\underline{Y})}^{\downarrow}, \boxed{c_4(\underline{Z})}^{\downarrow})})}^{\uparrow}$$

3. the wave-rule condition follows from the context, *e.g.*

$$H \vdash C$$

4. resulting inward directed wave-fronts are potentially removable, *e.g. sinkable or cancellable, i.e.*

$$\ldots \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow} \ldots$$

or

$$\ldots \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow} \ldots$$

Output sequent:

$$H \vdash G[\boxed{c_5(\underline{f_1(\ldots, \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}, \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow})})}^{\uparrow}]$$

Figure 3: The rippling method.

corresponding to a universally quantified variable within the given hypothesis, then the wave-front can be eliminated via the specialisation of the universal hypothesis. This is known as *sinking* a wave-front, and the $\lfloor\ldots\rfloor$ annotation within the goal is used to indicate sink positions. Secondly, multiple wave-fronts can sometimes cancel each other out, a kind of destructive interference. So moving wave-fronts closer together can also make sense. The manipulation of wave-fronts is achieved via so called *wave-rules*. A wave-rule is a rewrite rule that has been annotated by wave-fronts. A key property of wave-rules is that they preserve the unannotated structure of the goal, the so called *skeleton*. Preserving skeleton maximises the chances of eventually applying the given hypothesis. In the schematic example given above, the following represents an applicable wave-rule:

$$f(\boxed{c_1(\boxed{X})}^{\uparrow}, Y) \Rightarrow \boxed{c_2(\boxed{f(X, \boxed{c_3(\boxed{Y})}^{\downarrow})})}^{\uparrow}$$

Note that $\Rightarrow$ represents a rewrite, while $\rightarrow$ is used for logical implication. In general a proof plan contains *methods* and *critics* [21]. Rippling is represented by a single method as given in Figure 3. While methods represent common patterns of reasoning, critics are used to define patchable exceptions. When a method fails, its associated critics analyse the proof-failure and initiate a proof patch [21, 22, 23]. Typically the proof patching process makes use of meta-variables as place-holders for missing structure with the expectation that the constraints of the proof will provide instantiations during the planning of the remainder of the proof. This style of patching a proof is known as *middle-out reasoning* [7]. An example of a proof patch which exploits rippling and middle-out reasoning will be given in §4. For a complete description of rippling see [8].

3 Invariant verification

In [24], rippling was used to verify Hoare-triple properties as illustrated in Figure 2. To verify a Hoare-triple, it is converted into a *verification condition* (VC), a purely logical statement, by a *verification condition generator* (VCG). The VC is then verified by a theorem prover. However, before this can be done each statement must be turned into a Hoare-triple. This is mostly an automatic process, however finding and verifying an invariant which holds for the ***while*** loop, known as the *loop invariant*, is the hardest part. Thus, we will only focus on the loop invariant here. In the case of Figure 2, we use an invariant of the form $r + (x * y) = inp_1 * inp_2$. In the proof, the invariant is assumed beforehand, and this assumption is called the *invariant hypothesis* (IH)

$$\mathsf{IH}:\ r + (x * y) = inp_1 * inp_2. \tag{3}$$

Using the assumption, it is shown to hold after the loop has executed. By using wave-annotation, this goal is expressed as $\boxed{(\boxed{r} + x)}^{\uparrow} + (x * \boxed{(\boxed{y} - 1)}^{\uparrow}) = inp_1 * inp_2$. The proof requires the following wave-rules:

$$\boxed{(\boxed{X} + Y)}^{\uparrow} + Z \Rightarrow X + \boxed{(Y + \boxed{Z})}^{\downarrow} \tag{4}$$

$$X * \boxed{(\boxed{Y} - 1)}^{\uparrow} \Rightarrow \boxed{(\boxed{X * Y}) - X}^{\uparrow} \tag{5}$$

$$\boxed{(X + \boxed{\boxed{Y} - X}^{\uparrow})}^{\downarrow} \Rightarrow Y, \tag{6}$$

and is derived as follows:

$$\begin{array}{rl}
\boxed{(\boxed{r} + x)}^{\uparrow} + (x * \boxed{(\boxed{y} - 1)}^{\uparrow}) = inp_1 * inp_2 & \text{[apply (4)]} \\
r + \boxed{(x + (x * \boxed{(\boxed{y} - 1)}^{\uparrow}))}^{\downarrow} = inp_1 * inp_2 & \text{[apply (5)]} \\
r + \boxed{(x + (\boxed{\boxed{(x * y)} - x}^{\uparrow}))}^{\downarrow} = inp_1 * inp_2 & \text{[apply (6)]} \\
r + (x * y) = inp_1 * inp_2 & \text{[apply IH]}
\end{array}$$

Hoare logic was developed for sequential programs, and cannot be be applied directly to Hume programs due to issues of concurrency. Moreover, note that although Hume has a finite state machine architecture, *model checking* [10] is not in general suitable for program verification, due to a strong dependency with the data-centric expression layer, as described in [14]. However, it is applicable for small subsets of Hume, as described in [18], using the TLC model checker for the TLA$^+$ [29].

In TLA, programs and properties are represented in the same uniform logic: a property P holds for a program S, if S implements P, and implementation is represented as logic implication:

$$\vdash S \rightarrow P.$$

TLA always attempts to reduce temporal properties to the action level. This is illustrated by the derived induction rule for proving invariants of specification as shown in (2):

$$\frac{\vdash I \rightarrow P \qquad \vdash P \wedge V' = V \rightarrow P' \qquad \vdash P \wedge N \rightarrow P'}{\vdash I \wedge \Box[N]_V \rightarrow \Box P}. \tag{7}$$

The first sub-goal (assumption) is the base case, which states that P holds in the initial state I. The second case captures that the sub-script of the action V, at least contains the free variables of P, which is required for stuttering invariance. The last case, $\vdash P \wedge N \rightarrow P'$ states that P is preserved by action N. Here, P' means that P is a predicate over the result state of N. We will only discuss the action level here since the temporal level proofs are trivial for our examples. Moreover, the first two sub-goals are normally trivial, thus we will only discuss the main sub-goal, i.e. $\vdash P \wedge N \rightarrow P'$.

The partial correctness property of the `mult` box, is given by (iv) in Figure 1, which corresponds to the Hoare triples for the imperative program of Figure 2. As in the imperative case of Figure 2, the main part of the overall proof is the loop invariant. This corresponds to (ii) of Figure 1. The proof of the loop invariant depends on (i), the "loop entry" step. The invariant follows directly from the Hume semantics, while the partial correctness property (iv) follows directly from (iii), where (iii) is the "loop exist" step. (iii) can also be proven directly, using the loop invariant (ii). As in the imperative case, we will only discuss the loop invariant (ii) henceforth.

The last match of `itermult` corresponds to the ***while*** loop in the imperative program. Since the match expression is the result of executing a Hume box, it refers to the primed result state of an action. Let $[\![C]\!]$ be the semantics of a Hume construct C represented in TLA[4]. Due to the wiring, graphically illustrated in Figure 1, the annotated result of $[\![\texttt{(*,r-x,x,y-1)}]\!]$ is represented as:

$$\begin{array}{c} r' = \boxed{(\boxed{r}+x)}^{\uparrow} \quad x' = x \quad y' = \boxed{(\boxed{y}-1)}^{\uparrow} \\ inp_1' = inp_1 \qquad inp_2' = inp_2. \end{array} \tag{8}$$

The "loop invariant" in TLA is the same as in the imperative program, and the IH for the invariant proof is also the same (IH: $r + (x * y) = inp_1 * inp_2$). The ripple proof derivation starts of as

$$\begin{array}{rl} r' + (x' * y') = inp_1' * inp_2' & \text{[apply (8)]} \\ \boxed{(\boxed{r}+x)}^{\uparrow} + (x * \boxed{(\boxed{y}-1)}^{\uparrow}) = inp_1 * inp_2 & [\cdots] \end{array}$$

and the remaining proof is identical to the imperative program. Note, that the annotation step shown here, is handled by the verification condition generator (VCG) in the imperative version.

Blockage:

$$\boxed{\boxed{r}+x}^{\uparrow} = inp_1 * inp_2$$

Critic precondition:

- Precondition 1 of rippling succeeds, *i.e.*
  1. there is a subterm T of G which contains a wave-front(s), e.g.

$$\boxed{\boxed{r}+x}^{\uparrow}$$

- Precondition 2 of rippling partially succeeds, *i.e.*
  2. there exists a wave-rule which partially matches, e.g.

$$\boxed{\boxed{r}+x}^{\uparrow} \quad \text{with} \quad \boxed{(\boxed{X}+Y)}^{\uparrow} + Z \Rightarrow \ldots$$

Proof patch:
Speculate additional term structure within the conjecture such that preconditions 2, 3 and 4 will also potentially succeed, *i.e.*

$$F_1(\boxed{\boxed{r}+x}^{\uparrow}, x, \boxed{\boxed{y}-1}^{\uparrow}) = inp_1 * inp_2,$$

where F_1 is a higher-order meta-variable.

Figure 4: A tail-invariant proof critic instantiation.

4 Loop invariant discovery

The hardest part of Hoare-triple proofs, is the undecdable tasks of finding a strong enough loop invariant, like (3). [24] contains novel work, where proof critics [22] are used to explore partial ripple successes to discover a strong enough loop invariant. Firstly, both a ***while*** loop and a Hume feedback loop, as in Figure 1, require a tail-invariant, and the proof critic thus provides a tail-invariant patch. We will now apply the work described in [24] to discover the Hume "feedback loop invariant", required for the proof in the previous section. The post-condition for the property is $out = inp_1 * inp_2$, which is updated as follows: $out' = o$ and $o' = r$. Thus, an obvious first approximation of the loop invariant becomes:

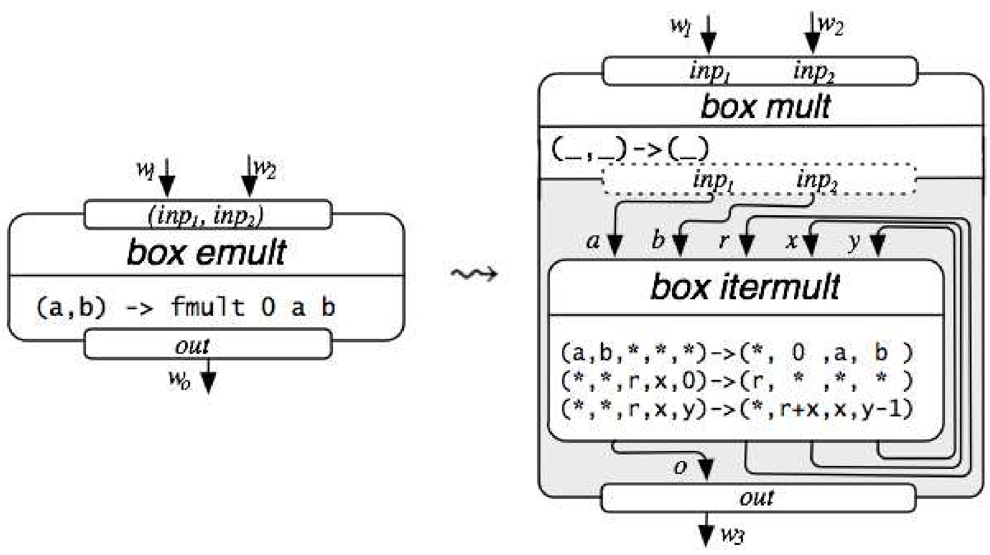
$$\mathsf{IH}: \; r = inp_1 * inp_2.$$

By the TLA "induction rule" (7), and (8), the proof of the "loop action" blocks when attempting to ripple

$$\underbrace{\boxed{\boxed{r}+x}^{\uparrow}}_{\textbf{blocked}} = inp_1 * inp_2.$$

A proof is blocked when there are no applicable wave rules, hence rippling is not possible. However, the precondition of the tail-invariant proof critic succeeds, as illustrated in Figure 4. That is, a partial match

[4]The Hume to TLA translation is not the topic here, and has thus been omitted. However, note that his is an operational representation of the Hume semantics, and in the translated TLA, we will use *italic font* instead of `sans serif`. Please see [13, 15, 18] for details of the TLA representation of Hume programs.



```
(a,b) -> fmult 0 a b
```

```
(a,b,*,*,*)->(*, 0 ,a, b )
(*,*,r,x,0)->(r, * ,*, * )
(*,*,r,x,y)->(*,r+x,x,y-1)
```

Figure 5: Box multiplication as recursion to iteration transformation

between the blocked wave front and the available wave rules suggests a schematic invariant of the form

$$F_1(r,x,y) = inp_1 * inp_2, \tag{9}$$

where F_1 is a second-order meta-variable. The expectation is that F_1 will be instantiated during the course of a rippling proof. The primed variables in $F_1(r',x',y') = inp'_1 * inp'_2$ are first rewritten using (8)

$$F_1(\boxed{r+x}^{\uparrow}, x, \boxed{y-1}^{\uparrow}) = inp_1 * inp_2.$$

By wave-rule (4), a new meta-variable F_2 is introduced by instantiating F_1 to $\lambda X.\lambda Y.\lambda Z.X + F_2(X,Y,Z)$. Application of (4) then derives

$$r + \boxed{x + F_2(\boxed{r+x}^{\uparrow}, x, \boxed{y-1}^{\uparrow})}^{\downarrow} = inp_1 * inp_2.$$

Here, wave rule (5) suggests an instantiation for F_2, i.e. $\lambda X.\lambda Y.\lambda Z.F_3(X,Y,Z) * (Z-1)$, which gives

$$r + \boxed{x + \boxed{F_3(\boxed{r+x}^{\uparrow}, x, \boxed{y-1}^{\uparrow}) * y - F_3(\boxed{r+x}^{\uparrow}, x, \boxed{y-1}^{\uparrow})}^{\uparrow}}^{\downarrow}.$$

Finally, wave rule (6) suggest that F_3 is instantiated to $\lambda X.\lambda Y.\lambda Z.Y$, resulting in the following invariant:

$$r + (x * y) = inp_1 * inp_2.$$

Note, the invariant is identical to the invariant in §3, and the proof structure is the same.

5 Hume program transformations

To formalise the relationship between expressiveness/high-levelness and resource bounds, Hume introduces a set of *levels*, where each downward-level restricts expressiveness and thus increases the set

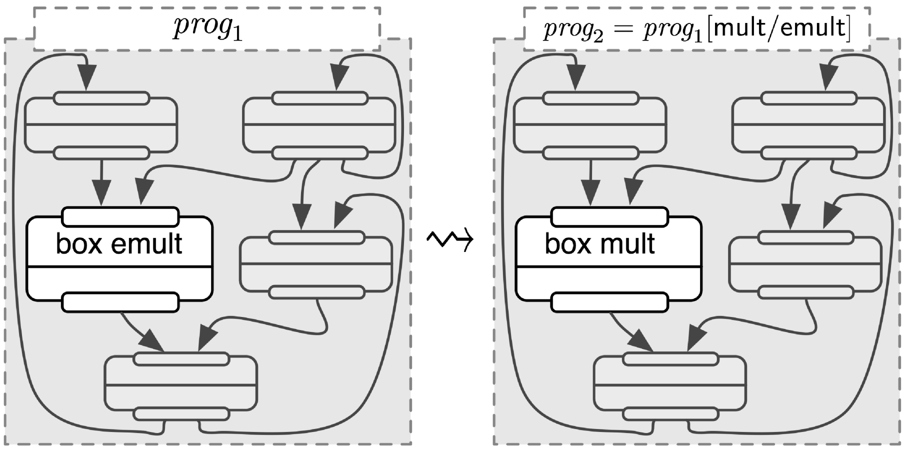


Figure 6: Program transformation example.

of decidable resource properties. Now, the Hume methodology is based around decidability analysis, which explores the different Hume levels: a high-level and expressive program is first created and resource costing is attempted; if it succeeds we are done; if it fails, the problem is identified and resource bounds are either altered, or the violating program constructs are transformed to a lower level and costing is reapplied. This process is iterative until costing succeeds. In the previous two sections, we showed how rippling and proof critics, used to verify imperative programs, can be applied to verify and discover Hume invariants using TLA. We will now show how this work can be extended to verify Hume transformations, a key concept of the Hume methodology.

Figure 5 shows the translation from a high-level Hume box, `emult`, which performs multiplication by primitive recursive addition, into the `mult` box previously discussed. In `emult` the addition is achieved by the `fmult` function:

```
fmult r _ 0 = r;
fmult r x y = fmult (r+x) x (y-1);
```

This `emult` box is in the *PR-Hume* level, where resource properties are undecidable. The `mult` box from Figure 1 on the other hand, is in the *FSM-Hume* level, which guarantees strong resource bounds. A typical Hume transformation, transforms a "recursive box", such as `emult`, into a more costable "iterative box", such as `mult`. Such a transformation is graphically illustrated in Figure 6, for an arbitrary program $prog_1$ containing `emult`. Note that

$$\texttt{fmult}\ 0\ x\ y = x * y$$

This is verified by first generalising it to $\forall r, x.\, \texttt{fmult}\ r\ x\ y = r + (x * y)$, which is then verified by induction on y. This proof has been mechanised in Isabelle/HOL, and is given in [13].

5.1 Transformation verification

In TLA, a transformation is represented as a *refinement*: a Hume program P_1 transforms into a Hume program P_2, which we write $P_1 \rightsquigarrow P_2$, iff P_2 *implements* P_1. Programs and properties are represented in the same logic, hence this is represented as a logical implication:

$$P_1 \rightsquigarrow P_2 \equiv [\![P_2]\!] \rightarrow [\![P_1]\!] \qquad (10)$$

Both $[\![P_2]\!]$ and $[\![P_1]\!]$ are assumed to be of form shown in (2). $P_1 \rightsquigarrow P_2$ is then verified by the following derived TLA proof rule:

$$\frac{\vdash J \rightarrow I \qquad \vdash W' = W \rightarrow V' = V \qquad \vdash M \rightarrow [N]_V}{\vdash J \wedge \Box[M]_W \rightarrow I \wedge \Box[N]_V}. \tag{11}$$

If the two first givens are indeed theorems, then they are normally trivial to verify. The main part is the proof of $\vdash M \rightarrow [N]_V$ which asserts that a step in the transformed program is either a step of the original source program, or all variables are left unchanged. Note, that as in (7), the rule (11) reduces the temporal level to the action level.

Let *correct nesting* of a box-to-box transformation denote that the inputs and outputs are the same, and any computation on wires is nested inside a box, hence both boxes compute results in one scheduling step. For example, the `emult` ($prog_1$) to `mult` ($prog_2$) transformation depicted in Figure 6 has correct nesting. Although informally, [16] shows that correctly nested box-to-box transformations, reduces to a proof of function correctness, meaning the following derivation holds:

$$\begin{array}{cl} prog_1 \rightsquigarrow prog_2 & \text{[apply (10)]} \\ [\![prog_2]\!] \rightarrow [\![prog_1]\!] & \text{[apply definition of } prog_2\text{]} \\ [\![prog_1[\mathsf{mult/emult}]]\!] \rightarrow [\![prog_1]\!] & \text{[apply result from [16]]} \\ [\![\mathsf{mult}]\!] \rightarrow [\![\mathsf{emult}]\!] & (\Pi) \end{array}$$

Moreover, [16] also shows that the proof of (Π) reduces to a pre-post condition proof, similar to the ones in the previous sections[5]. Here, the output buffer in the transformed box (e.g. `mult`) must produce the same output as the source box expression (e.g. `fmult 0 a b` of `emult` where `a` equals inp_1 and `b` equals inp_2). Hence, (Π) reduces the transformation proof to the following invariant:

$$out = \mathsf{fmult}\ 0\ inp_1\ inp_2 \tag{12}$$

Note that since the `mult` box is the assumption in the implication in (Π), *out*, inp_1 and inp_2 are "assumed updated" by the mult box. Moreover, with respect to the overall $\vdash M \rightarrow [N]_V$ conjecture, all but the termination step implies $\vdash V' = V$, while in the termination step $\vdash M \rightarrow N$. Since, N implies the application of `fmult`, this conjecture follows from this pre-post condition.

As in the previous sections, the key to the proof of (12), is the loop invariant of the nested feedback loop of mult. From the definition of `fmult` the following conditional wave-rule is derived:

$$Y \neq 0 \rightarrow \mathsf{fmult}\ \boxed{(\underline{R} + X)}^{\uparrow}\ X\ \boxed{(\underline{Y} - 1)}^{\uparrow} \Rightarrow \mathsf{fmult}\ R\ X\ Y \tag{13}$$

This is the only rule required in the proof. The loop invariant here is fmult $r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$, and the invariant hypothesis is obviously the same:

$$\text{IH: } \mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The "loop step" of the `mult` box then induces the following derivation:

$$\begin{array}{cl} \mathsf{fmult}\ r'\ x'\ y' = \mathsf{fmult}\ 0\ inp_1'\ inp_2' & \text{[apply (8)]} \\ \mathsf{fmult}\ \boxed{(\underline{r} + x)}^{\uparrow}\ x\ \boxed{(\underline{y} - 1)}^{\uparrow} = \mathsf{fmult}\ 0\ inp_1\ inp_2 & \text{[apply (13)]} \\ \mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2 & \text{[apply IH]} \end{array}$$

Note that in the application of (13), the pre-condition of the rule holds, by the definition of the corresponding pattern. If $y = 0$, the second match would have succeeded.

[5]Mechanised case-studies in Isabelle/HOL, which appear in [13], have provided empirical evidence for this approach as well.

5.2 Invariant discovery

The tail-invariant critic, where one particular instantiation is illustrated in Figure 4, can be reused to discover the loop invariant in this example, although the particular rules will deviate. Similar to §4, our first approximation of the invariant is $r = \mathsf{fmult}\ 0\ inp_1\ inp_2$, gives rise to a blocked ripple, i.e:

$$\underbrace{\boxed{\boxed{r} + x}^{\uparrow}}_{\textbf{blocked}} = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

However, the precondition of the tail-invariant patch succeeds, which results in the introduction of a meta-variable F_1:

$$F_1(r,x,y) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The definition of the priming operators (8) results in:

$$F_1(\boxed{\boxed{r}+1}^{\uparrow}, x, \boxed{\boxed{y}-1}^{\uparrow}) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

Wave-rule (13) then suggests that F_1 is instantiated in terms of fmult, and the same loop invariant as in the previous section is obtained:

$$\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

6 Relevant work

Our Hume/TLA work build directly on [25, 24], which uses rippling [8] and proof critics [22] to verify imperative programs. Further, we have extended these ideas to transformation proofs [16]. In [16], it is shown how a Hume transformation proof can be reduced to an invariant proof. The work with rippling and critics is novel for Hume/ TLA and transformations, and the generic work with Hume and TLA is novel with respect to using TLA at the programming language level. We believe TLA is more suitable for Hume verification compared to process algebras [4], like CCS, CSP or the Π-calculus.

The work presented here comes out from the first authors PhD thesis [13]. Parts of this work involves a mechanisation of TLA in Isabelle/HOL [31], and a representation of the semantics of Hume programs on top of this. Several proof tactics were developed to automate these proofs. The invariant discovery ideas presented in this paper would obviously increased the degree of automation of these tactics.

Previously, transformation verification has been described [16]. [14] outlines a box calculus for Hume. There, a set of behaviour preserving rules and strategies were defined to transform a Hume program into another. TLA has also been used to model check programs at the lowest, least expressive, HW-Hume level [18], and to reason about different Hume scheduling strategies [15].

Finally, the work presented here is at the action level of TLA, which is similar to Action Systems [3] and Event-B [2]. Thus, we believe this work is also applicable in these formalisations.

7 Future work

This paper has applied rippling and critics to one small example. Extending this to programs of multiple concurrent boxes will not have any impact on the approach, since the invariant will remain invariant due to the strict wire communication. We have not implemented the ideas presented in this paper, and this will be our next step. For imperative programs, the loop invariant generation techniques have already been implemented and tested [25, 24], In the Hume case, Hume/TLA proofs have been mechanised in the Isabelle/HOL theorem prover [13], so via IsaPlanner [11], invariant discovery should not be hard to implement.

A longer term goal is to be able to synthesise transformations. Building directly upon [26], we will now outline how we believe the work described here can be extended to the synthesis of transformations.

In [26], the problem is stated as: given a Hoare triple $\{P\}C\{Q\}$, find an instantiation of C in a small generic imperative program (containing assignment, conditionals and ***while*** loops) such that the triple is valid. The approach combines proof planning with conventional partial order planning. Here, proof planning is used for the local perspective, i.e. finding correct statements, while partial order planning is used to achieve a correct order of statements. This is implemented in a tool called Bertha, which is automatic with the exception that the user has to provide loop invariants.

In Hume, this work would be adapted to automate the synthesis of transformations which are a result of a failed costing. However, an additional requirement here is that the transformations are level-reducing (i.e. it becomes more decidable with respect to time and space properties). To illustrate, consider again the recursion to iteration transformations shown in Figures 5 and 6. The program starts with $prog_1$, and the coster fails on the `emult` box. Next, failure analysis is applied and the problem is that the recursive `fmult` function cannot be costed, which causes a "`recursion_to_iteration` proof method (plan)" to be applied. Now, this uses the expression of the `emult` box to create the specification, i.e.

$$out = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

Moreover, the program knows that a nested box with feedback loop must be created, since this is the only way to represent iteration in Hume. Moreover, we know the program must contain one or more entry matches, loop matches and exit matches, while the loop invariant can be directly found using the techniques described in Section 5.

Partial order planning could then be used to find the correct order of the matches, for example when pattern matching is used, the loop exit (∗,∗,r,x,0) -> (r,∗,∗,∗) match must precede the loop body (∗,∗,r,x,y) -> (∗,r+x,x,y-1) match. If not, the box will never terminate. Moreover, the loop entry step has different input wires (a,b,∗,∗,∗) than the body and exit steps, while the exit step has different output wires (r,∗,∗,∗). Note that with respect to the box calculus [14] described in the previous section, this approach is more flexible since it is not constrained by an existing set of rules.

TLA, and the full TLA$^+$ specification language [29], which combines TLA with a variant of ZF set theory, has been used both in industry and academia [5, 12, 27]. In [12], Gafni and Lamport illustrate the building of a sufficiently strong invariant by verifying the Disc Paxos algorithm. The algorithm is verified in a bottom-up fashion, where smaller invariants are gradually built up until they are strong enough to verify the main theorem. In a top-down approach the main theorem is first attempted to be verified, and from a partial success, a required invariant is found and verified. This process is iterated until a sufficiently small invariant is found which can be proved directly. Gafni and Lamport argues that both a top-down and bottom-up approach can be used. The top-down approach is a similar approach to the one discussed here. The work described in this paper is exploring Hume programs represented as TLA formulas. Hence, we are only manipulating the TLA terms, and not the Hume code. Thus, we believe that an approach based on rippling and proof planning is applicable to automate the verification of generic TLA invariants, also outside the Hume/TLA context, like in [12].

Another possible role of proof planning is properties involving the $\exists$ operator, which is used to hide internal details of a specification. In this paper, we have ignored the $\exists$ operator, since it was not that relevant for the Hume context. This follows from the fact that $\exists$ is handled in the same way for all programs[6]. In general, to prove a refinement with bound variables in TLA, that is, of the form:

$$(\exists B.\, J \wedge \Box[M]_W) \rightarrow (\exists A.\, I \wedge \Box[N]_V)$$

[6]The details will appear in [13]

one must first remove/instantiate the $\boldsymbol{\exists}$ bound variables A and B. Then, (11) can be applied. This is achieved with proof rules similar to those for standard predicate existential quantification $\exists$. A key step in such a proof, is to find the correct witness for A, known as the *refinement mapping* [1]. Proof planning has previously been used to find complex witnesses for $\exists$ bound variables [9, 30]. We believe that due to the similarity between the rules for $\exists$ and $\boldsymbol{\exists}$, a proof planning approach can also be used to find refinement mappings.

8 Conclusion

Hume is a Turing-complete programming language, designed to guarantee space and time bounds, whilst working on a high-level. Correctness properties, such as invariants and transformations, have previously been verified using the temporal logic of actions (TLA). Such verification efforts require mathematical induction. Rippling was developed for guiding inductive proofs, and rippling based proof critics have been developed to discovery and generalisations of invariants.

Here, we have shown the use of rippling to both verify and discover invariants, based on existing work on verifying and discovering loop invariants for imperative programs. This work has then been extended to transformations. We believe that the work is also applicable in a generic TLA setting, which we have elaborated upon.

Acknowledgements

This work has been supported by EU FP6 EmBounded.

References

[1] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 31 May 1991.

[2] Jean-Raymond Abrial. *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To be published.

[3] Ralph-Johan Back. Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, pages 67–93, London, UK, 1990. Springer-Verlag.

[4] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Scisence*, 335(2-3):131–146, 2005.

[5] Brannon Batson and Leslie Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, number 2852 in Lecture Notes in Computer Science, pages 242–261. Springer, March 17 2003.

[6] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *Proc of CADE'88*, pages 111–120. Springer-Verlag.

[7] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proc. of UK IT 90*, pages 221–6. IEE, 1990.

[8] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling – Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[9] Alan Bundy, Alan Smaill, and Jane Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proc. of UK IT 90*, pages 221–6. IEE, 1990.

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

[11] Lucas Dixon and Jaques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[12] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[13] Gudmund Grov. *Reasoning about Correctness Properties of a Coordination Programming Language*. PhD thesis, Heriot-Watt University, 2009. January 2009 submission. Subject to oral examination.

[14] Gudmund Grov and Greg Michaelson. Towards a Box Calculus for Hierarchical Hume. In Marco T. Morazan, editor, *Trends in Functional Programming*, volume 8, chapter 5, pages 71 – 88. Intellect, 2007.

[15] Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007.

[16] Gudmund Grov, Robert Pointon, Greg Michaelson, and Andrew Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. In *Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing*, volume 1 of 3, pages 126 – 127, 1515 Broadway, New York, March 2008. The Association for Computing Machinery, Inc.

[17] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards Formally Verifiable WCET Analysis for a Functional Programming Language. In *Proceedings of 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[18] Kevin Hammond, Gudmund Grov, Greg Michaelson, and Andrew Ireland. Low-level programming in Hume: an exploration of the HW-Hume level. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *In Proceedings of Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, pages 91 – 107. Springer, 2006.

[19] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003.

[20] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[21] A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *In Proc of LPAR'92*, LNCS 624, pages 178–189. Springer-Verlag, 1992.

[22] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[23] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[24] Andrew Ireland and Jamie Stark. On the Automatic Discovery of Loop Invariants. In *The Fourth NASA Langley Formal Methods Workshop*, number 3356. NASA Conference Publication, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[25] Andrew Ireland and Jamie Stark. Proof Planning for Strategy Development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001.

[26] Andrew Ireland and Jamie Stark. Combining Proof Plans with Partial Order Planning for Imperative Program Synthesis. *Automated Software Engineering Journal*, 13(1):65–105, 2006.

[27] Peter B. Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy Caching in TLA. *Distributed Computing*, 12(2-3):151–174, 1999.

[28] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[29] Leslie Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading, Massachusetts, 2002.

[30] Erica Melis and Jörg Siekmann. Knowledge-based proof planning. 115(1):65–105, 1999.

[31] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[32] N. Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

Invariant Assertions, Invariant Relations, and Invariant Functions

Asma Louhichi, Olfa Mraihi, Lamia Labed Jilani and Ali Mili
Institut Superieur de Gestion, Tunis, Tunisia
NJIT, Newark NJ, USA
lamia.labed@isg.rnu.tn, mili@cis.njit.edu

Abstract

Invariant assertions play an important role in the analysis and documentation of while loops of imperative programs. Invariant functions and invariant relations are alternative analysis tools that are distinct from invariant assertions but are related to them. In this paper we discuss these three concepts and analyze their relationships. The study of invariant functions and invariant relations is interesting not only because it provides alternative means to analyze loops, but also because it gives us insights into the structure of invariant assertions, and may help us enhance techniques for generating invariant assertions.

Keywords

Invariant assertions, invariant functions, invariant relations, loop invariants, program analysis, program verification, while loops.

1 Introduction

In [5], Hoare introduced the concept of an *invariant assertion* as a useful tool in the analysis of while loops. The study of invariant assertions, and their use in the analysis of while loops, has been the focus of active research in the seventies and eighties, and the subject of renewed interest in the last few years. In this paper we wish to investigate the relationships between this well known, throughoutly researched, concept, and two distinct but related concepts: invariant functions [10] and invariant relations [8]. We consider a while loop w on space S defined by $w =$ `while t do B`. These three concepts can be summarily and informally characterized as follows:

- An invariant assertion (as defined for the purposes of our discussion) is a predicate α on S that is preserved by application of the loop body.
- An invariant function is a total function on S whose value is preserved by application of the loop body.
- An invariant relation is a reflexive transitive relation containing pairs of states (s,s') such that s' is obtained from s by application of an arbitrary number of iterations of the loop body.

In section 3, we give definitions of all three concepts, using a uniform model, namely the calculus of relations, which we introduce in section 2. In section 4 we discuss the interrelations between these concepts, and explore complementarities between their respective insights on loop behavior; some of the results we present in this section are proven propositions, while others are mostly unproven conjectures for the time being. In section 5 we summarize our results and explore venues of further research.

2 Relational Mathematics

2.1 Elements of Relations

We consider a set S defined by the values of some program variables, say x, y and z; we typically refer to elements of S by s, and we note that s has the form $s = \langle x,y,z\rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ to refer to the x-component, y-component and z-component of s. We may sometimes user x to refer to x(s) and x' to refer to $x(s')$, when this raises no ambiguity. We use relations to represent program specifications and program functions; we then refer to S as the *space* of the specification, and also refer to it as the space of any program that manipulates these variables. Constant relations on some set S include the *universal* relation, denoted by L, the *identity* relation, denoted by I, and the *empty* relation, denoted by $\emptyset$. Given a predicate t, we denote by $I(t)$ the subset of the identity relation defined as follows: $I(t) = \{(s,s')|s' = s \wedge t(s)\}$.

Because relations are sets, we use the usual set theoretic operations between relations (union, intersection, complement, cartesian product). Operations on relations also include the *converse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$. The *product* of relations R and R' is the relation denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s,s')|\exists t : (s,t) \in R \wedge (t,s') \in R'\}$. The *prerestriction* (resp. *post-restriction*) of relation R to predicate t is the relation $\{(s,s')|t(s) \wedge (s,s') \in R\}$ (resp. $\{(s,s')|(s,s') \in R \wedge t(s')\}$). We admit without proof that the pre-restriction of a relation R to predicate t is $I(t) \circ R$ and the post-restriction of relation R to predicate t is $R \circ I(t)$. The *domain* of relation R is defined as $dom(R) = \{s|\exists s' : (s,s') \in R\}$. The *range* of relation R is denoted by $rng(R)$ and defined as $dom(\widehat{R})$. We admit without proof that for a relation R, $RL = \{(s,s')|s \in dom(R)\}$ and $LR = \{(s,s')|s' \in rng(R)\}$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined as $R\widehat{R}$.

We say that R is *deterministic* (or that it is a *function*) if and only if $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$, or equivalently, $RL = L$. Given two total deterministic relations R and R', we say that R is *more-injective* than R' if and only if

$$R\widehat{R} \subseteq R'\widehat{R'}.$$

To understand this definition, consider that each function partitions its domain into equivalence classes, called the *level sets* of the function [6]; a more-injective function is one whose level sets define a finer partition of the domain. A total deterministic relation R is said to be *injective* if and only if it is more-injective than I.

A relation R is said to be *rectangular* if and only if $R = RLR$. A relation R is said to be *reflexive* if and only if $I \subseteq R$, *transitive* if and only if $RR \subseteq R$ and *symmetric* if and only if $R = \widehat{R}$. We are interested in two special types of rectangular relations: rectangular surjective relations are called *vectors* and satisfy the condition $RL = R$; rectangular total relations are called *invectors* (inverse of a vector) and satisfy the condition $LR = R$. In set theoretic terms, a vector on set S has the form $A \times S$, and an invector has the form $S \times A$, for some subset A of S. Vector $A \times S$ can also be written as $I(A) \circ L$.

2.2 Refinement Ordering

We define an ordering relation on relational specifications under the name *refinement ordering*:

Definition 1. *A relation R is said to* refine *a relation R′ if and only if*

$$RL \cap R'L \cap (R \cup R') = R'.$$

In set theoretic terms, this equation means that the domain of R is a superset of (or equal to) the domain of R', and that for elements in the domain of R', the set of images by R is a subset of (or equal to) the set of images by R'. This is similar to, but different from, refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [4, 12]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit that, modulo traditional definitions of total correctness [3, 4, 7], the following propositions hold:

- A program P is correct with respect to a specification R if and only if $[P] \sqsupseteq R$, where $[P]$ is the function defined by P (we may, by abuse of notation use P to refer to $[P]$ when the context allows).

- $R \sqsupseteq R'$ if and only if any program correct with respect to R is correct with respect to R'.

Intuitively, R refines R' if and only if R represents a stronger requirement than R'.

2.3 Refinement Lattice

We admit without proof that the refinement relation is a partial ordering. In [2] Boudriga et al. analyze the lattice properties of this ordering and find the following results:

- Any two relations R and R' have a greatest lower bound, which we refer to as the *meet*, denote by $\sqcap$, and define by:
$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$
- Two relations R and R' have a least upper bound if and only if they satisfy the following condition (which we refer to as the *consistency condition*):
$$RL \cap R'L = (R \cap R')L.$$
Under this condition, their least upper bound is referred to as the *join*, denoted by $\sqcup$, and defined by:
$$R \sqcup R' = \overline{RL} \cap R' \cup \overline{R'L} \cap R \cup (R \cap R').$$
Intuitively, the join of R and R', when it exists, behaves like R outside the domain of R', behaves like R' outside the domain of R, and behaves like the intersection of R and R' on the intersection of their domain. The consistency condition provides that the domain of their intersection is identical to the intersection of their domains.
- Two relations R and R' have a least upper bound if and only if they have an upper bound; this property holds in general for lattices, but because the refinement ordering is not a lattice (since the existence of the join is conditional), it bears checking for this ordering specifically.
- The lattice of refinement admits a *universal lower bound*, which is the empty relation.
- The lattice of refinement admits no *universal upper bound*.
- Maximal elements of this lattice are total deterministic relations.

See Figure 1. The outline of this figure shows the overall structure of the lattice of specifications.

3 Invariants

In this section we present in turn the three invariants in parallel and present, for each: a formal definition in relational terms, an ordering relation between invariants, a characterization of the strongest invariant, and an elucidation of their relation to the function of the loop.

3.1 Formal Definitions

We consider a while statement of the form w = `while t do B` on some space S, and we assume that w terminates normally for any initial state s in S. We define in turn, invariant assertions, invariant relations, and invariant functions. For the sake of uniformity, we use the same mathematical baggage to represent them, namely the calculus of relations.

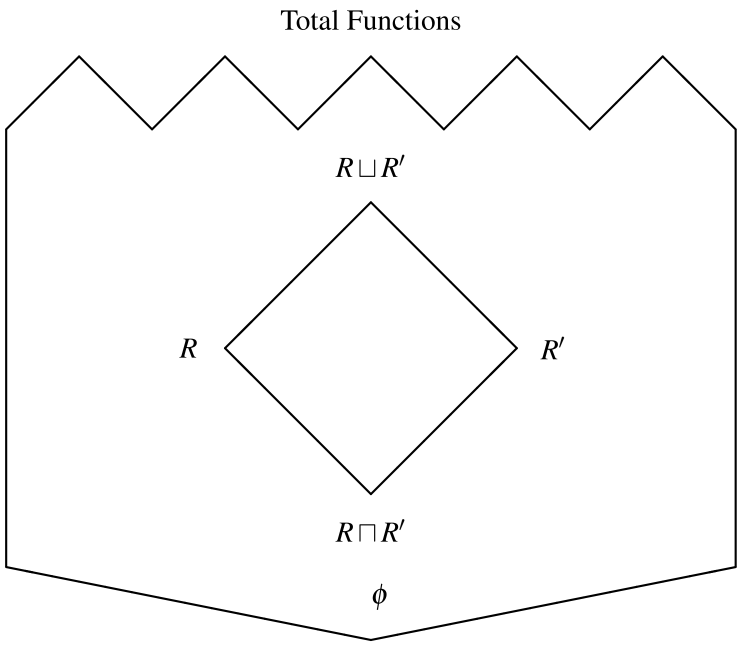


Figure 1: Lattice Structure of Refinement

For the sake of illustration, we consider a running example on which we apply our definitions and propositions. We consider the following while loop on array variables *a* and *b*, real variables *x* and *y*, and index variables *i* and *j*:

```
while (i!=N+1)
{x=x+a[i]; y=y+b[j]; i=i+1; j=j-1;}
```

3.1.1 Invariant Assertions

Traditionally, an invariant assertion α for the while loop

$$w = \texttt{whiletdoB}$$

with respect to a precondition/ postcondition pair (ϕ, ψ) is defined as a predicate on S that satisfies the following conditions:

- $\phi \Rightarrow \alpha$.
- $\{\alpha \wedge t\}B\{\alpha\}$.
- $\alpha \wedge \neg t \Rightarrow \psi$.

As defined, the invariant assertion is dependent not only on the while loop, but also on the loop's specification, in the form of a precondition/ postcondition pair. This precludes meaningful comparisons with invariant relations and invariant functions, which are dependent solely on the loop. Hence we redefine the concept of invariant assertion in terms of the second condition alone. Also, to represent an invariant assertion, we map the predicate on S into a vector (a relation) on S. Specifically, we represent the predicate α by the vector A defined by

$$A = \{(s,s')|\alpha(s)\}.$$

Definition 2. *Given a while statement of the form,* `w = while t do B`*, an* invariant assertion *is defined as a non-empty vector A on S that satisfies the following condition:*

$$A \cap T \cap [B] \subseteq \widehat{A},$$

where T is the vector defined by predicate t, i.e. $T = \{(s,s')|t(s)\}$.

This is a straightforward interpretation, in relational terms, of the second condition (above),

$$\{\alpha \wedge t\}B\{\alpha\}.$$

For the sake of illustration, we submit that the following vector satisfies the condition of Definition 2:

$$A = \{(s,s')|x = \sum_{k=1}^{i-1} a[k]\}.$$

We compute the left hand side of the definition:

$$\begin{array}{ll}
 & A \cap T \cap [B] \\
= & \qquad \{ \text{ Substitutions } \} \\
 & \{(s,s')|x = \sum_{k=1}^{i-1} a[k] \wedge i \neq N+1 \wedge a' = a \wedge b' = b \wedge \\
 & x' = x + a[i] \wedge i' = i+1 \wedge y' = y + b[j] \wedge j' = j-1\} \\
\subseteq & \qquad \{ \text{ Set Theory } \} \\
 & \{(s,s')|x = \sum_{k=1}^{i-1} a[k] \wedge i \neq N+1 \wedge a' = a \wedge x' = x + a[i] \wedge i' = i+1\} \\
= & \qquad \{ \text{ Simplification } \} \\
 & \{(s,s')|x = \sum_{k=1}^{i-1} a[k] \wedge i \neq N+1 \wedge a' = a \wedge x' = \sum_{k=1}^{i-1} a[k] + a[i] \wedge i' = i+1\} \\
= & \qquad \{ \text{ Simplification } \} \\
 & \{(s,s')|x = \sum_{k=1}^{i-1} a[k] \wedge i \neq N+1 \wedge a' = a \wedge x' = \sum_{k=1}^{i} a[k] \wedge i' = i+1\} \\
= & \qquad \{ \text{ Substitution } \} \\
 & \{(s,s')|x = \sum_{k=1}^{i-1} a[k] \wedge i \neq N+1 \wedge a' = a \wedge x' = \sum_{k=1}^{i'-1} a'[k] \wedge i' = i+1\} \\
\subseteq & \qquad \{ \text{ Set Theory } \} \\
 & \{(s,s')|x' = \sum_{k=1}^{i'-1} a'[k]\} \\
= & \qquad \{ \text{ Substitution } \} \\
 & \widehat{A}.
\end{array}$$

Note that we have not proven that the assertion $x = \sum_{k=1}^{i-1} a[k]$ holds after each iteration; rather we have only proven that if this assertion holds at one iteration, then holds at the next iteration. This is in effect an inductive proof without a basis of induction.

3.1.2 Invariant Relations

Invariant relations are relations that contain pairs of states (s,s') such that s' follows from s by application of an arbitrary number of iterations. Because the number of iterations separating s and s' can be zero, invariant relations are reflexive; and because the number of iterations can take an arbitrary value greater than zero, such relations are naturally transitive (since they are not dependent on the number of iterations). We define them as follows.

Definition 3. *Given a while loop of the form* `w = while t do B` *on some space S, and given a relation R on S, we say that R is an* invariant relation *for w if and only if R is reflexive, transitive, and satisfies the following conditions (where T is the vector defined by predicate t and $\overline{T}$ is the complement of T):*

- *The Invariance Condition:*
$$T\cap[B]\subseteq R.$$
- *The Convergence Condition:*
$$R\circ\overline{T}=L.$$

To highlight its important properties, we may sometimes refer to an invariant relation as a *reflexive transitive invariant relation*; these two terms refer to the same concept. Note that unlike the definition of invariant assertions (Definition 2) the definition of invariant relations (Definition 3) is not recursive: the term R appears on one side only of each equation. What makes it recursive/ inductive, nevertheless, is the fact that R is reflexive and transitive; reflexivity serves the basis of induction, and transitivity serves the inductive step.

As for the *convergence condition*, it provides that any state in S can be mapped by R onto a state in S that satisfies $\neg t$. Given that R represents the effect of applying the loop body an arbitrary number of times, this condition ensures that these applications eventually produce a final state, i.e. a state that causes the loop to terminate.

To illustrate this concept, we consider again the example of the array sum, presented above, and propose the following invariant relation for it.

$$R=\{(s,s')|x+\sum_{k=i}^{N}a[k]=x'+\sum_{k=i'}^{N}a'[k]\wedge i\leq i'\}.$$

This relation is clearly reflexive and transitive. To check the invariance condition, we compute the intersection $T\cap[B]\cap R$ and check that it equals $T\cap[B]$.

$$\begin{aligned}
& T\cap[B]\cap R\\
= & \qquad \{\text{ Substitution }\}\\
& \{(s,s')|i\neq N+1\wedge x'=x+a[i]\wedge i'=i+1\wedge a'=a\\
& \wedge y'=y+b[j]\wedge j'=j-1\wedge b'=b\wedge x+\textstyle\sum_{k=i}^{N}a[k]\\
& =x'+\textstyle\sum_{k=i'}^{N}a'[k]\wedge i\leq i'\}\\
= & \qquad \{\text{ Substitution }\}\\
& \{(s,s')|i\neq N+1\wedge x'=x+a[i]\wedge i'=i+1\wedge a'=a\\
& \wedge y'=y+b[j]\wedge j'=j-1\wedge b'=b\wedge x+\textstyle\sum_{k=i}^{N}a[k]\\
& =x+a[i]+\textstyle\sum_{k=i+1}^{N}a'[k]\wedge i\leq i'\}\\
= & \qquad \{\text{ Simplification }\}\\
& \{(s,s')|i\neq N+1\wedge x'=x+a[i]\wedge i'=i+1\wedge a'=a
\end{aligned}$$

$$
\begin{aligned}
&\wedge y' = y + b[j] \wedge j' = j-1 \wedge b' = b \wedge x + \textstyle\sum_{k=i}^{N} a[k] \\
&= x + \textstyle\sum_{k=i}^{N} a'[k] \wedge i \leq i'\} \\
= &\qquad\qquad \{ \text{ Logical Simplification } \} \\
&\{(s,s')|i \neq N+1 \wedge x' = x + a[i] \wedge i' = i+1 \wedge a' = a \\
&\wedge y' = y + b[j] \wedge j' = j-1 \wedge b' = b \wedge i \leq i'\} \\
= &\qquad\qquad \{ \text{ Associativity } \} \\
&\{(s,s')|i \neq N+1 \wedge x' = x + a[i] \wedge (i' = i+1 \wedge i \leq i') \\
&\wedge a' = a \wedge y' = y + b[j] \wedge j' = j-1 \wedge b' = b\} \\
= &\qquad\qquad \{ \text{ Logical Simplification } \} \\
&\{(s,s')|i \neq N+1 \wedge x' = x + a[i] \wedge i' = i+1 \\
&\wedge a' = a \wedge y' = y + b[j] \wedge j' = j-1 \wedge b' = b\} \\
= &\qquad\qquad \{ \text{ Substitution } \} \\
&T \cap [B].
\end{aligned}
$$

To check the convergence condition, we compte $R \circ \overline{T}$ and check that it is the full relation.

$$
\begin{aligned}
&R \circ \overline{T} \\
= &\qquad\qquad \{ \text{ Substitution } \} \\
&\{(s,s')|x + \textstyle\sum_{k=i}^{N} a[k] = x' + \textstyle\sum_{k=i'}^{N} a'[k] \wedge i \leq i'\} \circ \{(s,s')|i = N+1\} \\
= &\qquad\qquad \{ \text{ Relational Product } \} \\
&\{(s,s')|\exists s'' : x + \textstyle\sum_{k=i}^{N} a[k] = x'' + \textstyle\sum_{k=i''}^{N} a''[k] \wedge i \leq i'' \wedge i'' = N+1\} \\
= &\qquad\qquad \{ \text{ Simplification } \} \\
&\{(s,s')|\exists s'' : x + \textstyle\sum_{k=i}^{N} a[k] = x'' \wedge i \leq N+1 \wedge i'' = N+1\} \\
= &\qquad\qquad \{ \text{ Simplification/ Interpretation } \} \\
&\{(s,s')|\textbf{true} \} \\
= &\qquad\qquad \{ \text{ Definition } \} \\
&L.
\end{aligned}
$$

3.1.3 Invariant Functions

Definition 4. *Let w be a while statement of the form* `while t do B` *that terminates normally for all initial states in S. We say that a function F on S is an* invariant function *if and only if it is total and*

$$(T \cap [B]) \circ F = T \cap F.$$

To illustrate the concept of invariant function, we consider the array program, and submit the following function:

$$
F\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix} = \begin{pmatrix} a \\ x + \sum_{k=i}^{N} a[k] \\ N+1 \\ a \\ x + \sum_{k=i}^{N} a[k] \\ N+1 \end{pmatrix}.
$$

We briefly verify that this function is invariant with respect to application of the loop body (assuming s satisfies condition t).

$$F\left([B]\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix}\right) = F\begin{pmatrix} a \\ x+a[i] \\ i+1 \\ b \\ y+b[j] \\ j-1 \end{pmatrix} = \begin{pmatrix} a \\ x+a[i]+\sum_{k=i+1}^{N} a[k] \\ N+1 \\ a \\ x+a[i]+\sum_{k=i+1}^{N} a[k] \\ N+1 \end{pmatrix} = \begin{pmatrix} a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \\ a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \end{pmatrix} = F\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix}.$$

3.2 Orderings Invariants

Invariants are not created equal. For example **true** is an invariant assertion for any loop, though not a very useful assertion; the full relation (L) is an invariant relation for any loop, though not a very useful relation; finally a constant function is an invariant function for any loop, though not a very useful function. In this section we briefly review how each type of invariant is ordered.

3.2.1 Ordering Invariant Assertions by Implication

Invariant assertions are naturally ordered by logical implication. As we recall from Definition 2, the invariant assertions are represented by non-empty vectors; hence stronger invariant assertions are represented by smaller vectors. Even though $\alpha =$ **false** does satisfy the condition

$$\{\alpha \wedge t\}B\{\alpha\},$$

we do not consider it as a legitimate invariant assertion because it cannot be represented by a non-empty vector. The inclusion ordering among non empty vectors defines a lattice structure, where the join is the intersection of vectors and the meet is the union of vectors.

3.2.2 Ordering Invariant Relations by Refinement

Invariant relations are ordered by refinement, which, as we have discussed in section 2.2, has lattice-like properties. More refined relations give more information on loop behavior. Because they are by definition reflexive, invariant relations are total. If we consider the definition of refinement (Definition 1), we find that it can be simplified as follows

$$\begin{array}{ll} & RL \cap R'L \cap (R \cup R') = R' \\ \Leftrightarrow & \qquad \{ R \text{ and } R' \text{ are total} \} \\ & L \cap L \cap (R \cup R') = R' \\ \Leftrightarrow & \qquad \{ \text{Set Theory} \} \\ & R \cup R' = R \\ \Leftrightarrow & \qquad \{ \text{Set Theory} \} \\ & R \subseteq R'. \end{array}$$

Hence for invariant relations, refinement is synonymous with set inclusion. We illustrate this ordering with a simple example. We leave it to the reader to check that the following two relations are invariant relations for the array sum program.

$$R_0 = \{(s,s')|x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \wedge a' = a\}.$$

$$R_1 = \{(s,s')|x+\sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k]\}.$$

Invariant relation R_0 refines invariant relation R_1; it also provides more information on loop behavior.

3.2.3 Ordering Invariant Functions by Injectivity

Invariant functions are total by definition, hence they all have the same domain. When we write an equation such as

$$F(s) = F([B](s)),$$

this gives all the more information on B that F is more injective, i.e. partitions its domain finely. If F were a constant, then this equation does not tell us anything about B, since it holds for all B. Hence we order invariant functions by injectivity.

To illustrate this ordering, we consider the following two invariant functions, and we leave it to the reader to check that they are indeed invariant functions for the array sum program:

$$F_0\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix} = \begin{pmatrix} a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \\ a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \end{pmatrix}.$$

$$F_1\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix} = \begin{pmatrix} a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \\ b \\ y+\sum_{k=1}^{j} a[k] \\ i+j-N-1 \end{pmatrix}.$$

To check which (if any) of F_0 and F_1 is more-injective than the other, we compute their nuclei, and find

$$\mu(F_0) = \{(s,s')|a = a' \wedge x+\sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k]\}.$$

$$\mu(F_1) = \{(s,s')|a = a' \wedge x+\sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \wedge b = b' \wedge y+\sum_{k=1}^{j} b[k] = y' + \sum_{k=1}^{j'} b'[k] \wedge i+j = i'+j'\}.$$

Clearly, F_1 is more-injective than F_0.

3.3 Invariants and the Loop Function

In this section we present theorems that relate the invariant assertions, invariant relations and invariant functions to the function of the loop.

3.3.1 Invariant Assertions and Loop Functions

In this section we consider a theorem that provides an interesting link between the function of the loop and an adequate (in a sense to be defined) invariant assertion. We consider the following program structure, which we annotate by intermediate assertions and invariant assertions:

```
f =
begin
{s=s0}
init;
{s=[init](s0)}
while t do
   {F(s)=F(s0) && s in dom(W inter F)}
   B;
{s=F(s0)}
end.
```

A theorem by Mili et al. [10], which is based on earlier findings by Morris and Wegbreit [13], Mills [11] and Basu and Misra [1] provides that program f computes some total function F on S if:

1. The specification Y defined by
$$Y = F\widehat{F} \cap L(\widehat{F \cap W})$$
(where W is the function of the while loop) is total.

2. Segment `init` is correct with respect to specification
$$Y = F\widehat{F} \cap L(\widehat{F \cap W}).$$

3. The following predicate is an invariant assertion:
$$\alpha(s_0, s) \equiv (s_0, s) \in F\widehat{F} \cap L(\widehat{F \cap W}).$$

3.3.2 Invariant Relations and Loop Function

The links between invariant relations and loop functions can be summarized in the following premises:

- If W is the function of the loop then $\mu(W)$ is an invariant relation.
- If R is an invariant relation for w then $R \circ I(\neg t)$ is a lower bound of W.

3.3.3 Invariant Functions and Loop Function

A theorem by Mili et al. [10] provides that the function of the loop is an invariant function of the loop.

4 Relations between Invariants

4.1 Invariant Assertions and Invariant Relations

We have shown in [9] that from an invariant relation we can infer an invariant assertion. We have not found a way to infer an invariant relation from an invariant assertion, because invariant assertions are unary relations whereas invariant relations are binary relations.

4.2 Invariant Relations and Invariant Functions

In [9] we have shown that

- From an invariant function F, we can derive an invariant relation R as the nucleus of F. Such invariant relations are symmetric, in addition to being reflexive and transitive.
- Any invariant relation that is symmetric, in addition to being reflexive and transitive, is the nucleus of an invariant function. For example, the following invariant relation

  $$R = \{(s,s')|x+\sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \wedge a = a'\}.$$

  is the nucleus of the invariant function,

  $$F\begin{pmatrix} a \\ x \\ i \\ b \\ y \\ j \end{pmatrix} = \begin{pmatrix} a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \\ a \\ x+\sum_{k=i}^{N} a[k] \\ N+1 \end{pmatrix}.$$

  whereas the following invariant relation

  $$R = \{(s,s')|x+\sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \wedge i \leq i'\}$$

  is not the nucleus of any invariant function because it is not symmetric.

Contrary to what the names suggest, an invariant function is not an invariant relation that happens to be deterministic. The only relation that is reflexive and deterministic at once is the identity, and the only loop body that accepts the identity as an invariant relation is `skip`, which is of no interest.

4.3 Invariant Assertions and Invariant Functions

Contrary to what the names may suggest, an invariant assertion is not an invariant function that takes boolean values. Rather an invariant function takes the same values before and after application of the loop body whereas an invariant assertion may be false before and become true after. This seems to indicate that a more appropriate name for invariant assertions is *monotonic assertions*. Hence we may distinguish between *monotonic assertions*, which satisfy

$$\{\alpha \wedge t\}B\{\alpha\}$$

(traditionally called invariant assertions) and genuinely invariant assertions, which satisfy

$$\{\alpha \wedge t\}B\{\alpha\}$$

$$\{\neg\alpha \wedge t\}B\{\neg\alpha\}.$$

This distinction is not a mere exercise in hair-splitting. We conjecture (but have not yet proven) that an invariant assertion is usually made up of an invariant part and a monotonic part; and that the invariant part can be derived from invariant functions. Because we have some machinery to derive invariant functions by static code analysis, this may be useful to derive the invariant part of invariant assertions.

5 Brief Concluding Remarks

This is an unfinished work; we intend to explore in detail the characteristics of three types of invariants, and investigate their relations, but there is a lot more to say about these concepts than we could fit in this this paper. We view this line of research as having two broad goals: first broaden the scope of tools that we use to analyze while loops; second, deploy the insights gained from invariant relations and invariant functions to enhance the study of invariant assertions.

Despite the tentative nature of our preliminary conclusions, we are confident of two premises:

- Given an invariant assertion, we cannot use it to derive an invariant relation.
- Given an invariant generation method, we cannot use it to derive an invariant relation generation method.

Because invariant functions and invariant relations appear to be useful in generating invariant assertions, we are exploring means to refine our invariant function and invariant relation generation techniques; we are also interested in analyzuing the structure of invariant assertions, to see which parts can be derived by means of invariant assertions and invariant functions.

References

[1] S.K. Basu and J.D. Misra. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1):76–86, 1975.

[2] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[3] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[4] D. Gries. *The Science of programming*. Springer Verlag, 1981.

[5] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, October 1969.

[6] R.C. Linger, H.D. Mills, and B.I. Witt. *Structured programming*. Addison Wesley, 1979.

[7] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[8] Ali Mili. Reflexive transitive loop invariants: A basis for computing loop functions. In *First International Workshop on Invariant Generation*, Hagenberg, Austria, June 2007.

[9] Ali Mili, Shir Aharon, Chaitanya Nadkarni, Olfa Mraihi, Asma Louhichi, and Lamia Labed Jilani. Reflexive transitive invariant relations: A basis for computing loop functions. *Journal of Symbolic Computation*, 2009.

[10] Ali Mili, Jules Desharnais, and Jean Raymond Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.

[11] H.D. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.

[12] C.C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[13] J.H. Morris and B. Wegbreit. Program verification by subgoal induction. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume II, chapter 8. Prentice Hall, Englewood Cliffs, NJ, 1977.

Refinement and Term Synthesis in Loop Invariant Generation

Ewen Maclean and Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh
Scotland
EH14 4AS
{eahm2,air}@macs.hw.ac.uk

Robert Atkey and Lucas Dixon
Division of Informatics
University of Edinburgh
Edinburgh
Scotland
EH8 9AB
{bob.atkey,lucas.dixon}@ed.ac.uk

Abstract

We present a technique for refining incorrect or insufficiently strong loop invariants in correctness proofs for imperative programs. We rely on previous work [16] in combining program analysis and Proof Planning, and exploit IsaPlanner's use of meta-variables and goal-naming to generate correct loop invariants. We present a simple example in detail and discuss how this might scale to more complex problems.

1 Introduction

Our primary interest is the automatic generation of correct and sufficiently strong loop invariants to allow correctness proofs about imperative programs to succeed. Here we present a simple example for proving exception freedom from array bounds errors. When proving imperative programs correct, sufficiently strong loop invariants must be discovered. Existing tools and techniques for discovering loop invariants are successful for array bound exception freedom proofs – where the access to an array is shown not to fall outside its bounds – but we propose to extend these techniques by using proof failure analysis to refine candidate loop invariants.

This paper outlines a cooperative approach where program analysis systems can be combined with theorem proving in a Higher Order setting. Our hypothesis is that combining the two we can successfully infer correct loop invariants for difficult problems about functional correctness. This is an extension of an idea described for example in [16]. In the work described here we focus on Higher Order Logic but as is described in Section 6, it is being performed in parallel with work in automating proof in Separation Logic [19].

2 Existing Technologies

We describe first existing systems which find candidate loop invariants, and go on to a description of the important features of IsaPlanner – the theorem proving system we adopt.

2.1 Program Analysis Systems

In our work to date we have explored different systems in order to determine the likely loop invariant candidates which would be suggested. Existing techniques for discovering loop invariants roughly fall into four main categories:

Dynamic Approaches This technique runs input templates on the loops and uses techniques such as machine learning to try and work out invariants from the results after the application of the loop. Work of this kind is the Daikon system, which has been successfully integrated in ESC/Java2, as presented in [22].

Top-down Static Approaches [21] presents work in taking the postcondition of a loop, and propagating it through the loop backwards in order to generate candidates for the loop invariant.

Bottom-up Static Approaches The most research has been done on this technique, which works by propagating the precondition of a loop forwards through the loop in order to yield candidates for loop invariants. The various different techniques used are too numerous for this report, but some pertinent techniques are those used in [9] for calculating linear equalities in loop invariants, and in [11] for evaluating program states using difference equations.

Predicate Abstraction Techniques Predicate abstraction is used widely in conjunction with model checking for program verification. It places abstracted predicates at every choice point in the control flow of a program, and refines them as necessary. Predicate abstraction has also been used successfully in the Houdini system [12] to generate loop invariants for ESC/Java.

Constraint-based Methods Possibly the most sophisticated and modern techniques use complex mathematics such as Gröbner bases to calculate non-linear constraints on invariants and then extract actual program invariants. This is demonstrated in [17] for example.

Some of these systems are combined with provers and can verify the correctness of a loop invariant. We are interested in cases where the loop invariant is not strong enough and must be modified.

2.2 IsaPlanner

Proof assistants, such as Isabelle [23], Coq [27] and HOL [13], provide a framework for formalisation tasks such as software verification and mechanised mathematics. Typically, automation is developed by writing programs, called *tactics*, that combine operations from a small trusted kernel. Although many forms of proof automation are already available, developing new tactics and extending existing ones can be difficult. Higher-level concepts, such as search space and heuristic guidance, must be developed on top of the the logical kernel. *Proof Planning* is an approach to providing this kind of high-level machinery in order to encode and apply common patterns of reasoning [6]. Further to this machinery there is a mechanism for recognising common failure patterns and suggesting ways of "patching" the failed proof, which is encapsulated in *Proof Critics*. We refer to the set of critics built-up for a particularly theory as its library of critics.

IsaPlanner [10] is a Proof Planner for the Isabelle proof assistant. The features of the system which are of particular importance for the work in this paper are:

- it provides a programatic way to manage the names of assumptions and goals;
- it efficiently manages the meta-variables which can be shared across different conjectures, and which once instantiated result in several theorems;
- it allows a large collection of automated proof methods to be combined, including counter-example finding, inductive theorem proving with rippling [4], linear arithmetic decision procedures, simplification, and tableaux methods.

We now briefly outline the nature of these features and the arguments that justify finding a solution for them.

2.2.1 Naming Goals and Assumptions

Most tactic based theorem provers represent the assumptions to a theorem (and those to a goal) as a list of terms. A tactic that performs forward reasoning specifies the assumptions on which it works by indices in the list. Similarly, applying tactics to goals typically involves providing an index for the goal. The result is that the list has new elements inserted, and possibly old elements re-arranged. Such an approach has two problems:

- To perform further reasoning on another assumption its position must be re-determined, possibly involving a complete traversal of all assumptions, even if its old position were known. Positions, which are used as names, are not stable over a tactics application. The same problem occurs for subgoals held in a list. Index-based names for referring to goals are not stable over tactic application. Tactics can re-order goals and insert new ones, as well as accidentally solve the goal you planned to tackle with another procedure.
- Having too many assumptions is not a theoretical problem in logics with weakening, as they can be ignored. But from the perspective of automated theorem proving they are problematic. For instance, certain unfortunate assumptions can cause simplification tactics to loop forever, or simply perform the wrong simplification. Some mechanism to say which assumptions should be used is needed.

Structured proof languages, such as Isar and Mizar, provide the means for the user to name assumptions as they interactively write a proof script. However, most proof systems have no such mechanisms for managing the names of assumptions, conjectures, and goals. IsaPlanner is the first system we know of to explicitly give stable names to assumptions and goals, and provide a programatic means for the generation of names to the tactic language. This allows proof machinery to refer to names of assumptions and goals which helps make the Proof Planning techniques applied more robust and simpler to express.

2.2.2 Meta-variables

Most tactic based systems struggle with proof automation when goals contain meta-variables - such as those resulting from existentially quantified variables in the goal or universally quantified ones in assumptions. The inherent logical problem with such variables for proof automation is that the search space suffers an exponential explosion. Furthermore, from a pragmatic point of view, the work involved in managing instantiations is difficult to achieve in a verifiable and efficient manner.

When systems do provide a form of meta-variables, it can only be used within a single proof attempt. Typically, several partial proof attempts cannot be started if they share meta-variables. This is for two main reasons. Firstly, meta-variables are free-variables over a sequent; there is usually no global name space for free/meta-variables - having one would cause several other problems including the inability to perform proof checking or proof search in parallel. The second reason is that meta-variables are not stable over tactic application. Tactics often instantiate old ones, rename them, and introduce new ones. The pragmatic problem with meta-variables is that upon instantiation, every term in which the variable occurs must be updated. Without efficient means to track the dependencies, or lazily instantiate variables, this becomes a slow operation.

IsaPlanner deals with these issues by naming meta-variables in a manner that is stable over tactic application and shared over a collection of proof attempts. Managing dependencies then reduces to operations on a binary relation between two finite sets. The two sets being the variable names and the set of names for assumptions, conjecture and goals. Allowing meta-variables in conjectures also allows synthesis to be treated in a first class manner where a conjecture in refined and modified so that the final result is not abstracted away to an existential variable. This allows the instantiation found to be used in other proof attempts. For instance, this has been used to synthesise a custom induction scheme as part of the inductive proof attempt for proving the correctness of quicksort [5].

2.2.3 Combining Proof Tools

The final feature of IsaPlanner used in this paper is automated proof tools. These include many powerful automatic tactics from Isabelle such as the simplifier, the classical reasoner, counter-example finding, and linear arithmetic [8] as well as the efficient inductive theorem proving tools in IsaPlanner [10].

2.3 Current Automation

At present we are able to automatically generate the verification conditions and generate correct loop invariants. We can prove array bounds exception freedom for some simple programs which manipulate arrays which have no user provided annotation. Section 4 shows a solution of an exception freedom verification problem which has been done completely automatically. Throughout this section when we write "we" we are referring to the reader's interpretation of the automatic proof as produced by IsaPlanner.

Although the problem shown in Section 4 is one which would be solved by most program analysis systems, we describe our methodology and extend it to the more difficult functional correctness problem in Section 5. In this Section the methodology is combined by hand, and we claim no more automation than the verification condition generation. We fully expect however that the proof itself and synthesis problem which is central to the invariant generation are automatable.

3 Verification Condition Generation

Our verification condition generation procedure follows standard Hoare style rules [14] to generate entailments in Higher Order Logic. Initially, we ignore the values of array elements, only concentrating on exception freedom. The Hoare Logic rules for arrays are initially ignored, and assertions are inserted into the code where array elements are accessed to ensure the index is within the requisite bounds. We introduce here the rules we use to generate the weakest-precondition entailment, and give an example program. Extensions to functional correctness are given in Section 5.

3.1 Imperative Language

The version of Java we consider is based on Middleweight Java [3], extended with loops and arrays. This is a cut-down version of Java that retains enough features to demonstrate the properties we require, but is small enough to allow quick experimentation. In the proofs that follow, all types of variables are implicitly integers since this is the type given in the programming languages. For this reason we write $i+1$ instead of $s(i)$ since this is more pertinent to the inductive structure of the natural numbers.

3.2 Exception Freedom Rules

We use the standard notion of Hoare Triples to generate entailments for correctness proofs. In the case of verification of array bounds correctness – i.e. that no array index is out of the bounds of the array – we are not concerned with the value of array elements. Figure 1 shows the set of Hoare Logic rules that we use. Importantly, the only rule which requires creative input is the rule for the `while` construct in our imperative language since a loop invariant must be discovered.

4 Example Problem

Figure 2 shows a program which makes a copy of an array with the elements reversed. We use a weakest precondition analysis to generate the set of hypotheses and conclusions shown in Figure 3. Here $\mathscr{X}$ stands for a meta-variable which depends on the free variables in the hypotheses and conclusions. This is equivalent to an existentially quantified higher-order variable standing for the function which describes

$$\frac{}{\{P[E/x]\}x := E\{P\}} \qquad \frac{\{P\}C\{Q\} \quad \{Q\}D\{R\}}{\{P\}C;D\{R\}}$$

$$\frac{\{Cond \wedge P\}C\{Q\} \quad \{\neg Cond \wedge P\}D\{Q\}}{\{P\}\texttt{if}\ Cond\ C\ \texttt{else}\ D\{Q\}} \qquad \frac{\{Cond \wedge P\}C\{P\}}{\{P\}\texttt{while}\ Cond\ C\{\neg Cond \wedge Q\}}$$

Figure 1: The Hoare Rules used to Generate Verification Conditions

the loop invariant. For example, a natural candidate for the loop invariant would be

$$\mathscr{X} \equiv \lambda a, i.\ i \geq 0 \wedge i < len(a)$$

since this describes the limits of the loop variable imposed in the code, noticing that the index is incremented each pass through the loop. At this point we are disregarding the pre and post conditions since we are interested purely in exception freedom.

```
 procedure revArray(int[ ] a) {
{α_0 = elements(a, 0, len(a))}
    int[ ] b = new int[a.length];
    int i = 0;
    while (i! = a.length){
        b[i] = a[a.length − (i + 1)];
        i = i + 1;
    }
{elements(a, 0, len(a)) = α_0 ∧ elements(b, 0, len(b)) = rev(α_0)}           }
```

Figure 2: A procedure for copying an array with the elements reversed

We consider proving that there are no array bounds errors in the loop of the program shown in Figure 2. This verification has been performed in IsaPlanner and the proof is totally automatic thanks to the mechanisms described in Section 2.2, using Isabelle's built in `auto` tactic and its counter-example checker. We choose this particular verification in order demonstrate the invariant generation techniques we develop. Existing systems such as ESC/Java2 [22] and Houdini [12] are already capable of proving this program free from array bounds exceptions. We discuss a more complicated example in Section 5.

We imagine first of all that no loop invariant has been given and that it is represented by a meta variable $\mathscr{X}$ as described in Section 4. After a weakest precondition analysis, we perform Isabelle's `safe` tactic which reduces the goal to a set of hypotheses and conclusions as shown in Figure 3.

We omit some details of the result of the verification generation process, and present slightly simplified versions for presentation clarity. While we are just proving the correctness of the loop, we can assume $len(a) = len(b)$ since this is set at the start of the code when the array b is allocated. At the assignment of an array cell, the variable is changed and an extra hypothesis is added ensuring that the length of the array is unchanged. This is represented by $len(b') = len(b'')$ in the code, which is a simplification of a full description of array update as applied in Section 5. In the presentation, b' is the state of array b at some point at the start of the loop, and b'' is the state of array b at the end of the loop code.

h1 : $\mathscr{X}(a,b',i)$	c1 : $i \geq 0$
h2 : $i \neq len(l)$	c2 : $i < len(b')$
h3 : $len(a) = len(b)$	c3 : $len(a) - (i+1) \geq 0$
h4 : $len(b') = len(b'')$	c4 : $len(a) - (i+1) < len(a)$
	c5 : $\mathscr{X}(a,b'',i+1)$

Figure 3: The Verification Conditions Generated with no Loop Invariant

4.1 Dependent Bi-Abduction

[7] presents a method for automatically annotating programs with pre and post conditions using a method known as *bi-abduction*. This is focussed on verification of programs which use pointers, and uses separation logic to verify properties about the shape of inductive structures on the heap (see Section 6).

Abduction is a proof-technique which determines what must be true in hypotheses in order to render a conclusion correct. For example, consider the following theorem:

$$i, j, n : \mathbb{N}.\ j \leq n \vdash n - (i+j) \geq 0 \wedge n - j \geq 0$$

The conclusion $n-(i+j) \geq 0$ cannot be proven from the hypotheses. In order for the proof to go ahead, we must add the hypothesis that $i \leq n - j$. Determining this extra hypothesis is a form of abduction.

The premise behind bi-abduction is that it should be possible, by program analysis, to determine not only hypotheses which allow verification proofs to succeed, but also to determine extra conclusions which can be proved from the hypotheses. In the case of the work presented in [7] for example, this happens in program verification where such extra hypotheses pertain to a part of memory which is not passed to a procedure. The extra conclusions which are added pertain to parts of memory which must exist in order for post-conditions of procedures to be satisfied.

In our work we see a parallel between this work, and the synthesis and refinement of loop invariants. When using a proof system to determine a correct loop invariant, we need to both use it as part of the hypotheses and in the conclusion. In order to prove the conclusions correct we can use abduction to determine what must be true of the loop invariant. However in adding extra hypotheses through instantiation of a loop invariant we also add extra conclusions. As a result this is bi-abduction, since we are determining new hypotheses and conclusion, but we observe dependencies between them.

4.2 Refinement

As discussed in [15], Proof Planning techniques can be used to analyse failure of a verification in order to discover correct loop invariants. We consider here the verification conditions shown in Figure 2, imagining that we are trying to prove the program correct given an insufficiently strong suggestion for a loop invariant – i.e. one which contains some necessary conditions, but not all of them.

IsaPlanner has a critics mechanism built-in as described in Section 2.2. Although some of the refinements described here have been done interactively, the proposal is to extend the library of critics to analyse the failure patterns we discuss here. This is an extension of the work described for example in [16].

4.2.1 Invariant Candidate Refinement by Addition of Conjuncts

Imagine that we have been given the suggestion

$$\mathscr{X} \equiv \lambda a, i.\ i \geq 0 \wedge i \leq len(a)$$

which creates the set of hypotheses and conclusions shown in Figure 4. When IsaPlanner attempts to prove this set of verification conditions it is capable of annotating which of the goals it has not been possible to prove. This report is motivated by the use of a counter-example checker, as described in Section 2.2. Each conclusion in Figure 4 is marked with a tick or a cross denoting whether it is provable from the hypotheses.

`h1` :$i \geq 0$	`c1` :$i \geq 0$	✔
`h2` :$i \leq len(a)$	`c2` :$i < len(b')$	✘
`h3` :$i \neq len(a)$	`c3` :$len(a) - (i+1) \geq 0$	✔
`h4` :$len(a) = len(b)$	`c4` :$len(a) - (i+1) < len(a)$	✔
`h5` :$len(b') = len(b'')$	`c5` :$i+1 \geq 0$	✔
	`c6` :$i+1 \leq len(a)$	✔

Figure 4: The Verification Conditions Generated with Loop Invariant Suggestion (1)

In this case we analyse the failure pattern of the conclusions and see that the failing goal is $i < len(b')$. A naive patch would be to add this to the hypotheses by means of extending the loop invariant. This would involve instantiating the loop invariant to

$$\mathscr{X} \equiv \lambda a, i.\ i \geq 0 \wedge i \leq len(a) \wedge i < len(b).$$

This results in divergence since the set of generated verification conditions include the conclusion $i+1 < len(b)$ which is included by the array access assertion conditions. Analysing this divergence is similar to the fixed point analysis presented in [20], and the divergence critic developed and discussed in [28].

Our solution is to augment the existing loop invariant candidate with an extra uninstantiated conjunct. We are motivated to do this in this case noticing that the existing loop invariant does not contain any relation between the variables which exist within the failed conclusion. Accordingly we write

$$\mathscr{X} \equiv \lambda a, b, i.\ i \geq 0 \wedge i \leq len(a) \wedge \mathscr{Y}(a, b, i)$$

which renders the verification complete given IsaPlanner's term synthesis machinery as described in Section 4.3.

4.2.2 More Complicated Invariant Candidate Refinement

Imagine now that we have been given the flawed invariant candidate

$$\mathscr{X} \equiv \lambda a, i.\ i \geq 0 \wedge i < len(a)$$

based purely on the initial inspection that all accesses to array elements should be within the bounds of the arrays. Given this loop invariant IsaPlanner produces the set of verification conditions shown in Figure 5.

Now inspecting the failed conclusions we see that $i+1 < len(a)$ and $i < len(b')$ cannot be solved. This time we have a failing conclusion which is not independent of the existing conjuncts in the loop invariant. We therefore identify the failing conjunct within the loop invariant to be $i < len(a)$. We want to replace this with a more general term, so we insert a corresponding meta-variable with arguments a, i. We also want to add an extra conjunct as described above to include the extra variable b. We thus remove the conjunct $i < len(a)$ and add the two uninstantiated conjuncts

$$\mathscr{Y}(a, i) \qquad\qquad \mathscr{Z}(a, i, b)$$

h1 :$i \geq 0$	c1 :$i \geq 0$	✔
h2 :$i < len(a)$	c2 :$i < len(b')$	✘
h3 :$i \neq len(a)$	c3 :$len(a) - (i+1) \geq 0$	✔
h4 :$len(a) = len(b)$	c4 :$len(a) - (i+1) < len(a)$	✔
h5 :$len(b') = len(b'')$	c5 :$i+1 \geq 0$	✔
	c6 :$i+1 < len(a)$	✘

Figure 5: The Verification Conditions Generated with Loop Invariant Suggestion (1)

Since $\mathscr{X}$ subsumes $\mathscr{Y}$ we adjust the loop invariant to

$$\mathscr{X} \equiv \lambda a, b, i.\ i \geq 0 \wedge \mathscr{X}(a, i, b).$$

which renders the verification complete given IsaPlanner's term synthesis machinery.

4.2.3 Counter-Example Driven Refinement

When a verification attempt fails, IsaPlanner uses the built-in Isabelle counter-example checker to give values of variables for which the verification fails. This helps in giving suggestions for how the invariants should be refined. IsaPlanner exploits its nominals to identify which goals have failed. We can analyse the counter-example given. For example, in some cases the counter-example found involves mismatching list lengths, so we can automatically add len to the term synthesis machinery.

4.3 Term Synthesis

Given a refined loop invariant which has meta-variables inserted as described above, we can exploit IsaPlanner's term synthesis mechanism to generate and test loop invariant candidates. The set of function and constant symbols that we add to the term synthesis machinery are

$$0 \quad 1 \quad \lambda x.len(x) \quad \lambda x, y.x \leq y \quad \lambda x, y.x = y \quad \lambda x, y.x < y \quad \lambda x, y.x \wedge y$$

The current machinery for term synthesis combines terms using meta-variables until a ground term is found, which is then checked using the counter-example checker. Once a term is constructed for which a counter-example is not found, the proof is attempted using IsaPlanner and Isabelle's `auto` tactic.

This yields a correct loop invariant automatically in IsaPlanner. The process of narrowing the possible function symbols down to those shown above is currently hard-coded. In general choosing a complete set of such functions automatically is very difficult, but we aim to address this problem. Since we use breadth-first search, this is guaranteed to find a correct candidate if one exists.

5 Functional correctness

We extend the techniques described up to this point to verify the functional correctness of the program shown in Figure 2. This requires an extension of the verification condition generation to array assignment, and a language for proving the resulting entailment from the weakest precondition analysis.

5.1 Additional Rules

The rule for array assignment is given as

$$\frac{}{\{P[upd(a,i,E)/a]\}a[i] := E\{P\}}$$

and all occurrences of array lookup are replaced by the function *ele*. The function *elements* allows inductive decomposition of an array so we can reason about the structure of the data. This mimics the linked list structures found when reasoning about pointer programs [29, 2]. Extending Separation Logic to account for arrays is briefly discussed further in Section 6. Rewrite rules augmented with rippling annotations, known as wave rules, are given for these in Figure 6.

$$
\begin{array}{rcll}
ele(upd(X,Y,Z),Y) & \Rightarrow & Z & (1)\\
\neg Y = N \rightarrow ele(\boxed{upd(\underline{X},Y,Z)}^{\uparrow},N) & \Rightarrow & ele(X,N) & (2)\\
elements(N,X,0) & \Rightarrow & nil & (3)\\
X+(Y+1) \leq len(A) \rightarrow elements(A,X,\boxed{\underline{Y}+1}^{\uparrow}) & \Rightarrow & \boxed{ele(A,X) :: \underline{elements(A,\boxed{\underline{X}+1}^{\uparrow},Y)}}^{\uparrow} & (4)\\
X+(Y+1) \leq len(A) \rightarrow elements(a,X,\boxed{\underline{Y}+1}^{\uparrow}) & \Rightarrow & \boxed{\underline{elements(A,X,Y)} <> ele(A,X+N)}^{\uparrow} & (5)\\
rev(nil) & \Rightarrow & nil & (6)\\
rev(\boxed{X <> \underline{h :: nil}}^{\uparrow}) & \Rightarrow & \boxed{h :: \underline{rev(X)}}^{\uparrow} & (7)\\
nil <> t & \Rightarrow & t & (8)\\
h :: l <> t & \Rightarrow & \boxed{h :: \underline{l <> t}}^{\uparrow} & (9)\\
elements(X,0) = nil & \Leftrightarrow & len(X) = 0 & (10)\\
X + \boxed{\underline{Y}+Z}^{\uparrow} & \Rightarrow & \boxed{\underline{X+Y}+Z}^{\uparrow} & (11)\\
\boxed{\underline{X-(\boxed{\underline{Y}+Z}^{\uparrow})}+Z}^{\uparrow} & \Rightarrow & X-Y & (12)
\end{array}
$$

Figure 6: Wave Rules Used in the Functional Verification

5.2 Entailment With Unknown Loop Invariant

When proving the functional correctness of the program, there are three stages to consider, all of which rely on a correct loop invariant. We represent the unknown loop invariant as usual with the meta-variable $\mathscr{X}$. This makes the bi-abduction more complicated than the example shown in Section 4. We must determine that the loop invariant follows from the pre-condition and code before the loop; that the loop invariant is indeed invariant; finally that the post-condition follows from the loop invariant and code after the loop. These three entailments are combined into the set of hypotheses and conclusions shown in Figure 7.

`h1` :$elements(a,0,len(a)) = \alpha_0$	`c1` :$\mathscr{X}(0,a,b)$
`h2` :$len(a) = len(b)$	`c2` :$\mathscr{X}(i+1,a,b')$
`h3` :$\mathscr{X}(i,a,b)$	`c3` :$elements(a) = \alpha_o$
`h4` :$b' = upd(b,i,ele(a,len(a)-(i+1)))$	`c4` :$elements(b,0,len(b)) = rev(\alpha_0)$
`h5` :$\neg i = len(a)$	
`h6` :$\mathscr{X}(len(a),a,b)$	

Figure 7: Verification Conditions with an Uninstantiated Loop Invariant

5.3 Program Analysis Suggestion for Loop Invariant

A standard interpretation of how to generate a loop invariant for such a program is to create an invariant by analysing the "work done" and "work left to do" in a loop. In the case of the program we are analysing, we can suggest that the update to b has been done up to some point between $i = 0$ and $i < len(a)$, since i is incremented at each stage. A good initial candidate then assumes that the array b is updated up to, but not including, the value of i:

$$\mathscr{X} \equiv \lambda i,a,b.\ i \geq 0 \ \wedge\ i \leq len(a) \wedge \forall j.0 \leq j < i \ \rightarrow\ ele(b,j) = \mathscr{Y}(a,b,j,i)$$

This yields a set of hypotheses and conclusions as shown in Figure 8.

`h1` :$elements(a,0,len(a)) = \alpha_0$	`c1` :$\forall j.0 \leq j < 0 \ \rightarrow\ ele(b,j) = \mathscr{Y}(a,b,j,0)$
`h2` :$len(a) = len(b)$	`c2` :$\forall j.0 \leq j < i+1 \ \rightarrow\ ele(b',j) = \mathscr{Y}(a,b',j,i+1)$
`h3` :$\forall j.0 \leq j < i \ to\ ele(b,j) = \mathscr{Y}(a,b,j,i)$	`c3` :$elements(a,0,len(a)) = \alpha_o$
`h4` :$b' = upd(b,i,ele(a,len(a)-(i+1)))$	`c4` :$elements(b,0,len(b)) = rev(\alpha_0)$
`h5` :$\neg i = len(a)$	`c5` :$0 \geq 0$
`h6` :$\forall j.0 \leq j < len(a) \ to\ ele(b,j) = \mathscr{Y}(a,b,j,len(a))$	`c6` :$0 \leq len(a)$
`h7` :$i \geq 0$	`c7` :$i+1 \geq 0$
`h8` :$i \leq len(a)$	`c8` :$i+1 \leq len(a)$

Figure 8: Verification Conditions with Partial Instantiation

5.4 Term Synthesis

We study the case where the set of verification conditions shown in Figure 8 is the initial state for the synthesis attempt. We use the post-condition, program analysis, and the set of function symbols available to create a large set of possible term symbols over which to instantiate the possible loop invariant. The set of possible term symbols is

$$0 \quad 1 \quad \lambda x,y.x+y \quad \lambda x,y.x-y \quad \lambda x,y.x=y \quad \lambda x,y.x \wedge y \quad \lambda x.len(x) \quad \lambda x,y.ele(x,y)$$

which will generate the correct loop invariant:

$$\mathscr{X} \equiv \lambda i,a,b.\ i \geq 0 \ \wedge\ i \leq len(a) \wedge \forall j.0 \leq j < i \ \rightarrow\ ele(b,j) = ele(a,len(a)-(j+1)).$$

5.5 Proof

Given an instantiation of

$$\mathscr{Y} = \lambda a,b,i,j. \equiv ele(a,len(a)-(j+1))$$

the interesting and non-trivial conjuncts to prove are `c2` and `c4` in Figure 8. We do not discuss here the actual proof in IsaPlanner, or the reasoning techniques which must be employed. Our interest is in the correct synthesis of a loop invariant. The proofs that follow are all automtable in IsaPlanner.

5.5.1 Goal `c2`

After some simplification for clarity of presentation, the entailment resulting from a weakest precondition analysis is

$$\begin{gathered} elements(a,0,len(a)) = \alpha_0 \ ,i < len(a), i \geq 0 \\ \forall j.0 \leq j < i \ \rightarrow\ ele(b',j) = ele(a,len(a)-(j+1)) \vdash \\ \forall j.0 \leq j < i+1 \ \rightarrow\ ele(upd(b',i,ele(a,len(a)-(i+1))),j) = ele(a,len(a)-(j+1)). \end{gathered}$$

We now perform first a case split on i, proving a branch where $i = 0$ and a branch where $i > 0$. The first branch leads with $i = 0$ to

$$\begin{array}{c} elements(a,0,len(a)) = \alpha_0 \ , i < len(a), i = 0 \\ \forall j. 0 \leq j < 0 \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j. 0 \leq j < 1 \ \rightarrow \ ele(upd(b', 0, ele(a, len(a) - 1)), j) = ele(a, len(a) - (j+1)) \end{array}$$

which reduces to

$$\begin{array}{c} elements(a,0) = \alpha_0 \ , i < len(a), i = 0, j = 0 \\ \texttt{true} \vdash \\ ele(upd(b', 0, ele(a, len(a) - 1)), 0) = ele(a, len(a) - 1) \end{array}$$

which reduces to an identity with application of rule (1). In the second branch where $i > 0$ we obtain

$$\begin{array}{c} elements(a,0) = \alpha_0 \ , i < len(a), i > 0 \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j. 0 \leq j < i+1 \ \rightarrow \ ele(upd(b', i, ele(a, len(a) - (i+1))), j) = ele(a, len(a) - (j+1)) \end{array}$$

for which we now split up the conclusion into cases where $0 \leq j < i$ and $j = i$. This yields firstly the goal

$$\begin{array}{c} elements(a,0) = \alpha_0 \ , i < len(a), i > 0 \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(\boxed{upd(\underline{b'}, i, ele(a, len(a) - (i+1)))}^{\uparrow}, j) = ele(a, len(a) - (j+1)) \end{array}$$

for which rule (3) applies since we can prove that $0 \leq j < i \ \rightarrow \ j \neq i$. This reduces to

$$\begin{array}{c} elements(a,0) = \alpha_0 \ , i < len(a), i > 0 \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)). \end{array}$$

For the case where $j = i$ we yield

$$\begin{array}{c} elements(a,0) = \alpha_0 \ , i < len(a), i > 0 \\ \forall j. 0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ ele(upd(b', i, ele(a, len(a) - (i+1))), i) = ele(a, len(a) - (i+1)) \end{array}$$

which using rule (1) reduces to an identity.

5.5.2 Goal `c4`

The goal we need to prove is

$$\begin{array}{c} elements(a,0,len(a)) = \alpha_0, \ \ len(a) = len(b) \\ \forall j. 0 \leq j < len(a) \ to \ ele(b, j) = ele(a, len(a) - (j+1)) \vdash \\ elements(b, 0, len(b)) = rev(\alpha_0). \end{array}$$

We write this in a simpler form as

$$\forall j. 0 \leq j < len(a) \ \rightarrow \ ele(b,j) = ele(a, len(a) - (j+1)) \vdash$$
$$elements(b, 0, len(a)) = rev(elements(a, 0, len(a))$$

in order to prove this, we generalise the theorem. The Generalisation of the theorem to segments within the array is described by the diagram in Figure 5.5.2. This sort of generalisation is possible within IsaPlanner, and can be done automatically since the class of induction is bounded. This means that in the theorem the induction variable is constrained to a finite range. Once generalised we can set up an inductive proof on the length of the segment.

$$\forall j. 0 \leq j < len(a) \ \rightarrow \ ele(b,j) = ele(a, len(a) - (j+1)) \vdash$$
$$\forall m,n. len(a) < m \geq 0 \ \wedge \ len(a) \leq n \geq 0 \ \wedge \ len(a) \leq m+n \geq 0 \rightarrow$$
$$elements(b, len(a) - (m+n), n) = rev(elements(a, m, n))$$

This can be proved by IsaPlanner by induction on n and with subsequent case-splits on $s(n) > 0 \vee s(n) = 0$. In the case where $s(n) = 0$ the conclusion in trivial, so we concentrate here on the step case when we know that $s(n) > 0$. For ease of presentation we omit the precedents to the implications. This complicates the proof, but IsaPlanner is capable of dealing with this sort of problem in inductive proof. We now have the goal

$$\forall j.\ 0 \leq j < len(a) \ \rightarrow \ ele(b,j) = ele(a, len(a) - (j+1)) \qquad \dagger$$
$$\forall m'. elements(b, len(a) - ('m+n), n) = rev(elements(a, m', n)) \ \vdash$$
$$elements(b, len(a) - (m + \boxed{n+1}^{\uparrow}), \boxed{n+1}^{\uparrow}) = rev(elements(a, m, \boxed{n+1}^{\uparrow}))$$

We apply rules (4),(6) on the left and right of the equality to obtain the conclusion

$$\boxed{ele(b, len(a) - (m+n+1)) :: \boxed{elements(b, \boxed{\boxed{len(a) - (m + \boxed{n+1}^{\uparrow})} + 1}^{\uparrow}, n)}}^{\uparrow} =$$
$$rev(\boxed{\boxed{elements(a, m, n)} <> ele(a, m+n)}^{\uparrow})$$

Now applying rule (7),(11) and (12) we obtain

$$\boxed{ele(b, len(a) - (m+n+1)) :: \boxed{elements(b, len(a) - (m+n), n}}^{\uparrow} =$$
$$\boxed{ele(a, m+n) :: \boxed{rev(elements(a, m, n))}}^{\uparrow}$$

at which point the induction hypothesis applies leaving us to prove the equality

$$ele(b, len(a) - (m+n+1)) = ele(a, m+n)$$

which we can solve using hypotheses †. The hypothesis applies since we are in a branch where $n+1 > 0$.

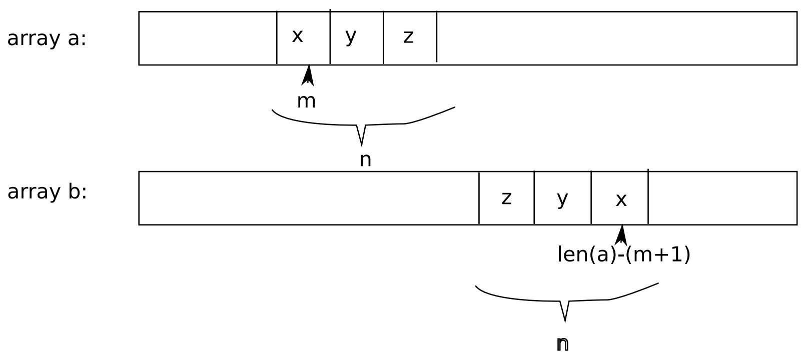


Figure 9: The generalised form of goal c4 pertaining to array segments

5.6 Middle-Out Reasoning

As described in [18] and [26] for example, middle out reasoning can be applied in these situations to instantiate meta-variables such as $\mathcal{Y}$ during the progress of the proof. Our approach here differs in that a term is constructed and tested in order to complete the proof. The key of the work is to make sure that the set of possible constant and function symbols is small, and that the eventual term which must be synthesise is minimal.

6 Separation Logic And Future Work

Separation Logic as described in [25] for example, is a substructural logic which is based on the Logic of Bunched Implications [24] and has been extended with Hoare Logic rules to describe manipulation of data on the heap. It is primarily designed for pointer programs where aliasing and resource management have traditionally been a difficult problem. We intend to extend our work on arrays by exploiting existing and ongoing work on functional correctness proofs in pointer programs, to treat arrays as indexed data structures on the heap.

The advantage of separation logic is that it exploits a semantics which introduces a separating conjunction which is associative-commutative and does not allow weakening or contraction. This means that one can ensure memory safety properties. Since we are interested in functional correctness we exploit the existing tools such as Smallfoot [1] to indicate how a proof proceeds when the values of data cells are not the focus of the proof. Arrays introduce a new problem, which is that the separating conjunction must be indexed to describe allocation and deallocation of array elements. Reynolds calls this the *Iterated Separating Conjunction*. This is denoted by an expression of the form $\odot^{n}_{exp=m} p(exp)$ where m and n are indices to the array. In this formalisation *emp* is given to mean the empty heap.

In the formalisation demonstrated here, since we are using IsaPlanner in conjunction with Isabelle/HOL, we have extended the logic with the function symbols *elements* and *ele* which describe the data within the array. The inductive definition of *elements*, given by equations (4) and (6), is comparable to the construction of the iterated separating conjunction whose definitions. The main difference is that instead of start and end markers for the array indices in the case of Separation Logic, we have a start marker and then a number of subsequent array cells. Figure 10 shows the equivalences between the definitions, and demonstrates that the Iterating Separating Conjunction approach is more expressive. The equivalent statement of the conclusion for example for goal c4 then becomes

$$\odot^{len(a)-(m+1)}_{i=len(a)-(m+n)} b(i) = rev(\odot^{m+n-1}_{i=m} a(i)).$$

This definition is more elegant, and the underlying separating logic allows for a important safety aspects of arrays to be proved. In particular we can use the Iterating Separating Conjunction to reason about the size of arrays. For functional correctness we need to define new rules for the manipulation of the arrays. This means we will be adding terms to the *pure* part of the goal – i.e. that which does not contain any non-classical connectives. For example, a possible recursive definition of the *rev* function applied to the iterated separating conjunction would be:

$$rev(\odot_{i=m}^{m+s(n)} p(i)) \iff q(m) * \odot_{i=m}^{m+n} p(i) \wedge q(m) = p(m+s(n))$$

Here the *pure* part is what follows after the classical conjunction. After the heap assertions have been satisfied we will be left with a pure residue which is similar to the reasoning we have presented in Section 5. Other Proof Planning techniques such as rippling extend to Separation Logic formulae.

$m > n \rightarrow \odot_{i=m}^{n} p(i) \leftrightarrow emp$	$n \leq 0 \rightarrow elements(p,m,n) = nil$
$m = n \rightarrow \odot_{i=m}^{n} p(i) \leftrightarrow p(m)$	$elements(p,n,1) = ele(p,n)$
$k \leq m \leq n+1 \rightarrow \odot_{i=k}^{n} p(i) \iff (\odot_{i=k}^{m-1} p(i) * \odot_{i=m}^{n} p(i))$	$n+k+m \leq len(a) \rightarrow elements(a,m,n+k) = elements(a,m,n) <> elements(a,m+n,k)$
$\odot_{i=m}^{n} p(i) \iff \odot_{i=m-k}^{n-k} p(i+k)$	

Figure 10: A comparison between the Iterating Separation Conjunction and Our Approach

7 Conclusion

The approach presented here is the initial findings of our work on loop invariant generation techniques for functional properties of programs. We exploit existing work on Program Analysis to collect good candidates for loop invariants. These are often insufficient and need to be modified. In order to do this we use a refinement technique and make use of IsaPlanner's treatment of meta-variables and its ability to name hypotheses and goals.

Although the work is incipient we believe the success of the approach at verifying array bounds exception freedom will scale to functional correctness problems. In particular this work progresses in parallel with other work on verifying functional correctness of programs which manipulate pointers.

References

[1] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137. Springer, 2006.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS 2005*, volume 3780, pages 52–68. Springer-Verlag, 2005.

[3] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. Mj: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, 2003.

[4] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, April 2005.

[5] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. In S. Allen, J. Crossley, K.K. Lau, and I. Poernomo, editors, *Proceedings of the ETAPS-05 Workshop on Constructive Logic for Automated Software Engineering (CLASE-05), Edinburgh*, pages 4–18. LFCS University of Edinburgh, 2006. Appears in ENTCS 2006.

[6] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *JAR*, 7(3):303–324, 1991.

[7] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[8] Amine Chaieb. Parametric linear arithmetic over ordered fields in isabelle/hol. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 246–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[10] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[11] Bernard Elspas, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman. An assessment of techniques for proving program correctness. *ACM Computing Surveys*, 4(2):97–147, June 1972.

[12] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[15] A. Ireland. Towards automatic assertion refinement for separation logic. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2006.

[16] A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[17] Neil D. Jones and Xavier Leroy, editors. *Non-linear loop invariant generation using Gröbner bases.* ACM, 2004.

[18] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[19] E. Maclean, A. Ireland, and I. Hind. Proving functional properties in separation logic. In *Automated Reasoning Workshop – Bridging the Gap Between Theory and Practice*, 2008.

[20] S. Magill, A. Nanevski, E. Clarke, and L. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Proceedings of the Third Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE '06)*, pages 47–60, 2006.

[21] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91(5):233–244, 2004.

[22] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[23] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[24] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[25] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[26] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998.

[27] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004.

[28] T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[29] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS*, 2002.

Generalisation of Induction Formulae based on Proving by Symbolic Execution

Angela Wallenburg[*]
Chalmers University of Technology and University of Gothenburg, Sweden
angelaw@chalmers.se

Abstract

Induction is a powerful method that can be used to prove the total correctness of program loops. Unfortunately the induction proving process in an interactive theorem prover is often very cumbersome. In particular it can be difficult to find the right induction formula. We describe a method for generalising induction formulae by analysing a symbolic proof attempt in a semi-interactive first-order theorem prover. Based on the proof attempt we introduce universally quantified variables, meta-variables and sets of constraints on these. The constraints describe the conditions for a successful proof. By the help of examples, we outline some classes of problems and their associated constraint solutions, and possible ways to automate the constraint solving.

1 Introduction

Induction is a powerful proof method that can be used to prove that a property holds for a possibly infinite sequence of elements. In many program verification applications this is an essential tool. An excellent example is in proving the total correctness of loops, which is the focus of this paper. A standard induction proof requires proving that a property, the *induction formula*, holds for the first element in a sequence and then proving that if it holds for an arbitrary element in the sequence, then it holds for the next one. Though this basic idea of induction may seem straightforward, in a practical program verification setting induction is notoriously difficult to use and to automate.

A central complication is the degree of incompleteness that occurs in practice; that is, for a given property it is common that an inductive proof attempt fails even if the property is valid. To overcome this problem we can prove a property that is stronger than the one originally desired and then prove that the stronger property implies the original proof obligation, i.e. that it is a *generalisation*. We use a rule that we call natInduct both to prove that the (stronger) induction formula ϕ is valid *and* to prove that if ϕ holds, then the original proof obligation Δ holds:

$$\mathsf{natInduct}\ \frac{\Gamma \Longrightarrow \phi(0),\, \Delta \qquad \Gamma \Longrightarrow \forall n.\, 0 \leq n \wedge \phi(n) \rightarrow \phi(n+1),\, \Delta \qquad \Gamma,\, \forall n.\, 0 \leq n \wedge \phi(n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

In the calculus that we use $\alpha,\ldots,\gamma \Rightarrow \delta,\ldots,\varphi$ has the semantics $\alpha \wedge \ldots \wedge \gamma \rightarrow \delta \vee \ldots \vee \varphi$. The starting assumption is Γ and $\Longrightarrow$ is called the sequent arrow. For an early reference to sequent calculus, see [10], and for the sequent calculus that is the foundation of our system, see [11]. Note the presence of the induction formula, ϕ, in all three branches of an inductive proof: 1) in the *base case*: $\phi(0)$, 2) in the *step case*: $\forall n.0 \leq n \wedge \phi(n) \rightarrow \phi(n+1)$, and 3) in the *use case*: $\forall n.0 \leq n \wedge \phi(n) \rightarrow \Delta$. Any change to the induction formula propagates to all branches of the proof. In particular a generalisation of the induction formula both introduces more assumptions and—simultaneously and in different branches of the tree—increases the proof burden. Therefore a patch to the induction formula based on the information from one failed proof branch is likely to cause another branch to fail. A further problem—particularly for a real programming language with exceptions, aliasing etc.—is that the reasoning in all the "straight stretches" in between the induction-specific points of the proof is also non-trivial and has to be repeated for every failed proof attempt, which in an interactive verification system usually involves manual steps.

[*]Now at Praxis High Integrity Systems Ltd, Bath, U.K.

This paper presents an idea for finding the right induction formula by generalising the original proof obligation based on information from failed proof attempts. First we analyse the state changes during symbolic execution of the loop body to add suitable universally quantified variables and placeholders for parts of the induction formula, typically the postcondition. Next we make a proof attempt with the modified formula, the induction formula guess. Our induction proving process is syntactically driven and consists of a number of well-defined steps, all of which are necessary for a successful proof. Every open proof goal in the resulting proof attempt therefore generates a constraint on the values of the placeholders introduced in our induction formula guess. We know that if the constraints can be solved, then the proof will succeed. While solving the generated second-order constraints is undecidable in the general case, there are a number of classes of problems where the solutions are simple enough to be candidates for automatic constraint solving techniques.

Section 2 describes the verification system properties that our method builds on. Section 3 gives a motivating example, which illustrates the induction proving process, and the proposed constraint generation method. Section 4 describes the constraint generation method on a general level. Sections 5 and 6 give two additional examples of the constraint generation. While solving the generated constraints is beyond the scope of this article, Section 7 gives an overview of how the constraints can be solved by hand, and of some classes of possible automatic solutions. We then describe some limitations of the method and possible future work, some related work on induction formula generation and loop invariant generation, and finally provide a summary and some conclusions.

2 Verification Setting

In this section we describe some important properties of the verification system that our method is developed for. A typical formula in this system is written as:

$$\forall\, vars.(precondition \rightarrow \{state\ description\}\ \langle\texttt{program}\rangle\ postcondition)$$

and it means that if the precondition holds, and if the program is executed from the starting state described within the curly brackets, then it terminates in a state where the postcondition holds.

The calculus that we use can be described as a variant of weakest precondition calculus. The underlying logic is a version of dynamic logic (DL) [14]. It differs from the original DL in that there is a distinction between logical variables and program variables. Only logical variables can be quantified over, for example *vars* in the typical formula above have to be logical variables.

Furthermore, the calculus calculates the substitutions with respect to an *arbitrary* postcondition. Our analysis relies on this calculation of general substitutions, which we call *updates*. For example, if we would apply the assignment rule(s) to the formula $\{\texttt{i} := 3\}\langle\texttt{j = 2; i--;}\rangle\texttt{i} \doteq \texttt{j}$, we would get $\{\texttt{i} := 3\,||\,\texttt{j} := 2\}\langle\texttt{i--;}\rangle\texttt{i} \doteq \texttt{j}$, and then after another assignment rule application $\{\texttt{i} := 2\,||\,\texttt{j} := 2\}\langle\rangle\texttt{i} \doteq \texttt{j}$, which would simplify to $\{\texttt{i} := 2\,||\,\texttt{j} := 2\}\texttt{i} \doteq \texttt{j}$, and evaluate to *true*. This can be seen as symbolic execution, with the updates tracking all state changes. The state changes are kept within the curly brackets as a set of pairs of program locations and side-effect-free expressions, called *updates*. The updates can be seen as delayed substitutions (but there is much more to updates [25]). The calculus takes care of complications such as aliasing, exceptions, side-effects in branching conditions etc; and transfers those to side-effect free expressions in the updates, all without destroying the syntactic structure of the postcondition. This is a difference to other systems and something that we take advantage of in our analysis.

The system that we are using is called KeY [3] and the logic JAVA CARD Dynamic Logic [4, 6]. KeY is a system for integrated design, development and formal verification of JAVA CARD programs. In particular it has an interactive theorem prover for JAVA CARD DL with the above properties.

```
while (0 < i) {
  i--;
  s = s + j;
}
```

Figure 1: A program which calculates the product of two numbers. The same example appears in [26].

$\Longrightarrow$
$\forall il.\forall jl.(0 \leq il \rightarrow$
$\{\mathtt{i} := il \,||\, \mathtt{j} := jl \,||\, \mathtt{s} := 0\}$
$\langle$ `while (0 < i) {`
  `i--;`
  `s = s + j;`
 `}` $\rangle\ \mathtt{s} \doteq il * jl)$

Figure 2: Initial JAVA CARD DL proof goal.

3 Example

In this section we illustrate some of the problems encountered when using induction for proving the total correctness of a loop and at the same time we show our method at work. We wish to prove that the program in Figure 1 terminates in a state where `s` contains the product of the values that `i` and `j` had before the loop. That of course can only be proven under some assumptions about the initial values of `i`, `j` and `s`. We specify these requirements using a precondition and a postcondition, see Figure 2. In the precondition we express that the value of `i` in the starting state should be positive. The postcondition says that the value of `s` is the product of the starting values of `i` and `j`. In order to initialise `s` with $\mathtt{s} := 0$, we can either modify the program or we can modify the description of the starting state. Note that the *logical variables il* and *jl* are introduced since we cannot quantify over *program variables* (`i` and `j`).

In the general inductive proving process [27] there are at least three crucial points of interaction: 1) supplying the induction variable, 2) the choice of induction rule, and 3) supplying the induction formula. The last point is the focus of this paper, but we give a brief account of the entire process. First, to simplify the initial proof obligation (Figure 2) we use the rule allRight (see Appendix) to replace the universally quantified variables with Skolem constants il_c and jl_c and the rule impRight to decompose the implication. Applications of these two rules, and many others, such as commutativity and associativity, are performed automatically by the KeY prover. We will make them tacitly from now on.

The terminating condition for the loop tells us that the only suitable option for the induction variable is `i`. We expect the loop to terminate in the induction base case, that is, when the induction variable is zero. Again, we need a logical variable to quantify over, so we introduce *il* to be the induction variable. We make sure that `i` and *il* are connected in the initial state description of the induction formula $\phi(il)$. The update to the induction variable is given by `i--;` so the standard natInduct rule is suitable. The increment of the induction variable in the step case of natInduct $\forall n. 0 \leq n \wedge \phi(n) \rightarrow \phi(n+1)$ will then be cancelled by the decrement of the induction variable when symbolically executing the loop, thus removing one obstacle to closing the proof.

Trivial Induction Formula—Failed Proof Attempt After having chosen the induction variable *il* and natInduct as the induction rule it remains to supply the induction formula $\phi(il)$. The simplest possible approach would be to supply the simplified initial proof obligation modified only by replacing il_c with the induction variable *il*, the same formula as in Figure 2. The step case of a proof attempt using natInduct would (after skolemisation) be $\phi(il_c) \rightarrow \phi(il_c + 1)$. After having unrolled one iteration of the loop body in the succedent, the side effects would be collected in the state description. Since we cannot make the states in the antecedent $\{\mathtt{i} := il_c \,||\, \mathtt{j} := jl_c \,||\, \mathtt{s} := 0\}$ and in the succedent $\{\mathtt{i} := il_c \,||\, \mathtt{j} := jl_c \,||\, \mathtt{s} := jl_c\}$ syntactically equal, the proof attempt is stuck. Furthermore, the postcondition in the antecedent $\mathtt{s} \doteq il_c * jl_c$ and in the succedent $\mathtt{s} \doteq (il_c + 1) * jl_c$ cannot be made syntactically equal. At this point we are forced to start the induction proving process over with a better induction formula.

```
φ(il) ↔
∀sl.(0 ≤ il →
{i := il || j := jl_c || s := sl}
⟨ while (0 < i) {
    i--;
    s = s + j;
  } ⟩ POST(s, il, jl_c, sl))
```

Figure 3: Induction formula guess. Note that we have added a *meta-variable POST* to denote the unknown postcondition formula. We have also added a universally quantified variable *sl* so that we can later make the state descriptions syntactically equal.

```
∀il.∀sl.(0 ≤ il →
{i := il || j := jl_c || s := sl}
⟨ while (0 < i) {
    i--;
    s = s + j;
  } ⟩ POST(s, il, jl_c, sl)) ,
0 ≤ il_c
⟹
{i := il_c || j := jl_c || s := 0}
⟨ while (0 < i) {
    i--;
    s = s + j;
  } ⟩ s ≐ il_c * jl_c
```

Figure 4: Use case proof goal.

Generalised Induction Formula Guess Our idea is to construct a suitable induction formula without a number of tedious proof attempts but instead making use of the structure of the proof method. We first construct a *guess* for the induction formula $\phi(il)$, see Figure 3. The idea now is to perform the induction process as usual and learn the constraints on the correct postcondition. We first generate constraints for the use case, since the original proof obligation is a hard constraint.

Constraints from the Use Case After instantiating the use case with our induction formula guess we have the formula in Figure 4. To make the updates of the antecedent and succedent syntactically equivalent, we supply the substitution $[il/il_c, sl/0]$. To close this proof branch it only remains to show that the postconditions are syntactically equivalent, or that the guessed postcondition implies the postcondition of the starting proof obligation (under the assumption of the starting precondition). Since the postcondition is an unknown we generate the first constraint:

$$POST(\mathtt{s}, il_c, jl_c, 0) \land 0 \leq il_c \rightarrow \mathtt{s} \doteq il_c * jl_c \tag{E1}$$

Constraints from the Step Case We instantiate the step case with our induction formula guess and get the formula in Figure 5. After symbolic execution of the succedent, the proof branches at the while condition. In one branch the loop terminates, the program is of no more help, $il_c + 1$ must be zero and we get the following constraint:

$$POST(sl_c, 0, jl_c, sl_c) \tag{E2}$$

In the other branch (Figure 6), the loop is entered and the side effect of the loop body is visible in the update. Since an appropriate combination of induction rule and induction variable has been chosen, the effect on the induction variable is cancelled by the induction step. With the substitution $[sl/sl_c + jl_c]$ we achieve syntactic equivalence in the updates and we get rid of the quantifier. After impLeft and simplification we can prove the precondition branch $0 < il_c + 1 \rightarrow 0 \leq il_c$ and the remaining constraint for the postcondition is

$$0 \leq il_c \land POST(\mathtt{s}, il_c, jl_c, sl_c + jl_c) \rightarrow POST(\mathtt{s}, il_c + 1, jl_c, sl_c) \tag{E3}$$

$$
\begin{array}{l}
\forall sl.(0 \leq il_c \rightarrow \\
\{\mathtt{i} := il_c \,||\, \mathtt{j} := jl_c \,||\, \mathtt{s} := sl\} \\
\langle\, \mathtt{while\ (0\ <\ i)\ \{} \\
\quad \mathtt{i\text{--};} \\
\quad \mathtt{s\ =\ s\ +\ j;} \\
\ \mathtt{\}}\ \rangle\, POST(\mathtt{s}, il_c, jl_c, sl))\ , \\
0 \leq il_c + 1 \\
\Longrightarrow \\
\{\mathtt{i} := il_c + 1 \,||\, \mathtt{j} := jl_c \,|| \\
\ \ \mathtt{s} := sl_c\} \\
\langle\, \mathtt{while\ (0\ <\ i)\ \{} \\
\quad \mathtt{i\text{--};} \\
\quad \mathtt{s\ =\ s\ +\ j;} \\
\ \mathtt{\}}\ \rangle\, POST(\mathtt{s}, il_c + 1, jl_c, sl_c)
\end{array}
$$

Figure 5: Step case proof goal.

$$
\begin{array}{l}
\forall sl.(0 \leq il_c \rightarrow \\
\{\mathtt{i} := il_c \,||\, \mathtt{j} := jl_c \,||\, \mathtt{s} := sl\} \\
\langle\, \mathtt{while\ (0\ <\ i)\ \{} \\
\quad \mathtt{i\text{--};} \\
\quad \mathtt{s\ =\ s\ +\ j;} \\
\ \mathtt{\}}\ \rangle\, POST(\mathtt{s}, il_c, jl_c, sl))\ , \\
0 < il_c + 1 \\
\Longrightarrow \\
\{\mathtt{i} := il_c + 1 - 1 \,||\, \mathtt{j} := jl_c \,|| \\
\ \ \mathtt{s} := sl_c + jl_c\} \\
\langle\, \mathtt{while\ (0\ <\ i)\ \{} \\
\quad \mathtt{i\text{--};} \\
\quad \mathtt{s\ =\ s\ +\ j;} \\
\ \mathtt{\}}\ \rangle\, POST(\mathtt{s}, il_c + 1, jl_c, sl_c)
\end{array}
$$

Figure 6: Step case, loop entered.

$$
\begin{array}{l}
0 \leq 0 \\
\Longrightarrow \\
\{\mathtt{i} := 0 \,||\, \mathtt{j} := jl_c \,||\, \mathtt{s} := sl_c\} \\
\langle\, \mathtt{while\ (0\ <\ i)\ \{} \\
\quad \mathtt{i\text{--};} \\
\quad \mathtt{s\ =\ s\ +\ j;} \\
\ \mathtt{\}}\ \rangle\, POST(\mathtt{s}, 0, jl_c, sl_c)
\end{array}
$$

Figure 7: The base case proof obligation corresponds to termination of the loop.

Constraints from the Base Case Finally, let us consider the base case branch, see Figure 7. Since the loop condition is false when the induction variable is 0, we get the following constraint:

$$POST(sl_c, 0, jl_c, sl_c) \tag{E4}$$

3.1 Solution to the Constraints

We have generated one constraint from the use case (E1), one constraint from the base case (E4) (which is the same as the constraint from the terminating branch of the step case (E2)) and one constraint from the loop-entering branch of the step case (E3). We can solve these constraints (by hand) to the following which will yield a successful proof:

$$POST(\mathtt{s}, il, jl, sl) \leftrightarrow \mathtt{s} \doteq il * jl + sl$$

4 Deriving Constraints from a Symbolic Proof Attempt

We have shown by example how to construct an induction formula guess and how to use a structured induction proving process to generate the constraints that must be satisfied for a successful proof. We will now describe the method in more general terms, including a symbolic proof attempt.

1. The method has the original proof obligation as the starting point. First, basic simplification is performed. Universally quantified variables are replaced by arbitrary constants, so called Skolem constants, using the rule allRight. Implications are simplified using the rule impRight.

2. The next step is to supply the induction variable and the induction rule. The induction variable is found by observing the terminating condition of the loop. The induction rule is constructed to suit the particular program at hand by analysing the state change to the induction variable after symbolic execution of one loop iteration. For details on customised induction rules, see [13, 22]. In this paper we assume that the induction variable and a suitable induction rule are given, and instead focus on the remaining point: generalisation of the induction formula.

$$\begin{array}{l}
\Longrightarrow \\
\forall \overrightarrow{lv}.(pre_{uc}(\overrightarrow{lv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv})\} \\
\langle \, \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \, \rangle \, post_{uc}(\overrightarrow{\mathtt{pv}}, \overrightarrow{lv}))
\end{array}$$

Figure 8: General proof goal.

$$\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_{\phi}(il, \overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_{\phi}}(il, \overrightarrow{gv})\} \\
\langle \, \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \, \rangle \, POST_{\phi}(\overrightarrow{\mathtt{pv}}, il, \overrightarrow{gv}))
\end{array}$$

Figure 9: General induction formula guess.

3. As a first step in generalising the induction formula, we make a naive proof attempt where we supply the original proof obligation as the induction formula. As prescribed by the induction proving process in [27], in the step case of the inductive proof attempt we use the rule loopUnwind to symbolically execute one loop iteration. Then we use allLeft to instantiate any universally quantified variables in such a way that the states of the antecedent and succedent become syntactically equal for as many program variables as possible. From the remaining proof goal we learn how to construct a better induction formula.

4. Now we create a guess for the induction formula based on the information from the failed naive proof attempt. For every program variable that has a syntactic mismatch between the antecedent and succedent, we introduce a universally quantified variable. Furthermore, we introduce a meta-variable that is a placeholder for the correct but unknown postcondition formula. This depends on the previously used variables and the new universally quantified variables.

5. Then we start a new proof attempt, this time with the induction formula guess. We proceed with our structured induction proving process. In every open proof branch, there is a simplified proof goal that cannot be closed unless the meta-variables are instantiated. Thus every branch contributes one constraint on the meta-variables. If the constraints can be solved, we have found a correct generalisation of the induction formula and the proof is done.

4.1 Initial Proof Obligation

The original proof obligation ϕ_{pog} is of a general form that can be seen in Figure 8, where $\overrightarrow{lv}$ is a set of logical variables and $\overrightarrow{\mathtt{pv}}$ a set of program variables. There is a precondition pre_{uc}, a postcondition $post_{uc}$, and updates $\overrightarrow{\mu_{uc}}(\overrightarrow{lv})$ that describe the state of the program variables in the use case, in terms of the values of the logical variables.

After running basic simplifying heuristics and applying allRight, the logical variables in the succedent are replaced with skolem constants $\overrightarrow{lv_c}$. At this point a non-trivial proof attempt will be stuck and require the application of induction. Provided that we have found the right induction rule and induction variable, the most challenging task is to find the right induction formula.

4.2 Induction Formula Guess

Assume that—by an initial analysis of a trivial proof attempt—we know which variables that need to be generalised, we know the correct induction variable, how the induction variable is related to the program variables by a suitable update, and which induction rule to use. This part of our guess is given by [27, 22]. Then we make a general guess for the induction formula, $\phi(il)$, which is stated in Figure 9. The guess follows the same pattern as the proof obligation itself (Figure 8). The modifications consist of the addition of an induction variable, possibly additional generalised logical variables $\overrightarrow{gv}$, and updates

$$
\begin{array}{l}
\forall il.\forall \overrightarrow{gv}.(PRE_\phi(il,\overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}(il,\overrightarrow{gv})\} \\
\langle \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \rangle\ POST_\phi(\overrightarrow{\texttt{pv}},il,\overrightarrow{gv})) , \\
pre_{uc}(\overrightarrow{lv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})\} \\
\langle \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \rangle\ post_{uc}(\overrightarrow{\texttt{pv}},\overrightarrow{lv_c})
\end{array}
$$

Figure 10: Use case proof goal.

$$
\begin{array}{l}
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}([\sigma_{uc}]il,[\sigma_{uc}]\overrightarrow{gv})\} \\
\langle \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \rangle\ POST_\phi(\overrightarrow{\texttt{pv}},[\sigma_{uc}]il,[\sigma_{uc}]\overrightarrow{gv}) , \\
pre_{uc}(\overrightarrow{lv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})\} \\
\langle \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\;\; \texttt{\}} \rangle\ post_{uc}(\overrightarrow{\texttt{pv}},\overrightarrow{lv_c})
\end{array}
$$

Figure 11: Instantiated and simplified goal, postcondition branch.

$\overrightarrow{\mu_\phi}$ to relate these new logical variables to the program variables. Note that if we introduce additional universally quantified variables, then the generalised variable set $\overrightarrow{gv}$ is different from the logical variables $\overrightarrow{lv}$ in the original proof obligation (Figure 8). For example, in the product example in Section 3 the starting set of variables $\overrightarrow{lv}$ was il, jl and the generalised set of variables $\overrightarrow{gv}$ was il, jl, sl. The most important difference between the original proof obligation (Figure 8) and the guess (Figure 9) is the presence of $POST_\phi$, the yet unknown postcondition. We have introduced/selected $\overrightarrow{gv}$, il etc., in such a way that we know that the proof can be closed later, provided that we find a suitable $POST_\phi$. To turn the entire postcondition into an unknown is often too large a guess to be solvable. In the examples in the next section we will show some ways to make the guess more local. Since a different precondition is often necessary, we introduce a meta-variable PRE_ϕ for it.

After making this guess for the induction formula we will proceed with the induction proving process. If we can prove that the generalised induction formula is valid, and that this formula is enough to prove the original proof obligation, then we can close the proof. Let us follow the induction proving process and generate the constraints for $POST_\phi$ and PRE_ϕ.

4.3 Deriving Constraints from the Use Case

In this branch the task is always to prove that the induction formula (Figure 9) implies the original proof obligation (Figure 8). It is here that we apply the generalised induction formula which is still unknown and to be proved in the other branches of the induction proof. The use case branch (Figure 10) is the most constraining branch since it contains the problem as supplied by the user.

To generate the use case constraints, we first need to find a substitution σ_{uc} such that each update in $\overrightarrow{\mu_\phi}([\sigma_{uc}]il,[\sigma_{uc}]\overrightarrow{gv})$ is syntactically equal to the corresponding update in $\overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})$. Usually it is easy to find such a σ_{uc}. Then we use this substitution to instantiate il and $\overrightarrow{gv}$. After the instantiation and some simplification, the process has given one branch for the precondition, and one for the postcondition (Figure 11). From the precondition branch, we can generate a constraint (C1). If a solution for the constraint cannot be found, the constraint can be used to inform the user that the precondition that he has supplied is too weak, which is a common error.

$$pre_{uc}(\overrightarrow{lv_c}) \rightarrow PRE_\phi([\sigma_{uc}]il,[\sigma_{uc}]\overrightarrow{gv}) \tag{C1}$$

Next we consider the postcondition branch of the use case (Figure 11). Since σ_{uc} has been chosen to make the updates syntactically equal and the programs are the same, only the postcondition remains to

$$\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_\phi(il_c,\overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c,\overrightarrow{gv})\} \\
\langle\, \mathtt{while}\ (c_{while})\ \{ \\
\quad \mathtt{loop\ body;} \\
\ \ \}\ \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}},il_c,\overrightarrow{gv}))\ , \\
PRE_\phi(il_c+1,\overrightarrow{gv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\} \\
\langle\, \mathtt{while}\ (c_{while})\ \{ \\
\quad \mathtt{loop\ body;} \\
\ \ \}\ \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}},il_c+1,\overrightarrow{gv_c})
\end{array}$$

Figure 12: Step case goal.

$$\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_\phi(il_c,\overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c,\overrightarrow{gv})\} \\
\langle\, \mathtt{while}\ (c_{while})\ \{ \\
\quad \mathtt{loop\ body;} \\
\ \ \}\ \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}},il_c,\overrightarrow{gv})), \\
PRE_\phi(il_c+1,\overrightarrow{gv_c}), \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_{sym}}(il_c+1,\overrightarrow{gv_c})\} \\
\langle\, \mathtt{while}\ (c_{while})\ \{ \\
\quad \mathtt{loop\ body;} \\
\ \ \}\ \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}},il_c+1,\overrightarrow{gv_c})
\end{array}$$

Figure 13: Step case, loop entered.

$$\begin{array}{l}
PRE_\phi(0,\overrightarrow{gv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(0,\overrightarrow{gv_c})\} \\
\langle\, \mathtt{while}\ (c_{while})\ \{ \\
\quad \mathtt{loop\ body;} \\
\ \ \}\ \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}},0,\overrightarrow{gv_c})
\end{array}$$

Figure 14: Base case proof obligation.

be shown. Thus we generate a constraint on the postcondition (C2).

$$POST_\phi(\overrightarrow{\mathtt{pv}},[\sigma_{uc}]il,[\sigma_{uc}]\overrightarrow{gv}) \wedge pre_{uc}(\overrightarrow{lv_c}) \rightarrow post_{uc}(\overrightarrow{\mathtt{pv}},\overrightarrow{lv_c}) \tag{C2}$$

4.4 Deriving Constraints from the Step Case

The step case (Figure 12) holds two instances of the induction formula. First we symbolically execute the succedent of the proof obligation, which makes the proof branch at the while condition c_{while}. Recall that a negative while condition corresponds to termination of the loop so after symbolic execution the program has disappeared and the remaining constraint for this branch is (C3).

$$\begin{array}{l}
PRE_\phi(il_c+1,\overrightarrow{gv_c})\ \wedge\neg\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \\
\rightarrow \{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\}\ POST_\phi(\overrightarrow{\mathtt{pv}},il_c+1,\overrightarrow{gv_c})
\end{array} \tag{C3}$$

The case where the while condition is positive, and in the succedent the loop body has been executed once with the help of loopUnwind (Figure 13), gives rise to the new update $\overrightarrow{\mu_{sym}}$. The induction variable and the induction rule have already been chosen so that for the program variable that depends on the induction variable il, $\overrightarrow{\mu_{sym}}(il_c+1)$ is syntactically equivalent to $\overrightarrow{\mu_\phi}(il_c)$. Also here in the step case we must find a substitution σ_{sc} so that $\overrightarrow{\mu_{sym}}(il_c+1,\overrightarrow{gv_c})$ is syntactically equivalent to $\overrightarrow{\mu_\phi}(il_c,[\sigma_{sc}]\overrightarrow{gv})$ for the rest of the program variables. Then we use this substitution to instantiate $\overrightarrow{gv}$. After applying impLeft we get another branching, with one goal for the precondition and one goal for the postcondition. The first one gives another constraint on the precondition:

$$PRE_\phi(il_c+1,\overrightarrow{gv_c})\ \wedge\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \rightarrow PRE_\phi(il_c,[\sigma_{sc}]\overrightarrow{gv}) \tag{C4}$$

From the other branch we can generate a constraint on the postcondition:

$$\begin{array}{l}
PRE_\phi(il_c+1,\overrightarrow{gv_c}) \wedge\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c+1,\overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \wedge POST_\phi(\overrightarrow{\mathtt{pv}},il_c,[\sigma_{sc}]\overrightarrow{gv}) \\
\rightarrow POST_\phi(\overrightarrow{\mathtt{pv}},il_c+1,\overrightarrow{gv_c})
\end{array} \tag{C5}$$

4.5 Deriving Constraints from the Base Case

At this point, we prepare for one more constraint that corresponds to termination of the loop, see Figure 14. We have already decided upon an induction variable and updates such that the while condition c_{while}

$\forall nl.(0 \leq nl \rightarrow$
$\{\mathtt{n} := nl\}$
```
⟨ i=0;
  r=0;
  while (i < n) {
   i++;
   r = r+(i*i*i);
  } ⟩ 4r ≐ nl²(nl+1)²)
```

Figure 15: Cubic sum: Original proof obligation.

$\forall rl.(0 \leq nl_c \rightarrow$
$\{\mathtt{i} := nl_c - kl \,||\, \mathtt{n} := nl_c \,||\, \mathtt{r} := rl\}$
```
⟨ while (i < n) {
    i++;
    r = r+(i*i*i);
  } ⟩ POST(r,kl,rl,nl_c))
```

Figure 16: Cubic sum: Induction formula guess.

is false and the loop terminates when $il = 0$. So symbolic execution of the program in Figure 14 will generate an empty program and this gives us the last constraint:

$$PRE_\phi(0,\overrightarrow{gv_c}) \wedge \neg\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(0,\overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \rightarrow \{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(0,\overrightarrow{gv_c})\}\ POST_\phi(\overrightarrow{\mathtt{pv}},0,\overrightarrow{gv_c}) \qquad \text{(C6)}$$

Now it only remains to solve the constraints. In this paper we do not present a method for solving the constraints but a discussion of ideas for constraint solving is included in Section 7.

5 Example: Cubic Sum

Let us try our method at a program that computes the so called cubic sum.

$$\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4 \ .$$

The problem would be specified in JAVA CARD DL as in Figure 15. We introduce an induction variable kl, where $kl = \mathtt{n} - \mathtt{i}$ (see the terminating condition of the loop), and we choose to apply the normal natInduct rule (see the update to the induction variable inside the loop). Then we construct the induction formula guess in Figure 16.

We set the guess for the precondition PRE to $true$. We then generate the following constraints, from the use case, the step case and the base case respectively:

$$POST(\mathtt{r},nl_c,0,nl_c) \ \rightarrow\ 4\mathtt{r} \doteq nl_c^2(nl_c+1)^2 \qquad \text{(CS1)}$$

$$POST(\mathtt{r},kl_c,rl_c+(nl_c-kl_c)^3,nl_c) \ \rightarrow\ POST(\mathtt{r},kl_c+1,rl_c,nl_c) \qquad \text{(CS2)}$$

$$POST(rl_c,0,rl_c,nl_c) \qquad \text{(CS3)}$$

Now we can get hold of the desired postcondition by solving the constraints. The constraints are satisfied by the following $POST(\mathtt{r},kl,rl,nl_c)$, which makes the induction formula guess (Figure 16) provable:

$$POST(\mathtt{r},kl,rl,nl_c) \leftrightarrow 4(\mathtt{r}-rl) \doteq nl_c^2(nl_c+1)^2 - (nl_c-kl)^2(nl_c-kl+1)^2 \ .$$

6 Example: Find the Minimal Element in an Array

Now let us consider a more advanced example, finding the index of the smallest element in an array, see Figure 17. The postcondition of this loop is more complicated than in the previous examples, as it involves quantification over an array interval. In the following constraint generation, we will make the "guess" more local than before by introducing several meta-variables in the postcondition. This approach can be seen as an attempt at classifying the postcondition based on its structure. In this case the postcondition is that a property holds for an interval.

$\Longrightarrow$
```
⟨ m = 0;
  x = 0;
  while (x < a.length) {
   if (a[x] < a[m]) {
     m = x;
   }
   x++;
  } ⟩ ∀kl.0 ≤ kl < a.length
          → a[m] ≤ a[kl] )
```

Figure 17: Findmin: Original proof obligation.

$\forall ml.(PRE(\mathtt{x},\mathtt{m},il,ml,\mathtt{a.length}) \rightarrow$
$\{\mathtt{x} := \mathtt{a.length} - il \,||\, \mathtt{m} := ml\}$
```
⟨ while (x < a.length) {
    if (a[x] < a[m]) {
      m = x;
    }
    x++;
  } ⟩ ∀kl.(F1(x,m,il,ml,a.length) ≤ kl
            < F2(x,m,il,ml,a.length)
            → P(x,m,kl,il,ml,a.length))
        ∧ Q(x,m,il,ml,a.length))
```

Figure 18: Findmin: Induction formula guess.

Induction Formula Guess We start by choosing an appropriate induction variable. The `while` condition suggests that the loop terminates when $\mathtt{a.length} - \mathtt{x}$ is zero, so we introduce the induction variable $il = \mathtt{a.length} - \mathtt{x}$ and specify the relation between il and `x` in the update of `x`. The induction variable will be decreased by one in every loop iteration (via `x++`) so natInduct will be a suitable induction rule.

In the induction formula guess (Figure 18), we could put one meta-variable *POST* as a guess for the entire postcondition as described in Section 4. However, in our quest to generate constraints that are easier to solve, and to exploit the input from the user as much as we can, we make a slightly different guess. The idea is to use the structure of the postcondition that comes from the proof obligation given by the user. We keep the general shape, in this case quantification over a range, and modify it by replacing the ”leaves” with meta-variables. In addition to the part that comes from the original proof obligation, we add another unknown (here Q), to allow further strengthening of the postcondition. With a guess like this we can treat a whole class of postconditions where the user wants a certain property to hold for all elements of an array. We also introduce a meta-variable *PRE* for the precondition. We then perform a proof attempt in order to generate the constraints. The proof attempt and the generated constraints have been omitted for brevity, but they can be found in [28].

Solution to the Constraints The below solution to the constraints was derived by hand. See Section 7 for a discussion on how this solution could possibly be found automatically.

$$F_1(\mathtt{x},\mathtt{m},il,ml,\mathtt{a.length}) = \mathtt{a.length} - il \tag{FM1}$$

$$F_2(\mathtt{x},\mathtt{m},il,ml,\mathtt{a.length}) = \mathtt{a.length} \tag{FM2}$$

$$P(\mathtt{x},\mathtt{m},kl,il,ml,\mathtt{a.length}) \leftrightarrow \mathtt{a[m]} \leq \mathtt{a}[kl] \tag{FM3}$$

$$Q(\mathtt{x},\mathtt{m},il,ml,\mathtt{a.length}) \leftrightarrow \mathtt{a[m]} \leq \mathtt{a}[ml] \tag{FM4}$$

$$PRE(\mathtt{x},\mathtt{m},il,ml,\mathtt{a.length}) \leftrightarrow true \tag{FM5}$$

7 Constraint Solving

The method described in this article generates an induction formula guess with unknown parts, and constraints on those parts. If a solution for the constraints can be found, the resulting formula can be automatically proven by induction. The author solves by hand constraints like the ones in the examples in Sections 5 and 6 roughly according to this iterative recipe:

1. Pick values for all of the meta-variables to satisfy the constraints from the use case. The use case is always considered first because it contains a lot of concrete information about the problem as given by the user. It also only has one instance of the induction formula, making it easier to solve by purely syntactic methods. We try to include the induction variable so that, intuitively, the postcondition holds from the current value of the induction variable until the loop terminates.

2. Next we consider the step case postcondition constraint(s), because the step case is the heart of an induction proof. If the antecedent does not completely imply the succedent with the current meta-variable values, the values are amended. For example, in Section 6, the range over which the succedent quantifies includes one element more than the range in the antecedent. Thus we may need to amend the solution to account for that element. If so, we have changed the postcondition and we start over, with the new meta-variable instantiation. If there is basic type information missing, try to add it to the precondition.

3. Consider the constraint from the base case. In the process, the base case is mostly a sanity check. If the solution under consideration does not satisfy this constraint, we often need to rethink how the induction variable is used and go back to the use case.

4. If the precondition instantiation is not $true$, check all the constraints on the precondition.

This process is largely guided by trying to achieve syntactic equivalence wherever possible. For example, we can first try to solve an implication by looking for syntactic equality. Other tricks include splitting ranges, and weakening by removing conjuncts and shrinking ranges. This constraint solving method is guided by experience and intuition, and does not lend itself to direct implementation. However, it may be used as a starting point in optimising the search order in a mechanical solver.

An automatic method for solving the generated constraints is beyond the scope of this article. Indeed, since the constraints involve second-order variables, solving them is undecidable in the general case. However, as with many undecidable problems, there are partial algorithms for solving second-order constraints. If a given set of constraints has a solution where all meta-variables are instantiated with values in some known limited class, we can often solve them automatically. One example of such a class is polynomials (for integer-valued meta-variables) and polynomial equations (for truth-valued meta-variables), of some bounded degree. Such guesses have been used to solve constraints in loop invariant generation [26, 17]. Another interesting class is linear terms and linear inequalities. Gulwani et al. [12] convert second-order constraints for program analysis to SAT problems, replacing second-order meta-variables with linear inequalities.

The choice of the solution class to try could be guided by the original proof obligation as given by the user. For example, in the product example in Section 3, the original postcondition was a polynomial equation of degree 2 and the constraints could be solved with a polynomial equation of the same degree. In the cubic sum example in Section 5, both the user-supplied postcondition and the generated solution was a polynomial equation of degree 4. Likewise, in the findmin example in Section 6, the original proof obligation contained a linear inequality, as did the solution. Of course, if no solution is found in one class, other classes could be tried. If no solutions can be found at all, we could go back and try to refine the induction formula guess. In general, we can treat the constraint solving as a search problem, where we make progressively more refined guesses when simpler ones fail.

8 Limitations and Future Work

This article outlines an idea for a constraint-based approach to induction formula generalisation. There is plenty more to do before this is a practical and automatic method. Firstly, there is no implementation of the constraint generation method, though it should be rather straightforward to build a proof-of-concept implementation within the KeY system. Secondly, we have not tried to solve the generated constraints automatically, although the previous section presents some ideas for how that could be done.

The examples that we have shown outline classes of problems that can be handled by our method. It would be interesting to try to describe these classes formally, and to identify additional classes of problems. This would naturally also include trying a large number of additional examples.

The calculus that we have used, JAVA CARD DL, can handle more complex language features than we have used in the examples here. It would be interesting to try our method on examples that include array modification and exceptions. In the case of array modification, quantified updates [25] could probably be used to achieve syntactic equivalence in the updates of the array elements.

So far, we have only considered single (unnested) loops. Perhaps the method could be generalised to handle nested loops by generating (and solving) constraints for all the loops simultaneously. Furthermore, we have only dealt with induction over natural numbers, though it should be possible to extend the method to structural induction.

9 Related Work

The automation of inductive proofs has been a research topic for more than 30 years. Several systems for reasoning about programs in functional programming languages have induction heuristics, most notably ACL2 [8, 7], Verifun [29, 1], RRL [18], and Rippling [9].

ACL2 has a powerful mechanism for generating induction schemes from recursive definitions. However, the mechanism for induction formula generalisation is simpler. The ACL2 generalisation heuristic detects common subterms in the hypothesis and conclusion of the induction step, after unrolling the function in the conclusion and simplifying it, and then constructs a generalised induction formula by replacing the subterms with new universally quantified variables. For example, ACL2 does not automatically prove the correctness of the cubic sum example in Section 5 (translated to LISP) as it is originally stated. Thanks to the mature and widely available implementation of ACL2 it was easy to translate the cubic sum example to ACL2 and attempt to prove it correct. We translate the loop to a tail recursive function with an accumulator, implement the closed-form solution as another function, and try to prove that the two give the same result for any natural number:

```
(defun csum1 (i s) (if (not (zp i)) (csum1 (- i 1) (+ s (* i i i))) s))
(defun csum2 (n) (* n n (+ n 1) (+ n 1) 1/4))
(defthm csum-correct (implies (natp n) (equal (csum1 n 0) (csum2 n))))
```

However, ACL2 is not able to prove this. But if we instead use the generalized formula produced by our method, the proof succeeds (see [16] for related work on generalisation with accumulators):

```
(defthm csum-correct-gen
  (implies (and (natp n) (natp s)) (equal (csum1 n s) (+ (csum2 n) s))))
```

Rippling is an induction heuristic for guiding rewriting (proofs) by annotating both the formulae (proof goals) and the rewrite rules so that only those rewrites are allowed that actually make the goal syntactically closer to the assumption and in the end hopefully closable. The annotations can be used to derive suitable induction schemes, to create intermediate lemmas and to create induction formula generalisations. Our approach has several similarities to the rippling approach, most importantly that it is syntactically driven and that it analyses failed proof attempts. We were inspired by their use of meta-variables in the generalisation process.

A heuristic for lemma discovery in the rewrite-based induction theorem prover RRL [18], also analyses failed proof attempts. Their introduction of "non-induction variables" seems to correspond to our introduction of new universally quantified variables. A difference is that they need to search for the right instantiations for those variables, whereas that part is rather straightforward and syntactically driven in our approach, thanks to the update mechanism of JAVA CARD DL.

VeriFun [31] is a semi-automatic verification tool for functional programs written in the $\mathscr{L}$ [30] programming language. Its induction formula generalisation heuristics have been developed in a long line of research [29, 15, 1]. The heuristics are inverse applications of sound inferences. Like in ACL2, repeated subterms are replaced by new variables (*inverse substitution*). Other heuristics include *inverse substitution* to generalise apart multiple occurrences of a variable, *inverse weakening* to remove conditions, and *inverse functionality* to remove function applications. Aderhold [1] recently extended VeriFun to use a fast disprover [2] for detecting the over-generalisations that each of the heuristics may produce.

Several techniques for the automatic generation of loop invariants have been developed using abstract interpretation. In [20] a widening operator is built into an SMT solver, so that the widening operator can be applied by the solver whenever it needs to approximate loop invariants. In their previous approach [21], a separate static analyser and a theorem prover were used together so that loop invariants could be refined using counterexamples from the theorem prover. In [23] program statements are translated into transformations on polynomial ideals and a sound and complete method for automatically finding polynomial loop invariants by computing Gröbner bases is then given. The approach has the restriction that it must be possible to abstract conditionals in the program to polynomial equalities and disequalities, but it can solve many non-trivial examples. The algebraic foundations of inferring conjunctions of polynomial equations as invariants by computing Gröbner bases is further described in [24]. [19] also presents an algebraic method for finding polynomial equations as loop invariants. Another approach that generates constraints based on guessing that the invariant is a polynomial with real coefficients is presented in [26]. It also uses Gröbner bases to solve the constraints.

A very recent approach [12] shows a constraint-based way to perform program analysis. From the control-flow graph, constraints on loop invariants are generated, where the loop invariant itself is a second-order meta-variable. The constraints are transformed to a form that can be handled by an off-the-shelf SAT solver, by replacing the second-order variables with boolean combinations of linear inequalities over program variables. The approach in [17] also generates constraints where the loop invariant is a second-order meta-variable. The loop invariant is then hypothesised to be a polynomial equation and the constraints are solved by algebraic techniques. It would be interesting to use the constraint solving techniques from both of these approaches to solve the constraints generated in our approach.

10 Summary and Conclusions

We have presented an idea for mechanical generalisation of induction formulae. It introduces meta-variables in an existing proof obligation that has been provided by the user. It then uses a well-structured and syntactically driven process, that could be called a symbolic proof attempt, to generate constraints on the meta-variables. The method is based on a weakest-precondition calculus with forward symbolic execution that does not destroy the syntactic structure of the postcondition. Since the calculus that we use can transform programming language features with side-effects to side-effect-free formulae, we take advantage of this to handle a complex imperative language, JAVA CARD.

We have not presented a method for solving the constraints that our method generates. However, we have discussed existing work on solving classes of constraints similar to those that we generate. Furthermore, when solving our constraints by hand, we have made some observations that could be useful for an automatic constraint solver.

Most of the existing work on automatic verification of data correctness for loops is done in the context of a small (functional or imperative) language. In contrast, our approach is based on a logic which can directly handle a large imperative and object-oriented language. While it is true that any program in a large programming language can be translated to a program in a small language, this can make it much harder to give sensible feedback to the verification tool user. An important piece of future work would be to implement the method and evaluate it on a number of non-trivial examples. Hopefully, we can make use of the mechanism in KeY to replay proofs (attempts) [5] in the implementation.

Acknowledgements The author would like to thank Björn Bringert, Koen Claessen, John Hughes, Reiner Hähnle, Deepak Kapur, Philipp Rümmer, Mary Sheeran, and the anonymous reviewers, for their help and comments on earlier drafts of this paper.

References

[1] Markus Aderhold. Improvements in formula generalization. In Frank Pfenning, editor, *Proceedings of the 21st Conference on Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 231–246, Bremen, Germany, 2007. Springer-Verlag.

[2] Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser. A fast disprover for verifun. In Wolfgang Ahrendt, Peter Baumgartner, and Hans de Nivelle, editors, *Proceedings of the 3rd Workshop on Disproving, IJCAR 2006*, pages 59–69, Seattle (WA), USA, 2006.

[3] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.

[4] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[5] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press, 2004.

[6] Bernhard Beckert and Bettina Sasse. Handling JAVA's abrupt termination in a sequent calculus for Dynamic Logic. In *IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14, 2001.

[7] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[8] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.

[9] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[10] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969. Edited by M. E. Szabo.

[11] Martin Giese. First-order logic. In Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, pages 21–65. Springer, 2006.

[12] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. *SIGPLAN Not.*, 43(6):281–292, 2008.

[13] Reiner Hähnle and Angela Wallenburg. Using a software testing technique to improve theorem proving. In Alexandre Petrenko and Andreas Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003), Montréal, Canada*, volume 2931 of *LNCS*, pages 30–41. Springer-Verlag, 2004.

[14] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[15] B. Hummel. *Generation of Induction Axioms and Generalizations*. PhD thesis, Universität Karlsruhe, 1990.

[16] Andrew Ireland and Alan Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9(2):225–245, 1999.

[17] Deepak Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.

[18] Deepak Kapur and Mahadevan Subramaniam. Lemma discovery in automated induction. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 538–552, London, UK, 1996. Springer-Verlag.

[19] Laura Kovács. Reasoning algebraically about p-solvable loops. In *TACAS*, pages 249–264. Springer, 2008.

[20] K. R. M. Leino and F. Logozzo. Using widenings to infer loop invariants inside an smt solver, or: A theorem prover as abstract domain. In *International Workshop on Invariant Generation (WING'07)*, pages 70–84, Hagenberg, Austria, June 2007. RISC.

[21] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[22] Ola Olsson and Angela Wallenburg. Customised induction rules for proving correctness of imperative programs. In Bernhard Aichernig and Bernhard Beckert, editors, *Software Engineering and Formal Methods. 3rd IEEE International Conference, SEFM 2005, Koblenz, Germany, September 7–9, 2005, Proceedings*, pages 180–189. IEEE Computer Society Press, 2005.

[23] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *International Symposium on Static Analysis (SAS 2004)*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2004.

[24] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04)*, pages 266–273. ACM Press, 2004.

[25] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

[26] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 318–329, New York, NY, USA, 2004. ACM.

[27] Angela Wallenburg. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*, chapter Proving by Induction, pages 453–479. Springer, 2007.

[28] Angela Wallenburg. Generalisation of induction formulae based on proving by symbolic execution. Technical report, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, 2009.

[29] Christoph Walther. Mathematical induction. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming. Deduction Methodologies*, volume 2, chapter 3, pages 127–227. Oxford University Press, 1994.

[30] Christoph Walther, Markus Aderhold, and Andreas Schlosser. The L 1.0 primer. Technical Report VFR 06/01, FG Programmiermethodik, Technische Universität Darmstadt, 2006.

[31] Christoph Walther and Stephan Schweitzer. About VeriFun. In *Automated Deduction — CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer-Verlag, 2003.

Appendix: Some Relevant JAVA CARD DL Rules

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow [x/c](\phi),\ \Delta}{\Gamma \Longrightarrow \forall x.\phi,\ \Delta} \qquad \mathsf{allLeft}\ \frac{\Gamma,\ \forall x.\phi, [x/t](\phi) \Longrightarrow \Delta}{\Gamma,\ \forall x.\phi \Longrightarrow \Delta}$$

with $c : \to A$ a new constant, if $x{:}A$ with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{impLeft}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma,\ \psi \Longrightarrow \Delta}{\Gamma,\ \phi \to \psi \Longrightarrow \Delta} \qquad \mathsf{loopUnwind}\ \frac{\Longrightarrow \langle \pi\ \texttt{if (}e\texttt{) \{ }p\ \texttt{while (}e\texttt{) }p\texttt{ \}}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{while (}e\texttt{) }p\ \omega\rangle\phi}$$