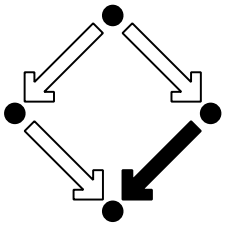
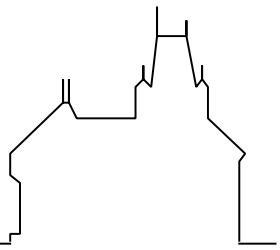


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



WING 2007

Workshop on Invariant Generation

—Working Proceedings—

Martin GIESE, Tudor JEBELEAN (eds.)

Hagenberg, Austria
June 25–26, 2007

RISC-Linz Report Series No. 07-07

Editors: RISC-Linz Faculty
B. Buchberger, R. Hemmecke, T. Jebelean, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by:
Austrian Science Fund (FWF project SFB F013)
European Commission Framework 6 Prog. for Integrated Infrastructures Initiatives (project SCIEnce)

Preface

Many groups around the world conduct research on formal methods for software development, and in most of these groups, some of the effort goes into the problem of reasoning about loops. There is of course a well-known classic way of dealing with loops, namely by having the software developer provide an invariant, which can be shown to be preserved by the loop. However, experience shows that writing these invariants can be difficult in some cases, trivial but tedious in others, unnecessary in still others. The hope of obviating the need for hand-written invariants, or at least simplifying their production, is what keeps the research on loops alive.

Research results tend to be presented at general conferences on formal methods, logic, or mathematics, usually depending on their scope and the methods employed, but this means that researchers using different approaches often do not know about each others' work. To improve this situation, we decided to organize a workshop bringing together people interested particularly in reasoning about loops.

Initially, we feared that calling for papers about loop reasoning in general might attract submissions about any work having to do with program verification, on the grounds that reasoning about loops is the theoretically most challenging aspect of deductive program verification. Should the call for papers be restricted to invariant generation techniques, or even to algebraic techniques? It has turned out that keeping the call for papers open was the right thing to do, since it has attracted an interesting variety of papers, almost all of them quite specific to loop reasoning.

The papers presented at this workshop range from early ideas to descriptions of mechanisms employed in established systems. Methods include Gröbner bases, combinatorics, separation logic, rippling, weakest preconditions, and many others. Some of the approaches are targeted toward full functional verification, i.e. inferring the strongest possible invariants, while others try to find invariants that are just strong enough to ensure that array indices are within bounds, pointers are not null, etc. We would like to thank the program committee members and reviewers for their effort. Their names are listed on the following page.

As organizers, we are extremely pleased that the WING workshop presents such a varied view of this very diverse field. It is our hope that during this workshop, every participant will encounter at least one new approach he or she was not previously aware of.

The workshop program is rounded off by invited talks by John Harrison (Intel Corporation, USA, joint with Calculemus), Reiner Hähnle (Chalmers University of Technology, Sweden), Deepak Kapur (University of New Mexico, USA), and Andrei Voronkov (Manchester University, UK), whom we thank for their participation and effort.

Finally, we are very grateful to the local organizers of the RISC Summer 2007 conference series, and in particular to Laura Kovács for taking care so competently of the many logistic and organizational issues.

June 2007

Martin Giese
Tudor Jebelean

Program Chairs
WING 2007

Organization

WING 2007 is organized by the Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Executive Committee

Workshop Chair:	Bruno Buchberger (RISC, Austria)
Program Chairs:	Martin Giese (RISC, Austria) Tudor Jebelean (RISC, Austria)
Local Chair:	Laura Kovács (RISC, Austria)

Program Committee

Nikolaj S. Bjørner (Stanford University and Microsoft Research, USA)
Ewen Denney (NASA Ames Research Center, USA)
Jens Knoop (Technical University of Vienna, Austria)
Enric Rodrguez Carbonell (Technical University of Catalonia, Barcelona, Spain)
Wolfgang Schreiner (RISC, Austria)
Helmut Seidl (Technical University of Munich, Germany)
Andrei Voronkov (Manchester University, UK)

Additional reviewer: Michael Petter (TU of Munich, Germany)

Sponsoring Institutions

- Special Research Program SFB F013—Numerical and Symbolic Scientific Computing—of the Austrian Science Fund (FWF)
- European Commission Framework 6 Programme for Integrated Infrastructures Initiatives, project SCIEnce—Symbolic Computation Infrastructure for Europe (Contract No. 026133)

Table of Contents

Integrating Linear Arithmetic into the Superposition Calculus
(abstract of invited talk)

Konstantin Korovin and Andrei Voronkov

The University of Manchester
{korovin|voronkov}@cs.man.ac.uk

Abstract. We present a method of integrating linear rational arithmetic into superposition calculus for first-order logic. One of our main results is completeness of the resulting calculus under some finiteness assumptions.

Some approaches to invariant generation depend on proving theorems about various data types used in programming languages, including arrays and integers. The existing methods of theorem proving for such theories use combinations of decision procedures for satisfiability of quantifier-free formulas.

The restriction on the formulas to be ground turns out to be restrictive when some data types are described by a collection of axioms with quantifiers. When one needs to use both decision procedures for quantifier-free formulas and arbitrary first-order axioms, the existing theorem provers try to guess instantiations of universally quantified variables in the axioms by ground terms.

This talk introduces a new approach to combining decision procedures with arbitrary first-order axiomatisations based on an extension of the superposition calculus by rules for built-in theories. It is illustrated on linear (rational) arithmetic where the rules for rational linear arithmetic are based on Gaussian Elimination for reasoning with equality and Fourier-Motzkin Elimination for reasoning with inequalities. The calculus is non-ground, so variable instantiation is decided by unification instead of guessing ground instances.

The talk will present the following ideas and results:

1. A formalisation of combination of first-order axioms and built-in domains.
2. Results about such combinations.
3. LASCA — a Linear Arithmetic Superposition CAlculus.
4. Incompleteness and completeness results for the calculus.

A more detailed version of this paper is to appear as [1]. We acknowledge the support of an EPSRC grant for both authors and the SCIEnce TAP (European Commission Framework 6 Programme for Integrated Infrastructures Initiatives) for the second author.

References

1. K. Korovin and A. Voronkov. Integrating linear arithmetic into the superposition calculus. In *Computer Science Logic (CSL 2007)*, 2007. To appear.

A Cooperative Approach to Loop Invariant Discovery for Pointer Programs

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK

Abstract. We propose a cooperative approach to reasoning about the correctness of pointer programs within the context of separation logic. A key hurdle to achieving scalable program proof is the burden imposed by the need for loop invariants. Pointer data structures can be viewed in terms of their *shape* and *content*. Our proposal exploits this distinction by combining two techniques for automating the discovery of loop invariants. Firstly, a technique developed at CMU for discovering shape invariants based upon symbolic evaluation. Secondly, middle-out proof planning, a theorem proving technique applicable to reasoning about loops and which supports loop invariant discovery. We believe that the complementary nature of these techniques will deliver significant advantages in terms of scalable pointer program proof.

1 Introduction

Pointers are a powerful and widely used programming mechanism. Developing and maintaining correct pointer programs, however, is notoriously hard. It is therefore highly desirable to be able to routinely reason about the correctness of pointer programs. The stumbling block to achieving such a goal has been the lack of scalable methods of reasoning. At the level of proof search, the need for invariants, and in particular loop invariants, remains a major obstacle to proof automation, and thus scalability. Here we propose a cooperative approach to reasoning. Using an invariant discovery technique developed at CMU [18], we describe how *symbolic evaluation* and *middle-out proof planning* [7] could be combined in order to support the automation of partial correctness proofs. In terms of logics, our proposal is grounded in *separation logic* [21, 22], which addresses aspects of the scalability problem through its support for local reasoning.

The paper is structured as follows. Background on separation logic and a motivating example are introduced in §2. In §3, an overview of the CMU invariant discovery technique is provided, together with background material on middle-out proof planning. The details of our proposed cooperative approach are described in §4. The potential benefits of the approach are discussed in §5, while related and future work are described in §6 and §7 respectively.

2 Separation Logic and the Burden of Loop Invariants

Separation logic was developed as an extension to Hoare logic with the aim of simplifying pointer program proofs. A key feature of the logic is that it focuses the reasoning effort on only those parts of the heap that are relevant to a program. Here we give a brief introduction to separation logic, for a full account see [22]. Separation logic extends predicate calculus with new forms of assertions for describing the heap:

- empty heap: the assertion *emp* holds for a heap that contains no cells.
- singleton heap: the assertion $X \mapsto E$ holds for a heap that contains a single cell, *i.e.* X denotes the address of a cell with contents E.
- separating conjunction: the assertion $P * Q$ holds for a heap if the heap can be divided into two disjoint heaps H_1 and H_2, where P holds for H_1 and Q holds for H_2 simultaneously.
- separating implication: the assertion $P \mathrel{-\!\!*} Q$ holds for a heap H_1, if whenever H_1 is extended with a disjoint heap H_2, for which P holds, then Q holds for the extended heap.

Typically one wants to assert that a pointer variable, say X, points to E within a larger group of heap cells. This can be represented by $(X \hookrightarrow E)$, which is simply an abbreviation for

$$(true * (X \mapsto E)) \tag{1}$$

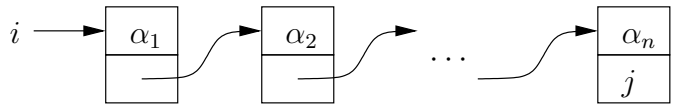
If (1) holds for the current heap, then it means that it can be divided into two parts: A singleton heap, for which $(X \mapsto E)$ holds; and the rest of the current heap, for which *true* holds. Note that *true* holds for any heap. In what follows we will focus on pointers that reference a pair of adjacent heap cells. So we will use $(X \mapsto E_1, E_2)$ as an abbreviation for $(X \mapsto E_1) * (X + 1 \mapsto E_2)$.

To motivate our proposal we outline the application of our approach to verifying the partial correctness of an in-place list reversal program. The example is taken from [22], where Reynolds uses a predicate called *list* to relate the notion of a *singly-linked* list to the abstract notion of sequences. An inductive definition for *list* is given below:

$$\begin{aligned} list([], Y, Z) &\leftrightarrow emp \wedge Y = Z \\ list([W|X], Y, Z) &\leftrightarrow (\exists p.\ (Y \mapsto W, p) * list(X, p, Z)) \end{aligned} \tag{2}$$

Note that the first argument denotes a sequence, where sequences are represented using the Prolog list notation. The second and third arguments delimit the corresponding linked-list structure. Note also that the definition excludes cycles. The *list* predicate is illustrated in Fig 1.

In addition to the *list* predicate, the specification of list reversal requires additional constraints on sequences. These constraints can be achieved via the following definitions for sequence reversal and concatenation:



The picture above corresponds to $list(\alpha, i, j)$, where i and j delimit the singly linked-list representing α, *i.e.* the sequence $[\alpha_1, \alpha_2, \ldots, \alpha_n]$.

Fig. 1. An acyclic singly linked-list

$$rev([]) = [] \qquad app([], Z) = Z$$
$$rev([X|Y]) = app(rev(Y), [X]) \qquad app([X|Y], Z) = [X|app(Y, Z)]$$

Using these definitions, we can now specify the list reversal program as shown in Fig 2. A key burden in automating the verification of such a program is the discovery of R, the loop invariant. In our proposal, we consider the invariant as the sum of two parts, *i.e.* its *shape* and *content*:

$$R : (\exists \alpha, \beta.\ \underbrace{list(\alpha, i, nil) * list(\beta, j, nil)}_{\textbf{shape}} \wedge \underbrace{rev(\alpha_{init}) = app(rev(\alpha), \beta)}_{\textbf{content}})$$

For each part we propose the use of a different discovery technique. However, we will argue that their exists real cooperation between the two techniques.

3 Symbolic Evaluation and Middle-out Proof Planning

As mentioned in §1, our proposal builds upon a technique described in [18] for automatically inferring loop invariants in separation logic for imperative list-processing programs. Starting with a programmer supplied precondition, symbolic evaluation is used to calculate the effect of the code on the heap. In terms of loops, the repeated evaluation of the loop body is undertaken in order to identify a fixed point. The search for a fixed point is not guaranteed to converge, as a consequence the technique is not complete. It should be noted that the work presented in [18] describes an approach to dealing with content as well as shape via predicate abstraction. We believe, however, that our approach is ultimately more general because it is based upon theorem proving.

Proof planning [4] is a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans*. A distinctive feature of proof planning is *middle-out reasoning* [7], a technique where meta-variables, typically higher-order, are used to delay choice during the search for a proof. An application of middle-out reasoning involves planning a proof within the context

```
{P}
j := nil;
{R}
while not(i = nil) do
   k := [i+1];
   [i+1] := j;
   j := i;
   i := k
od
{Q}
```

$P : (\exists \alpha.\ list(\alpha, i, nil)\ \wedge \alpha_{init} = \alpha)$

$Q : (\exists \beta.\ list(\beta, j, nil)\ \wedge rev(\alpha_{init}) = \beta)$

$R : (\exists \alpha, \beta.\ list(\alpha, i, nil) * list(\beta, j, nil)\ \wedge rev(\alpha_{init}) = app(rev(\alpha), \beta))$

Note that α_{init} denotes a ghost variable, which enables the content of the initial linked-list to be referenced within the invariant and postcondition. Note also that the notation `[x]` denotes the value stored at address `x` within the heap.

Fig. 2. Specification and code for in-place list reversal

of a schematic conjecture. Middle-out reasoning has been used to greatest effect within the context of *proof critics* [9], a technique that supports the automatic analysis and patching of failed proof attempts. Proof critics have been applied successfully to the problems of inductive conjecture generalization [11, 12] and loop invariant discovery [14]. Most recently, the loop invariant discovery techniques have been integrated with light-weight program analysis and applied to industrial strength problems [13].

4 Proposed Cooperation

Given a program, annotated with pre- and postconditions, our proposed approach aims to:

1. Discover a shape invariant via the CMU symbolic evaluation technique [18].
2. Specify a loop invariant by combining the shape invariant with a schematic content invariant.
3. Instantiate and verify the loop invariant via middle-out proof planning.

A key feature of this proposal is that we treat the process by which the shape invariant is discovered as an oracle. That is, we are not concerned about the soundness of the discovery process since the shape invariant is verified via proof planning. Note that we are aiming to maximize the level of automation that can be achieved. Where we see a need for interaction is at the level of defining inductive predicates, such as *list*.

4.1 Symbolic Evaluation

We first give a brief outline of the CMU technique and its application to the reversal example. For a full account see [18]. Abstracting away the data content of the *list* predicate, a simplified predicate called *ls* is used for encoding memory descriptions:

$$ls(p_1, p_2) \equiv (\exists x, k.\ p_1 \mapsto (x, k) * ls(k, p_2)) \vee (p_1 = p_2 \wedge emp)$$

Where a list is known to be non-empty, then ls^+, a special case of the ls predicate can be used:

$$ls^+(p_1, p_2) \equiv (\exists x, k.\ p_1 \mapsto (x, k) * ls(k, p_2))$$

Note that these definitions are hardwired within the CMU technique. The symbolic evaluation algorithm operates on a symbolic description of a memory, *i.e.* given a memory descriptor and a command the algorithm computes a new symbolic description corresponding to the execution of the command. Programs are annotated with a precondition of the form $H \wedge P$, where H denotes the shape of the heap while P imposes constraints on the store. Memory descriptors take the form of a triple, and are represented as follows:

$$(\exists \bar{v}.\ (H'; S; P'))$$

Here S denotes a list of equalities of the form $x = v$, where v is a new symbolic variable, corresponding to the program variable x. H' and P' are generated from H and P by replacing all occurrences of x by the corresponding v respectively. Where the algorithm gets interesting is when it encounters a loop. The algorithm simply iterates the approach used for straight-line code. That is, the loop body is symbolically evaluated for a small number of iterations. At the end of each iteration an attempt is made to weaken the resulting memory description via the application of a fold step. This iterative process terminates if the description of the memory converges to a fixed point. However, the process may diverge so termination is not guaranteed. Theorem proving is used to check that a fixed point has been reached. For the list reversal program we derived the following loop invariant using the CMU technique by hand:

$$(ls(i, nil) \wedge j = nil) \ \vee \ (ls(i, nil) * (j \mapsto (\_, nil))) \ \vee \ (ls(i, nil) * ls^+(j, nil))$$

Intuitively one can see that this is equivalent to an abstract version of R, our hand-crafted invariant:

$$list(\_, i, nil) * list(\_, j, nil) \tag{3}$$

Note that we will return the gap which exists between the machine generated and hand-crafted invariants in §7.

4.2 Middle-out Proof Planning

We now consider the proof planning perspective. Recall that the postcondition associated with the list reversal program takes the form:

$$(\exists \beta.\ list(\beta, j, nil)\ \wedge rev(\alpha_{init}) = \beta)$$

This asserts that if the program terminates then j will point to a list containing the sequence β. The fixed point computation described above tells us that within the loop an auxiliary list, pointed to by i, is introduced. If we let α denote the contents of the auxiliary list, then what remains to be discovered is the relationship between α and β. This is where middle-out reasoning is applicable. We use meta-variables to speculate the relationship between α and β, as shown below:

$$(\exists \alpha, \beta.\ list(\alpha, i, nil) * list(\beta, j, nil) \wedge rev(\alpha_{init}) = F_1(\beta, \alpha)) \tag{4}$$

Note that F_1 denotes a second-order meta-variable. Verifying the loop using this schematic invariant gives rise to the following schematic verification condition:

Given:

$$(\exists \alpha', \beta'.\ list(\alpha', i, nil) * list(\beta', j, nil)\ \wedge rev(\alpha_{init}) = F_1(\beta', \alpha'))\ \wedge \neg(i = nil)$$

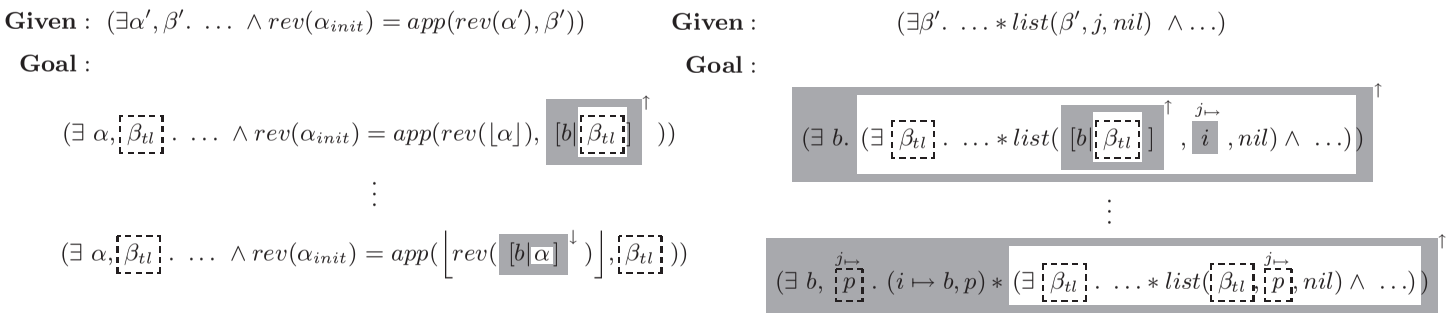
Goal:

$$\begin{array}{l}(\exists\ x_2.\ (\exists x, y.\ (i \mapsto x, y) * ((i \mapsto x, j) \mathrel{-\!\!*} \\ \qquad (\exists\ \alpha, \beta.\ list(\alpha, x_2, nil) * list(\beta, i, nil) \wedge \\ \qquad\qquad rev(\alpha_{init}) = F_1(\beta, \alpha))))\ \wedge (\exists x_1.\ (i \hookrightarrow x_1, x_2)))\end{array}$$

The proof planning of this conjecture can be viewed in terms of two parts. Firstly, the verification of the shape invariant. Secondly, the discovery and verification of the content invariant. The proof plan, known as *rippling*, is applicable to both parts. Rippling is a rewrite strategy that uses a difference reduction criterion to select applicable rewrite rules. This difference reduction criterion is made explicit via meta-level annotations. Rippling is typically used within a functional context, however, a relational version also exists. A complete account of rippling can be found in [6]. Within the context of separation logic, we require a hybrid form of rippling, *i.e.* functional rippling extended to deal differences arising from pointer references. To achieve this, the meta-level annotations need to be extended. We summarize the key features of functional rippling together with our proposed hybrid extension in Fig 3. For reasons of space we do not present the verification of the shape invariant, we focus here on the automatic discovery of the content invariant. In Fig 4 we show how rippling, via middle-out reasoning, can discover an appropriate instantiation for (4), *i.e.*

$$(\exists \alpha, \beta.\ list(\alpha, i, nil) * list(\beta, j, nil) \wedge rev(\alpha_{init}) = app(rev(\alpha), \beta)) \tag{5}$$

The process of instantiation is incremental, where rippling constrains the possible unifiers which are considered during the automatic search for a proof.



Shading is used above to highlight the difference between the goal and the given hypothesis. These meta-level annotations are known as *wave-fronts*. The upward and downward arrows are used to guarantee termination of rippling. In the purely functional case, a wave-front contains at least one unannotated subterm corresponding to a subterm within the given hypothesis. This is known as a *wave-hole*. Note that in the case of pointer rippling, wave-holes do not occur. Instead we record the corresponding pointer reference within the given hypothesis. Note also that existential variables denote *potential* wave-fronts and are represented here by dotted boxes, *i.e.* additional wave-fronts can be generated via existential witness terms. Wave-fronts are manipulated by annotated rewrite rules which are known as *wave-rules*. Example wave-rules are shown below:

$$app(X, \boxed{[Y|Z]}^{\uparrow}) \Rightarrow app(\boxed{app(\boxed{X}, [Y])}^{\downarrow}, Z) \tag{6}$$

$$\boxed{app(\boxed{rev(Y)}, [X])}^{\downarrow} \Rightarrow rev(\boxed{[X|Y]}^{\downarrow}) \tag{7}$$

$$\boxed{(\exists V.\ (\exists W.\ P(list(\boxed{[V|W]}^{\uparrow}, \overset{Z\mapsto}{\boxed{X}}, Y))))}^{\uparrow} \Rightarrow \boxed{(\exists V.\ (\exists W.\ P(\boxed{(\exists\ \overset{Z\mapsto}{\boxed{U}} .\ (X \mapsto V, U) * list(W, \overset{Z\mapsto}{\boxed{U}}, Y))}^{\uparrow})))}^{\uparrow} \tag{8}$$

Note that (7) and (6) arise from the recursive definition of *rev* and a property of *app* respectively, while (8) is derived from (2). The process of annotating goals and generating wave-rules is fully automatic. Note also the $\lfloor \ldots \rfloor$ around the existential α. This meta-level annotation, known as a *sink*, is used as part of the *rippling-sideways* strategy, in which wave-fronts are directed towards quantified variables within the given hypothesis. Previously sinks have corresponded to universally quantified variables, but here the notion of the sink is generalized to include existentially quantified variables.

Fig. 3. Functional and pointer rippling

4.3 Summary

To summarize, we have argued that the CMU technique can be used as the basis for generating a shape invariant corresponding to (3). Combining (3) with the given postcondition we generated a schematic loop invariant, *i.e.* (4). We then described how middle-out proof planning could be used to verify the schematic invariant, with the side-effect of instantiating the content part of (4) to give (5).

5 Benefits of Cooperative Reasoning Processes

Our proposal involves two distinct reasoning processes, i) symbolic evaluation, and ii) middle-out proof planning. The achievements of each process can be summarized as follows:

- Discovery of the shape invariant is achieved via symbolic evaluation.
- Verification of the shape invariant is achieved via proof planning.
- Discovery of the content invariant is achieved via middle-out reasoning.

In terms of communication, the flow of information is in one direction, *i.e.* the achievements of the symbolic evaluation process are communicated to the proof planning process, where they are extended and verified. There is also potential, however, for bi-directional communication. As mentioned in §4.1, symbolic evaluation may diverge, and thus fail to generate a shape invariant. An analysis of such a failure may reveal a missing rewrite rule (lemma) which will overcome the divergence via a fold step. This kind of failure analysis has been implemented within a *divergence critic* [25], and used successfully to automate the discovery of inductive lemmas. Building upon the divergence critic, symbolic evaluation and proof planning could overcome a divergence via the following bi-directional communication:

- The symbolic evaluation process detects divergence and communicates failure information to the proof planning process;
- The proof planning process analyses the failure information, via the divergence critic, and attempts to discover a patch, *i.e.* a rewrite rule that will allow symbolic evaluation to converge.

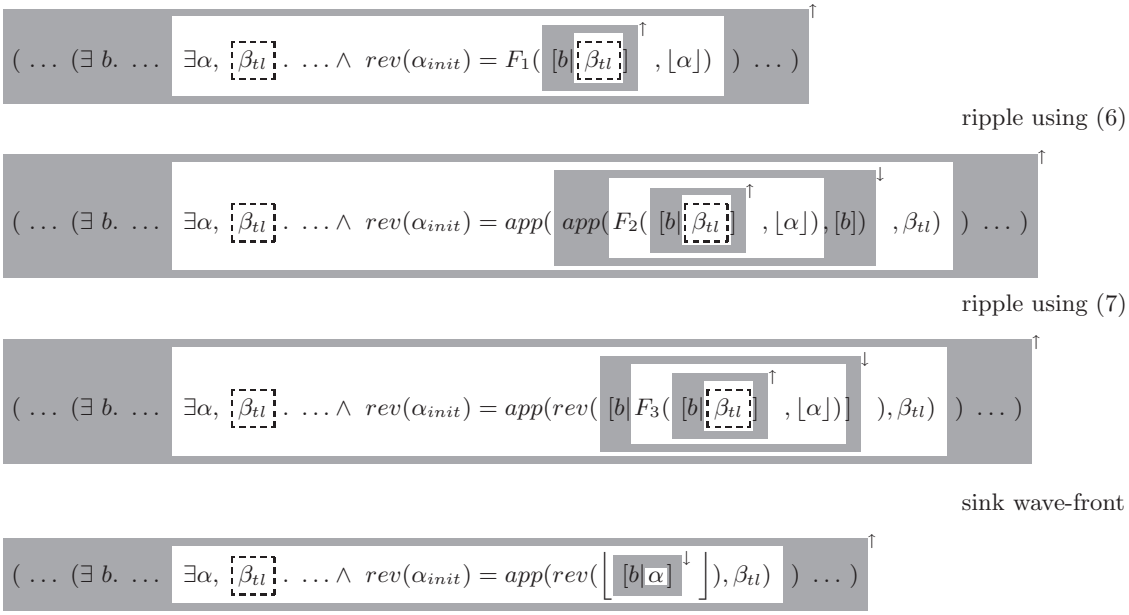
As highlighted in [5], the benefits of adopting a cooperation approach lie in the value added that can be achieved, *i.e.*

> *"By complementing each other, cooperating reasoning processes can achieve much more than they could if they only acted individually."*

But is this true of our proposal? Playing Devil's advocate, rippling via middle-out reasoning could discover shape invariants. However, the syntactic nature of rippling makes it better suited to discovering functional properties, where the term structure is more deeply nested. So the symbolic evaluation process mitigates a weakness within middle-out reasoning. Likewise the correctness of the

Given: $(\exists \alpha', \beta'.\ \ldots\ \wedge rev(\alpha_{init}) = F_1(\beta', \alpha'))\ \wedge\ \neg(i = nil)$ (9)

Goal and ripple:



Note the occurrence of the second-order meta-variable F_1 within the initial goal. As the middle-out proof planning proceeds, the constraints imposed by rippling are used to incrementally instantiate F_1. At the end of the ripple, F_1 has become instantiated to be $\lambda x.\lambda y.app(rev(y), x)$. In order to complete the ripple, the sink term, *i.e.* $[b|\alpha]$, must be matched against α' within (9). Since we are dealing with an existential sink, this requires additional inference involving the unfolding of α'. This unfolding step is sound because $list(\alpha', i, nil)$ and $\neg(i = nil)$ are given hypotheses.

Fig. 4. Automatic discovery of the content invariant via middle-out rippling

shape invariant could be dealt with as an add-on to the symbolic evaluation process, as is the case in [18]. However, proof planning provides a uniform framework in which to verify the invariant as a whole. In terms of discovering content invariants, the evidence of our previous work suggests that middle-out reasoning and rippling are well suited to this task.

6 Related Work

Closely related to the work presented in [18] is the SMALLFOOT tool [1]. SMALLFOOT is an experimental tool that supports the automatic verification of shape properties specified in separation logic. Based upon symbolic evaluation [2], SMALLFOOT verifies a program against user supplied pre- and postconditions. Unlike the CMU technique, SMALLFOOT does not attempt to discover loop invariants.

Out with separation logic, the Pointer Assertion Logic Engine (PALE) [19] allows relatively complex specifications to be expressed in monadic second-order logic. Verification is achieved via the MONA tool [16]. PALE requires a user to supply loop invariants. In contrast, the TVLA tool [23] provides significant verification automation, while not requiring loop invariants, TVLA may require a user to supply *instrumentation predicates*, *i.e.* predicates that encode *local* properties of datatypes. Instrumentation predicates, however, may be reused between applications.

With regards to proof planning, and in particular middle-out reasoning, the technique known as *lazy thinking* [3] is closely related to our proposal. Lazy thinking, like middle-out reasoning, allows the discovery process to proceed hand-in-hand with the process of verification. Lazy thinking, as applied to program synthesis, employs algorithm schemas in order to constrain search while middle-out reasoning is typically applied within the context of rippling, where the meta-level annotations provide the constraints. The work reported in [15] also makes use of second-order meta-variables to support inductive conjecture generalization within the context of proving properties of tail-recursive functions. This is similar to our earlier work on accumulator generalization [10–12] and tail-invariant discovery [14].

7 Future Work

The proposal presented in this paper represents work in progress. We have identified a number of issues that need to be addressed:

- The meta-level annotations of rippling need to be extended to deal with pointer references and existential sinks.
- The gap between the machine generated and "hand-crafted" shape invariants needs further investigation.

Achieving the proposal, however, will involve more than just these issues. We require a theorem proving environment upon which to develop our proof plans.

A strong candidate is Schirmer's verification environment for sequential imperative programs [24]. Integrated within Isabelle/HOL [20], Schirmer provides a generic framework for modelling and verifying imperative programs. Moreover, the framework has already been extended with separation logic, for the purposes of reasoning about pointer programs written in C [17]. Building upon Isabelle/HOL also makes sense from the proof planning perspective given that IsaPlanner [8], the current state-of-the-art proof planner, is Isabelle based. In terms of the symbolic evaluation component, if possible, we would like to build upon either Smallfoot or the CMU tool.

8 Conclusion

Building upon separation logic, we propose a cooperative approach to loop invariant discovery for pointer program proof. The proposal involves the integration of an existing shape invariant discovery technique with middle-out proof planning, a technique which has a proven track-record in terms of reasoning about loops and automating loop invariant discovery. Our belief is that adopting a cooperative approach will deliver significant benefits in terms of search control. Our next step will be to implement the approach so that this hypothesis can be tested.

Acknowledgements: The research reported in this paper is supported by EPSRC grants GR/S01771 and EP/E005713. Thanks go to Alan Bundy, Ianthe Hind, Paul Jackson, Ewen Maclean and Alan Smaill for their feedback an encouragement with this work. Thanks also goes to two anonymous WING-07 referees for their constructive feedback.

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
2. J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
3. B. Buchberger and A. Craciun. Algorithm synthesis by lazy thinking: Examples and implementation in Theorema. *Electronic Notes in Theoretical Computer Science*, 93:24–59, 2004.
4. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
5. A. Bundy. Cooperating reasoning processes: more than just the sum of their parts. In M. Veloso, editor, *Proceedings of IJCAI 2007*, pages 2–11. IJCAI Inc, 2007. Acceptance speech for Research Excellence Award.
6. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

7. A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.
8. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.
9. A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.
10. A. Ireland and A. Bundy. Extensions to a generalization critic for inductive proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.
11. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.
12. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.
13. 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.
14. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.
15. D. Kapur and N. A. Sakhanenko. Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2003.
16. N. Klarlund and A. Møller. MONA version 1.4 user manual. BRICS Notes Series NS-01-1, Dept. of Computer Science, University of Aarhus, 2001.
17. G. Klein, H. Tuch, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, January 2007. To appear.
18. S. Magill, A. Nanevski, E. Clarke, and P. 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, Charleston, SC, 2006.
19. A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of PLDI'01*, pages 221–231, 2001.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

21. P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
24. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In G. Klein, editor, *Proc. NICTA Workshop on OS Verification 2004*, 2004.
25. T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

Assertion-based Loop Invariant Generation

Mikoláš Janota*

School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Abstract. Many automated techniques for invariant generation are based on the idea that the invariant should show that something "bad" will not happen in the analyzed program. In this article we present an algorithm for loop invariant generation in programs with assertions using a weakest precondition calculus. We have realized the algorithm in the extended static checker ESC/Java2. Challenges stemming from our initial experience with the implementation are also discussed.

1 Introduction

Automated invariant generation techniques face two major challenges. The first challenge is the invariant itself, i.e., which formulas should be suggested as invariants. The second challenge is time complexity. Programs in practice require nontrivial, e.g., quantified, invariants; to be able to reason about such invariants, techniques commonly rely on an automated theorem prover. Since invocations of a theorem prover are time expensive, it is necessary to carefully limit the number of calls to the prover. In this article we introduce a light-weight technique that derives loop invariants from assertions, which express desired properties of the program. We have implemented the algorithm in the extended static checker ESC/Java2 [11]. The implementation operates on the intermediate representation of the program, where program annotations are translated into assertions and assumptions. Thus, the implementation takes into account both the program code and its specification.

The rest of the paper is structured as follows. Section 2 introduces the problem and its context. Section 3 describes the basic version of the loop invariant generation algorithm and Sections 4 illustrates the algorithm on an example. Section 5 describes an extension of the algorithm. Section 6 discusses the implementation and Section 7 lists related work. Finally, Section 8 concludes and proposes future work.

* This work was funded by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre." This work was partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

2 Background

The Java Modeling Language (JML) is a first–order logic annotation language for Java programs [12]. JML is embedded into Java programs as a special form of comments. The sign '@' indicates that a particular comment is a JML annotation. Figure 1 illustrates the use of JML.

```
//@ requires a != null;
//@ requires (\forall int x; (0 <= x & x < a.length) ==> a[x] != null);
void setToZero(int [][] a) {
  /*@ loop_invariant
    @   (\forall int x; (0 <= x & x < a.length) ==> a[x] != null);
    @ loop_invariant a != null;
    @ loop_invariant i >= 0; */
  for (int i = 0; i < a.length; i++) {
    /*@ loop_invariant j >= 0;
      @ loop_invariant a != null;
      @ loop_invariant
      @   (\forall int x; (0 <= x & x < a.length) ==> a[x] != null); */
    for (int j = 0; j < a[i].length; j++)
      a[i][j] = 0;
  }
}
```

Fig. 1. JML-annotated Java method.

The goal of ESC/Java2 is to determine whether JML-annotated Java code conforms to its annotations. For the given annotated program, ESC/Java2 generates a logic formula, called *verification condition* (VC), using a weakest precondition or a strongest postcondition calculus. Subsequently, it sends the VC to an automated theorem prover. If the theorem prover does not prove the VC valid, ESC/Java2 provides warnings describing how the program violates the JML-specification or how the program might cause run-time exceptions (e.g., by null-pointer dereferencing).

ESC/Java2 performs the translation from a JML-annotated program to the VC in several stages. For the space limitations we do not provide details for the whole translation process here and the interested reader is referred to the relevant documentation [7]. For the understanding of the article, however, it suffices to present one of the intermediate representations called *guarded command language* [13].

In the rest of the paper we will assume that we have the following. A set of program variables Var and a language of side-effect free expressions $\mathcal{E}$. We intentionally leave the grammar of expressions unspecified as the presented algorithm does not depend on their structure. A first-order logic language that contains at least the boolean expressions in $\mathcal{E}$, propositional connectives, and

the constants true, false. A theory T, known as *background predicate*, axiomatizing the semantics of expressions and we will write $T \models f$ to denote that the formula f is valid in the theory T. Additionally, we assume that an automated theorem prover is available. The prover accepts queries of the form $T \models f$ and responds whether the query was proven valid or not.

The following defines the grammar of the guarded command language:

$$\begin{aligned} cmd := \; & x \leftarrow expr \mid \mathbf{assume}\, f \mid \mathbf{assert}\, f \mid \mathbf{skip} \mid \\ & cmd \,[]\, cmd \mid cmd; cmd \mid \{\mathcal{I}\}\mathbf{while}\; expr\; \mathbf{do}\; cmd \end{aligned}$$

where $x \in Var$, $expr \in \mathcal{E}$, and $\mathcal{I}, f$ are logic formulas where the only free variables are in Var.

Informally, if the execution reaches $\mathbf{assume}\, f$ then f can be assumed and, if f does not hold, the execution blocks. If $\mathbf{assert}\, f$ is reached and f does not hold, an error occurs, in this case we say that the program *went wrong*. The command $C_1 \,[]\, C_2$ is a nondeterministic *choice* between C_1 and C_2; $C_1; C_2$ is a *sequence* of commands, $\{\mathcal{I}\}\mathbf{while}\; c\; \mathbf{do}\; B$ is a loop with the invariant $\mathcal{I}$, condition c and body B representing a program that repeats B while c holds. The command $x \leftarrow expr$ *assigns* the value of the expression $expr$ to the variable x.

In ESC/Java2, the guarded command language is used to represent Java code and its pertaining JML annotations[1]. For example, a call to a method is translated to an $\mathbf{assert}\, P$ representing the precondition of the called method and an $\mathbf{assume}\, Q$ representing its postcondition; an if statement is translated to a choice between the "then" and the "else" branch; the domains of variables contain Java primitive types and the model of the heap.

To formally capture the semantics of the guarded command language we will define a weakest precondition predicate transformer. We will do this is two steps, the first step is called *desugaring* and it defines the semantics of loops in terms of other commands. The second step defines a weakest precondition calculus on the desugared form of the guarded command language.

We will use the following auxiliary functions. The function *havoc* resets values of the given variables:

$$havoc(\{x_1, \ldots, x_n\}) \equiv x_1 \leftarrow v'_1; \ldots; x_n \leftarrow v'_n$$

where $v'_1, \ldots, v'_2$ are fresh variables. And, let $targets(C)$ be a function that returns all variables that might be modified by the command C. The desugaring is captured by the function desugar defined as follows:

$$\begin{aligned} &\text{desugar}\,(\{\mathcal{I}\}\mathbf{while}\; c\; \mathbf{do}\; B) \equiv \\ &\qquad \mathbf{assert}\, \mathcal{I}; \\ &\qquad \text{havoc}(\text{targets}(B)); \mathbf{assume}\, \mathcal{I}; \\ &\qquad ((\mathbf{assume}\, c; \text{desugar}(B); \mathbf{assert}\, \mathcal{I}; \mathbf{assume}\, \text{false}) \,[]\, (\mathbf{assume}\, \neg c)) \\ &\text{desugar}(C_1 \,[]\, C_2) \equiv \text{desugar}(C_1) \,[]\, \text{desugar}(C_2) \\ &\text{desugar}(C_1; C_2) \;\; \equiv \text{desugar}(C_1); \text{desugar}(C_2) \\ &\text{desugar}(C) \qquad \equiv C, \text{ all other commands} \end{aligned}$$

[1] ESC/Java2 supports full Java 1.4 source code.

$$
\begin{array}{ll}
\mathrm{wlp}(\textbf{skip}, N, W) & \equiv N \\
\mathrm{wlp}(x \leftarrow expr, N, W) & \equiv N[x \mapsto expr] \\
\mathrm{wlp}(\textbf{assume}\, f, N, W) & \equiv f \Rightarrow N \\
\mathrm{wlp}(\textbf{assert}\, f, N, W) & \equiv (f \Rightarrow N) \wedge (\neg f \Rightarrow W) \\
\mathrm{wlp}(C_1; C_2, N, W) & \equiv \mathrm{wlp}(C_1, \mathrm{wlp}(C_2, N, W), W) \\
\mathrm{wlp}(C_1 \,[\!]\, C_2, N, W) & \equiv \mathrm{wlp}(C_1, N, W) \wedge \mathrm{wlp}(C_2, N, W)
\end{array}
$$

Fig. 2. Weakest precondition calculus.

Intuitively, the initial assertion in the desugaring of a loop is a check for the invariant when the execution reaches the loop. The left branch of the choice command represents an arbitrary iteration of the loop and the **assume** $\neg c$ represents the termination of the loop.

The predicate transformer $wlp(C, N, W)$ defined in Figure 2 captures the semantics of the desugared guarded command language. For a command C and predicates N and W, if $wlp(C, N, W)$ holds then N holds if C terminates normally, W holds if C goes wrong or C does not terminate at all. For example, **assume** false never terminates or goes wrong.

In this context, the verification condition is defined so that the given program does not go wrong for any possible output. This is captured by the following definition.

Definition 21 *1. For a program C, the* verification condition *is the formula:*

$$\mathrm{wlp}(desugar(C), true, false)$$

2. A program C conforms to its specification *if and only if:*

$$T \models \mathrm{wlp}(desugar(C), true, false)$$

3 Invariant Inference from Assertions

The presented technique is motivated by a simple observation. In the subcommand of the desugared version of a loop representing an arbitrary iteration all we know about the loop's targets is whatever is in the loop invariant. Since the ultimate goal of the verification process is to show that the given program conforms to its specification, we need to ensure that the loop invariant is strong enough to prove that the assertions inside the loop do not go wrong (see Sections 4, 5 for examples of assertions and invariants).

To derive such loop invariants, the algorithm back-propagates assertions outwards to the outermost loop using the weakest precondition calculus. To explain the algorithm, we introduce the following terminology. We split the loops in the analyzed program into *layers*. The layer 0 contains exactly the loops that are not inside any other loop in the program. Loops in the layer i are exactly those loops nested in i other loops. We will refer to the loops in the layer 0 as *loop-nests* and we will use L_i to denote that the loop L_i is in the layer i. We will say that a

```
INFER-INVARIANTS(L_0, ..., L_i : loops, assert f : command hosted by L_i)
    loc := location of the given assertion in L_i
    I := f
    preserves := true
    for k := i downto 0
          do I := back-propagate I from loc to entry of L_k
             if (L_k preserves the invariant I)
                then add I to the invariant of L_k
                else  preserves := false
                      break
             if k ≠ 0
                then loc := location of the entry of L_k in L_{k-1}

    succeeded := preserves ∧ (I holds at the entry of L_0)
    if ¬succeeded
       then remove the added invariants
```

Fig. 3. Deriving invariants from an assertion.

.

loop L_i *hosts* the command C if and only if L_i contains C and there is no loop L_k with $k > i$ that contains C. We will say that a loop L *preserves* a formula f to express that if f held before an arbitrary iteration of L then it will hold once the iteration terminates; we will discuss the exact meaning of this term and back-propagation in Section 3.1.

The heart of the algorithm is the derivation of invariants from a given assertion. Consider the following. An assertion **assert** f is hosted by a loop L_i. And let $L_0, \ldots, L_i$ be a sequence of nested loops, i.e., L_k hosts L_{k+1} for $k \in 0 \ldots i-1$. In this scenario, we use the asserted formula f to infer invariants for the loops $L_0, \ldots, L_i$. This inference process starts by back-propagating the formula f to the top of L_i yielding a formula f_i. Subsequently, we determine whether f_i is preserved by L_i. If that is true, the process continues by back-propagating f_i to the top of L_{i-1}, yielding a formula f_{i-1}. This is repeated until the outermost loop L_0 is reached with the formula f_0. Finally, to verify that the suggested formulas are indeed invariants, we test whether f_0 holds at the entry of the loop-nest L_0. This process is captured by the pseudo-code in Figure 3.

Up to this point we have described how to infer an invariant from a single assertion. The whole analysis infers invariants from all assertions that are contained in any loop starting from the assertions in the innermost layers. This is captured by the following pseudo-code.

```
ANALYZE(C : command)
    for each loop-nest L_0 in C, using breadth-first search
         do ANALYZE(L_0)
```

ANALYZE($L_0, \ldots, L_i : loop$)
  for each loop K hosted in L, using breadth-first search
    do ANALYZE($L_0, \ldots, L_i, K$)
  for each assert f hosted in L, using breadth-first search
    do INFER-INVARIANTS($L_0, \ldots, L_i,$ **assert** f)

So far we have not explained the following: how expressions are back-propagated, how it is determined that a formula preserves a loop, and that a formula holds at the entry of a loop-nest. The following section provides details on these subjects.

3.1 Algorithm Details

For the purpose of this section we consider a control flow graph representation of the desugared guarded command language. We require that the control flow graphs have the following properties. Each node in the graph is labeled either with the command **skip**, **assume** f, **assert** f, or $x \leftarrow e$. The graph is directed acyclic and it has exactly one entry node, a node that dominates all the other nodes in the graph. Any subgraph G_L resulting from the desugaring of a loop has one entry node and one exit node. The entry node dominates all nodes in G_L and any path from any node in G_L to a node that is not in G_L contains the exit node (a postdominator). It is easy to observe that it is possible to construct such a graph for any desugared command. For example, the graph of the choice command has a "diamond" structure where the top and bottom tips are labeled with **skip** and the sides are graphs representing each choice.

In the context of a control flow graph G, back-propagation of the formula f from the node r to the node n is computed by the following function:

$$\begin{aligned} pre_G(n, r, f) &\equiv f, \text{if } n = r \\ &\equiv \text{wlp}(C_n, \textstyle\bigwedge_{c \text{ is a child of } n \text{ in } G} pre_G(c, r, f), \text{true}), \text{ otherwise} \end{aligned}$$

where C_n denotes the command labeling the node n. In plain English, the property of the function pre_G is that if $pre_G(n, r, f)$ held in n and the normal execution reached r then f holds.

Now we can describe back-propagation in an iteration of the loop in procedure INFER-INVARIANTS (Figure 3). For a loop $L_k \equiv \{f\}$**while** c **do** B it constructs a graph G with the entry node n_k of the command desugar(**assume** $c; B$). It identifies the entry node n_{k+1} of the subgraph of G that resulted from the desugaring of the loop L_{k+1}. Finally, it sets the suggested invariant $\mathcal{I}$ to $pre_G(n_k, n_{k+1}, \mathcal{I})$.

To determine whether the given invariant $\mathcal{I}$ holds at the entry of a loop-nest L_0, we construct a graph G with the entry node n from the desugared version of the whole program being analyzed, identify the entry node n_0 of the subgraph of G resulting from L_0, and send the query $T \models pre_G(n, n_0, \mathcal{I})$ to the theorem prover.

To determine whether the loop $L \equiv \{\mathcal{I}\}$**while** c **do** B preserves a formula f we send the following query to the theorem prover:

$$T \models f \Rightarrow \text{wlp}(C; \text{havoc}(\text{targets}(B)); \textbf{assume}\, f; \textbf{assume}\, c; \text{desugar}(B), f, \text{true})$$

where C is the loop's *context*, i.e., the preceding commands in the desuagared version of the program. For example, the inner loop in the program

$$C_0; (\{\mathcal{I}_0\}\textbf{while}\ c_0\ \textbf{do}\ (C_1; (\{I_1\}\textbf{while}\ c_1\ \textbf{do}\ B)))$$

has the following context:

$$\text{desugar}(C_0); \text{havoc}(\text{targets}(C_1; \{I_1\}\textbf{while}\ c_1\ \textbf{do}\ B)); \textbf{assume}\ (\mathcal{I}_0 \wedge c_0)$$

4 Example of Back-propagation

This section illustrates the analysis presented in the last section on an example. Consider the program in Figure 4, written in pseudo-code, which takes a two-dimensional array as its input and sets all its elements to zero.

```
 1: INPUT: a: array[1...N][1...M] of ℕ
 2: VAR i, j: ℕ;
 3: i ← 1;
 4: while i ≤ N do
 5:    j ← 1;
 6:    while j ≤ M do
 7:       ASSERT 1 ≤ i ∧ i ≤ N;
 8:       ASSERT 1 ≤ j ∧ j ≤ M;
 9:       a[i][j] ← 0;
10:       j ← j + 1;
11:    end while
12:    i ← i + 1;
13: end while
```

Fig. 4. Example of code with assertions.

The assignment to an element of the array requires that the values of i and j are in the bounds of the array a. This is captured by the two assertions. We will illustrate the steps of the algorithm on the first assertion.

The algorithm first back-propagates the assertion to the top of the inner loop, which results in the following formula:

$$(j \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

Subsequently, the algorithm tests whether this formula is preserved by the body of the inner loop. This follows from the fact that the inner loop does not change the value of i.

In the next step the algorithm propagates the previously obtained formula to the top of the outer loop. This results in the following:

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

This can be simplified to the equivalent formula:

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i)$$

Further, the algorithm tests whether the outer loop preserves the simplified formula. This follows from our previous result that the inner loop preserves the desired property and that i is increased.

The final step tests whether the suggested invariant is established by the code preceding the outer loop, i.e., the validity:

$$T \models (1 \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq 1)$$

this immediately follows from the reflexivity of $\leq$.

At this stage we know that

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i)$$

is an invariant of the outer loop. And,

$$(j \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

is an invariant for the inner loop. Together, these invariants guarantee that the first assertion is not violated (does not go wrong).

5 Invariant Alterations

This section describes an extension of the presented invariant inference technique called *invariant alterations*. It is obvious that in many cases the invariant suggested by the procedure Infer-Invariants (Figure 3) will not be strong enough to prove that it is preserved by the loop even if it is an invariant of that loop. A possible improvement is to alter the suggested invariant based on some heuristic.

The following is an example of how to obtain a set of alterations from a formula f:

$$\{(\forall v' \bullet f[v \mapsto v']) \bullet \ v \text{ is free in } f \text{ and } v' \text{ is a fresh variable}\}$$

All these alterations are stronger than the original formula f. Therefore, if we can show that one of these alterations is an invariant, it will still guarantee that the original assertion does not go wrong.

The rest of this section illustrates the use of alterations on an example. Consider the Java code in Figure 1 without the loop invariants. The assertions that this code yields are captured in the pseudo-code in Figure 5. Please note that any access to a pointer or array has to be prepended with the pertaining assertion, including the loop guards. The precondition of the method is translated into **assume** commands.

```
1: INPUT: a: array[][] of int
2: VAR i, j: int;
3: assume a ≠ null;
4: assume ∀m : int•m ≥ 0 ∧ m < a.length ⇒ a[m] ≠ null;
5: i ← 0;
6: while  assert a ≠ null; i < a.length do
7:    j ← 0;
8:    while assert (a ≠ null ∧a[i] ≠ null ∧0 ≤ i ∧ i < a.length); j < a[i].length do
9:       assert 0 ≤ i ∧ i < a.length;
10:      assert 0 ≤ j ∧ j < a[i].length;
11:      assert a ≠ null ∧a[i] ≠ null
12:      a[i][j] ← 0;
13:      j ← j + 1;
14:   end while
15:   i ← i + 1;
16: end while
```

Fig. 5. Example of code with assertions.

Analogously to the process in the previous example, we obtain the following invariants. Invariants $a \neq \text{null}$ is inferred for both loops. Further, the invariants for the outer loop will be:

$$a \neq \text{null} \wedge i < a.\text{length} \Rightarrow 0 \leq i$$
$$(j \geq 0 \wedge j < a[i].\text{length}) \wedge (0 \leq i \wedge i < \text{a.length}) \Rightarrow a[i] \neq \text{null}$$

And the invaraints for the inner loop will be:

$$a \neq \text{null} \Rightarrow 0 \leq i$$
$$a \neq \text{null} \wedge 0 \leq i \Rightarrow i < a.\text{length}$$
$$a \neq \text{null} \wedge 0 \leq i \wedge i < a.\text{length} \wedge a[i] \neq \text{null} \wedge j < a[i].\text{length} \Rightarrow 0 \leq j$$

The interesting case is the non-nullness of $a[i]$. The algorithm first suggests the following invariant for the inner loop:

$$(0 \leq j \wedge j < a[i].\text{length}) \wedge (0 \leq i \wedge i < \text{a.length}) \Rightarrow a[i] \neq \text{null}$$

which is indeed preserved by the inner loop. However, it is not preserved by the outer loop. If we perform the alteration by quantifying over i, we obtain the following:

$$\forall i' \bullet (0 \leq j \wedge j < a[i'].\text{length}) \wedge (0 \leq i' \wedge i' < a.\text{length}) \Rightarrow a[i'] \neq \text{null}$$

which is preserved by both loops and hence, it is inserted as an invariant for both loops.

6 Implementation

We have implemented the technique described by the method Analyze (Section 3) with the extension of the alterations described in Section 5. The implementation is build as a subcomponent of ESC/Java2, and utilizes the automated

theorem prover Simplify [6]. Based on our initial experiments with the implementation we have added the following enhancements to the algorithm.

Heuristic back-propagation. The back-propagation realized as described in Section 3.1 generates large formulas. Therefore we approximate the transformer wlp as follows. When we back-propagate a formula $\mathcal{I}$ over a node labeled with the command **assume** f or **assert** f, we first substitute the expression f in $\mathcal{I}$ for true, and then apply wlp only if $\mathcal{I}$ and f share at least one free variable.

Simplifications. Even with the heuristic back-propagation, the invariants often contain redundant information. To alleviate this problem we have implemented several straight-forward formula propositional simplifications and the following rule:

$$\frac{\forall x \bullet x = E \Rightarrow F, \text{ where } x \text{ is not free in E}}{F[x \mapsto E]}$$

Assertion breaking. Assertions often come as conjuncts of simpler expressions, e.g., $0 \leq j \wedge j < l$, and sometimes only one of the literals leads to a successful invariant discovery. Therefore, it has proven useful to break the assertion into the individual literals and back-propagate these individually.

The technique has proven successful in finding simple invariants; in particular invariants that guarantee that a certain variable is nonnegative, certain variable is non-null, or that a certain variable has the desired type (required by typecasting). On the other hand, for the technique be useful in practice we believe that taking into account assertions outside loops is necessary.

7 Related Work

One of the first documented implementation of invariant generation utilizing a theorem prover was realized by Suzuki and Ishihata [14]. Similarly to our work, their algorithm aims to prove that a given program does not violate array bounds.

The predominant approach to invariant generation is *abstract interpretation* introduced by Cousot and Cousot [5]. A popular variant of abstract interpretation is *predicate abstraction* [9], where the abstract space is formed by boolean expressions on a finite set of predicates.

One of the key issues in predicate abstraction is to come up with the appropriate set of predicates. An example of a technique that discovers predicates automatically is *counterexample refinement* [10]. This technique tries to refine the model whenever the current model is shown to be too coarse, i.e., when the model contains an error-trace in the abstract space that does not have a corresponding trace in the concrete space. This approach is similar to ours in the sense that is driven by the undesired behavior.

The counterexample refinement has proven suitable for implementation as it was applied to verification of C programs in the tools *BLAST*[2] and *SATABS*[3] [4]

Flanagan and Qadeer utilized predicate abstraction for loop invariant generation in ESC/Java [8]. In this work Flanagan and Qadeer enriched the technique by introducing skolem constants to be able to infer quantified invariants.

Abstract interpretation is used in the Spec# programming system[4] to infer loop invariants [1]; the implementation operates on the intermediate representation in the language Boogie PL. Chang and Leino, members of the Spec# team, in [3] describe how abstract interpretation can be used to infer object invariants. Further, the authors in [2] propose a technique that allows combining different abstractions.

8 Conclusion and Future Work

We have introduced a technique for loop invariant generation from assertions in the context of an automated theorem prover and a weakest precondition calculus. An advantage of the technique is that it requires relatively small number of calls to the theorem prover, compared to predicate abstraction for example. Moreover, it takes into account user's specifications, which focuses the technique on the relevant space of loop invariants. The technique is easy to implement as it does not rely on the underlying theories, such as arithmetics. On the other hand, this property is at the same time a disadvantage since exploiting the information specific to a particular domain enables inferring stronger invariants. It appears that the main disadvantage, however, is the "snowball effect" caused by the weakest precondition calculus increasing the size of the back-propagated formula.

We propose the flowing challenges for future work.

Invariant simplifications. How can we identify irrelevant parts of the suggested invariant and generate simpler invariants?

Invariant alterations. What are the alternations useful in practice? Can other invariant inference techniques be exploited?

Loop postcondition. In the presented work we utilize only the assertions that are inside the investigated loop. This approach could be leveraged by taking into account the assertions after the loop, i.e., the desired postcondition of the loop.

Feedback to the user. The presented implementation operates on the intermediate language. Therefore, the inferred invariants are difficult to interpret for the user. Thus, it is desirable to be able to translate the inferred invariants to the JML notation.

[2] http://mtc.epfl.ch/software-tools/blast
[3] http://www.verify.ethz.ch/satabs
[4] http://research.microsoft.com/specsharp

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceeding of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.
2. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proceeding of 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.
3. Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants, 2005.
4. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25, 2004.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.
6. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
7. Extended Static Checker for Java version 2 (ESC/Java2). At `http://secure.ucd.ie/products/opensource/ESCJava2/`.
8. Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.
9. Suzanne Graf and Hassen Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of 9th International Conference on Computer Aided Verification (CAV'97)*. Springer–Verlag, 1997.
10. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2002.
11. Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, January 2005.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
13. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.
14. Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 1977.

Mechanical Generation of Invariants for FOR-Loops

Stefan Kauer[1] and Jürgen F H Winkler[2]
[1] EADS Deutschland GmbH, Business Unit Defence Electronics, Claude-Dornier-Str.
D-88090 Immenstaad, Germany
Stefan.Kauer@eads.com
[2] Friedrich Schiller University, Institute of Informatics, Ernst-Abbe-Platz 2,
D-07743 Jena, Germany
winkler@informatik.uni-jena.de

Abstract. In the mechanical verification of programs containing loops it is often necessary to provide loop invariants additionally to the specification in form of pre- and postcondition. In this paper we present a method for the mechanical generation of invariants for a class of FOR-loops. The invariant is derived from the postcondition and the final bound of the loop only. The method is applicable if the final bound of the FOR-loop is of a simple form. This is often the case in practice. The incorporation of this method into an automatic program verifier would make the task of the SW engineer easier, because he has only to provide a pre-post-specification for a FOR-loop.

Keywords: mechanical verification, mechanical generation of loop invariants, FOR-loop.

1 Introduction

Program verification involves a great amount of mechanical formula manipulation. If done by hand, this is tedious and, even worse, error prone. Most of the theorems (verification conditions (VC)), which have to be proved, are quite trivial and can therefore be proved automatically by an automatic theorem prover. If all VCs are generated and proved automatically we speak of automatic program verification. A tool which performs automatic program verification is then called an automatic program verifier (APV). Examples of such tools are Boogie [BCD06], FPP [KW99a; Win97] and NPPV [Gumxx]. Other tools use a combination of automatic and interactive theorem proving and can therefore be called semi-automatic program verifiers (SAPV). Examples are KeY [ABB05], SPARK [Bar00] and Theorema [JKP03].

Most tools for the verification of concrete programs use the assertion based method (ABM) for the specification of the required behavior of the program (e.g. Boogie, FPP, KeY, NPPV, SPARK, Theorema). The specification is given by a pair (pre, post) of assertions which refer to entities of the program, and may also refer to entities which belong to the specification only.

ABM allows also the verification of program fragments and therefore can be used by the SW engineer in a continuous manner during program development, and not only for the verification of a finished program in one big step. In this situation, the use of an APV is especially convenient.

In ABM automatic verification on the basis of (pre, post) is rather straightforward for statements like declarations, assignment, IF, and CASE[1]. The verification of loops usually requires also an invariant [Dij76; Hoa72; Tur49; Win98], and for WHILE-loops additionally a termination function [Dij76; Flo67; Tur49]. It would be easier if loops could also be verified by giving only (pre, post). This can be done in two ways: (1) by computing wp(loop, post) resp. sp(pre, loop) and use this in the general verification condition pre $\Rightarrow$ wp(loop, post) resp. sp(pre, loop) $\Rightarrow$ post, or (2) by computing an invariant (and in case of a WHILE-loop a termination function) and perform then the verification using the corresponding VCs. Automatic computation of invariants from the code is seen as difficult in the general case [Bac06: 3]. Stefan Kauer has developed methods for the mechanical verification of classes of loops which are only specified by (pre, post) [Kau99]. For FOR-loops his method is based on the heuristic "replacing a constant in the postcondition by a variable (RCPV)" for the computation of an invariant. For WHILE-loops his method computes wp(loop, post). In this paper we report about the method for the computation of invariants for FOR-loops. For the verification of FOR-loops we use the proof rule of [Win98] which is less restrictive than that of [Hoa72].

An annotated FOR-loop (AFL) in Ada syntax looks like

```
   -- PRE
FOR i in LO..UP LOOP  BODY  END LOOP                    (1)
   -- POST
```

where i is the loop variable, the value of LO is the lower bound and the value of UP is the upper bound of the loop. (1) is an upwards counting loop. Many languages contain also downwards counting loops. In this paper we deal mainly with upwards counting FOR-loops. Downwards counting FOR-loops do not pose new problems and can be treated in an analogous manner [KW00; Win98].

Basic idea. RCPV tries to derive an invariant INV from the postcondition POST and the final bound of the FOR-loop only; the final bound of a FOR-loop is the upper bound for upwards counting loops and is the lower bound for downwards counting loops. Especially, BODY is not used for the derivation of INV, but it is used to check whether INV is really an invariant of the loop. By not using BODY we avoid the problem that a loop with an incorrect BODY may lead to the generation of invalid invariants resulting in redundant work for the APV. Another aspect of only using POST and the final bound is that nested loops have not to be treated as a special case as e.g. in [Weg74], but can be treated in a recursive manner. The method for the generation of hypothetical invariants is formulated as an algorithm which can be used in an APV.

Related work. Soon after the seminal work on program verification by Floyd and Hoare [Flo67; Hoa69] began a phase of intensive work on developing methods for the determination of loop invariants [e.g. Weg74; Cap75; KM76; MW77; Mis78; Bas80; Tam80; Ell81; Gri82; BD84; GDM85; Pai86; CEG99; Kau99; CEG00; BMM01; FQ02]. More recently several methods have been presented to determine especially

[1] This refers primarily to the generation of the VCs. The verification proper may still be rather difficult even for very simple statements: {True} skip; {Goldbach's conjecture}.

polynomial invariants [MS03; MSS04; JK05; PS05; KR06]. Some methods are for application by hand [e.g. Cap75; Mis78], some work in a semi-mechanized manner [e.g. Weg74; Tam80; BMM01; FQ02] and some are fully mechanized [e.g. Kau99; PS05; JK05; KR06].

The different approaches exploit the annotated loop in different ways:

Some methods use the loop only, i.e. derive invariants from the code [e.g. KM76; Bas80; Tam80; Ell81; GDM85; Pai86; MS03; MSS04; JK05; PS05; KR06]. In [CEG99, CEG00] the loop is instrumented in order to output interesting variables ("trace variables"). The method then tries to infer an invariant from the values of the trace variables for several executions of the loop. This is a special case of deriving an invariant from the loop, because the values of the trace variables are determined by the loop.

Another approach is to derive the invariant from the specification [Mis78; Gri82]. Misra [Mis78] mentions two approaches: "A loop invariant could be a proposition about "what has been done" or a proposition about "what remains to be done" ". Gries [Gri81, Gri82] derives an invariant of the kind "what has been done" from POST. Gries attributes this methodology to Dijkstra [Dij76]. Whereas Misra uses the invariant for the verification of an existing loop, Gries uses the invariant for the development of the loop itself.

Wegbreit [Weg74] and Kauer [Kau99] derive INV from POST and the loop-condition. The method of Kauer is inspired by [Gri82], but is mechanized, and is tailored to FOR-loops and to the verification of an existing loop. The details are the topic of this paper.

Most methods work with WHILE-loops. Since FOR-loops can be transformed into WHILE-loops these methods can also be applied to FOR-loops. If e.g. the method of [Weg74] is applied to the WHILE-loop corresponding to example (4) in sect. 3.1 no loop invariant seems to be produced, despite the fact that the candidates are also derived from POST and the loop-condition.

The rest of the paper is organized as follows. In section 2 we present the verification scheme for FOR-loops. Section 3 contains the method for the computation of an invariant and some examples of its application. Section 4 concludes the paper.

2 An Improved Proof Rule for FOR-loops

The proof rule for FOR-loops in [Win98] is based on that in [Hoa 72]. The main differences are that [Win98] does not require $\boldsymbol{I}([\,])$ to hold before the first execution of the loop body. The invariant $\boldsymbol{I}([\mathrm{LO} \,..\, \mathrm{i}])$ must only hold after executions of the loop body. Secondly, the loop variable may occur in the invariant, and thirdly, the proof rule also works for loops with zero repetitions. The strategy for the handling of the invariant is based on the following observations:

(1) the invariant is intended to be an assertion which is established by any execution of the loop body, especially the last one; therefore, it seems not necessary that the invariant holds before the FOR-loop. Collins calls such an invariant a "post-invariant" [Col88];

(2) the invariant of a FOR-loop is typically an inductive assertion which involves the loop variable. In [Hoa72] the loop variable must not occur in the invariant;

(3) in some programming languages the loop variable is declared locally in the loop and does not exist outside the loop [e.g. Algol 68, Ada, C#]. If the invariant contains the loop variable and must hold before the loop, this could lead to illegal uses of the loop variable;

(4) there are examples in which it seems difficult to derive $\boldsymbol{I}([\,])$ mechanically from $\boldsymbol{I}([LO\,..\,i])$. One example for this is [Win 98: 9]:

```
v := 5;
     -- v=5
FOR i IN 1 .. 10
LOOP  v := i;                                   (1a)
     --  inv ???
END LOOP;
     -- v=10
```

It is easy to see that $\boldsymbol{I}([1..i]) \equiv v = i$ is an invariant which fulfills (1a).

$\boldsymbol{I}([1..i])^{i}_{10} \equiv \boldsymbol{I}([1..10]) \equiv v=10$ is sufficient to establish the postcondition. We then have to determine $\boldsymbol{I}([\,])$ such that

$$[\, v=5 \Rightarrow \boldsymbol{I}([\,])\,] \wedge [\, \boldsymbol{I}([\,]) \Rightarrow wp(\text{“v:=1;“}, v=1\,] \qquad (1b)$$

holds. If we try $\boldsymbol{I}([\,]) \equiv \boldsymbol{I}([1..i])^{i}_{pred(1)} \equiv v=0$ we observe that it does not work: because $[v=5 \Rightarrow v=0] \equiv$ False. On the other hand, $\boldsymbol{I}([\,]) \equiv$ true does the trick; it is maximal in that it is the weakest solution of (1b). But it is not derived mechanically from $\boldsymbol{I}([1..i])$.

Apart from these differences, the verification scheme is expressed in a form suitable for automatic verification using ABM, whereas the proof rule in [Hoa72] is formulated as a logical derivation rule.

The verification scheme for FOR-loops used in this paper is:

$$\begin{aligned}
&[\, PRE \Rightarrow LO, UP \in Ti\,] \;\wedge \\
&[\, PRE \wedge LO > UP \Rightarrow POST\,] \;\wedge \\
&[\, PRE \wedge LO \le UP \Rightarrow wp(\, BODY^{i}_{LO}, INV^{i}_{LO}\,)\,] \;\wedge \\
&[\, LO \le i < UP \wedge INV \Rightarrow wp(\, BODY^{i}_{i+1}, INV^{i}_{i+1}\,)\,] \;\wedge \\
&[\, LO \le UP \wedge INV^{i}_{UP} \Rightarrow POST\,]
\end{aligned} \qquad (2)$$

where

PRE	is the precondition
POST	is the postcondition
INV	is the invariant
Ti	is the value set of the type of the loop variable i
[...]	denotes universal quantification over the program variables and the specification variables

This form of the verification scheme assumes that

(r1) the evaluation of LO and UP has no side effects

(r2) any evaluation of LO, UP or any of their subexpressions at any point in the FOR-loop yields the same value as in the initial evaluation at the beginning of the

execution of the FOR-loop. This means especially that LO and UP are not written to in BODY and that they do not contain calls of functions which are not referentially transparent.

Both restrictions hold for many loops used in practice. Restriction (r2) is not severe; [Win98; KW00] contain a scheme which does not require restriction (r2) by introducing two fresh variables vlo and vup which are assigned the values of LO and UP before beginning the repetitions of the loop body. Since the method for the computation of the invariant does not depend on the exact form of the loop verification scheme, we use the simpler form of the verification scheme for the examples in this paper.

In [KW00] we show that (2) implies the correctness of the loop (1) and that the correctness of (1) implies the existence of an invariant INV, which satisfies (2).

3 A Method for Computing Invariants of FOR-Loops

3.1 Basic Idea

The general wp-rule for a FOR-loop cannot always be solved exactly. Usually, some weaker form of correctness is used which uses a loop invariant [Gri81; Win98]. This means that the engineer has to determine a suitable invariant. If such an invariant can be computed mechanically the task of the engineer will be easier. In this section we present a method for the mechanical generation of invariants of FOR-loops, which are annotated by PRE and POST only.

The method is based on the heuristic "*replacing a constant in the postcondition by a variable(RCPV)*" [Gri81: 199], where in our case the variable is always the loop variable. The heuristic RCPV is typically applied by replacing the final bound in POST by the loop variable. For upwards counting loops the final bound is UP, and for downwards counting loops it is LO. In the following we show the derivation of the method for upwards counting loops. How it works for downwards counting loops is presented in [KW00].

The method works in two steps:

(1) try to derive a predicate HI (hypothetical invariant) from the AFL. There are cases in which the method does not generate a predicate HI, e.g. if UP is a non-linear expression. From a practical point of view those cases are rare. A check of [BG91] gave the following result: in most FOR-loops UP has one of the forms: (a) variable, (b) sum of two variables, or (c) sum of a variable and a constant. The Fortran program RWPL (= Randwertproblemlöser = boundary value problem solver), written by M. Hermann and D. Kaiser of our department, contains 1015 FOR-loops (DO-loops in Fortran), of which 998, i.e. almost all, are appropriate for our method.

(2) try to prove $(\text{FOR-rule})^{INV}_{HI}$. There are three possible answers:

a) the proof succeeds, i.e. HI is an invariant and the loop is correct.

b) the refutation succeeds. This can be due to the following reasons:

b1) the loop is correct, but HI is not an invariant.

b2) the loop is not correct. In this case HI may or may not be an invariant.

c) neither proof nor refutation succeed, i.e. the prover "gives up" or does not terminate. In this case it is unknown whether the loop is correct or incorrect, or whether HI is or is not an invariant.

Only in case a) does the method say that the loop is correct.

The idea behind this method is that most FOR-loops compute their final result by computing a sequence of partial intermediate results which approximate the final result better and better. The final result is described by POST and often depends on a characteristic constant or variable which usually is UP. $POST^{UP}_{i}$, which then depends on i, often characterizes these partial results.

A very simple example is a loop for the summation of the first 100 natural numbers:

```
    -- PRE: s = 0  ∧  s ∈ int32
FOR i in 1..100 LOOP s := s+i; END LOOP                    (3)
    -- POST: s = ⟨Σj: 1..100: j⟩  ∧  s ∈ int32
```

In (3) we assume that the type of s is int32. In (3) RCPV can be applied directly and gives the HI

$$HI \equiv POST^{100}_{i} \equiv (s = \langle \Sigma j: 1..100: j \rangle \wedge s \in int32)^{100}_{i}$$
$$\equiv s = \langle \Sigma j: 1..i: j \rangle \wedge s \in int32$$

The loop (3) with the invariant HI satisfies (2) and therefore, HI is an invariant of (3) and (3) is correct. Since $\langle \forall i \in 0..100: \langle \Sigma j: 1..i: j \rangle \in int32 \rangle$ holds, we could have ommitted `s ∈ int32` in (3). We included it for documentation purposes.

(3) is a very special loop because the upper bound is the fixed number 100. Often the upper bound will be a program variable whose value is constant in the FOR-loop. Such a more general FOR-loop is given in (4).

```
    -- PRE: s=0 ∧ 0≤n≤65535 ∧ n=N
FOR i in 1..n LOOP s := s+i; END LOOP                      (4)
    -- POST: s=⟨Σj: 1..n: j⟩ ∧ 0≤n≤65535 ∧ n=N
```

We assume that s and n are of type int32. N is a specification variable which is used to guarantee that the value of n after the loop is the same as before the loop. If we compute HI mechanically as $POST^{n}_{i}$ we obtain

$$HI \equiv POST^{n}_{i} \equiv (s=\langle \Sigma j: 1..n: j \rangle \wedge 0 \le n \le 65535 \wedge n=N)^{n}_{i}$$
$$\equiv s=\langle \Sigma j: 1..i: j \rangle \wedge 0 \le i \le 65535 \wedge i=N$$

We observe immediately that HI is not an invariant of (4) because i has not always the value N (if N > 1).

A strategy for avoiding this problem is to apply the substitution $(n \mapsto i)$ only to those conjuncts of POST which are not also a conjunct of PRE. In the example this results in

$$HI \equiv (s=\langle \Sigma j: 1..n: j \rangle)^{n}_{i} \wedge 0 \le n \le 65535 \wedge n=N$$
$$\equiv s=\langle \Sigma j: 1..i: j \rangle \wedge 0 \le n \le 65535 \wedge n=N$$

The loop (4) with HI as invariant satisfies (2).

A method for the identification of common conjuncts is given in sect. 3.3.

3.2 Bound Transformation

The method developed so far works only if UP is a constant or a variable. This is a severe restriction. One idea is to insert the assignment "vup := UP;" immediately before the loop, where vup is a fresh variable, and then use vup as the upper bound. But this does not work in general, since vup does not occur in POST. Replacing a nonoccurring variable does not change POST, so that POST itself had to be considered as an invariant, which does not work in most cases.

In the loop (5) UP is not a variable but a more complicated expression.

```
   -- PRE: s = 0
FOR i in 1..n+m LOOP s := s+i; END LOOP                    (5)
   -- POST: s = ⟨Σj: 1..n+m: j⟩
```

In (5) we have omitted the type constraints on s, n and m because they are not significant for the current discussion. In general, the value of n+m is constrained by the type of s: in (4) the constraint for n guarantees that $s \in int32$.

The introduction of vup and the application of RCPV results in

```
vup := n+m;
   -- PRE: s = 0 ∧ vup = n+m
FOR i in 1..vup LOOP s := s+i; END LOOP                    (6)
   -- POST: s = ⟨Σj: 1..n+m: j⟩
```

Since vup does not occur in POST we obtain

$$HI \equiv POST^{vup}_{i} \equiv s = \langle \Sigma\, j\colon 1..n+m\colon j\rangle^{vup}_{i} \equiv s = \langle \Sigma j\colon 1..n+m\colon j\rangle \equiv POST.$$

It is easy to see that HI is not an invariant.

It seems therefore better to try to transform the given loop L1 into an equivalent loop L2 with upper bound UP2, such that UP2 = v and $v \in free(UP1) \cap free(POST)$ holds.

In the following we distinguish between expressions such as LO and UP and their value, which we denote by s1(LO) and s1(UP) where s1 is the state just before the first execution of BODY.

In this paper we use transformations t which induce a translation of the range s1(LO1) .. s1(UP1) by a constant k = s1(e), where e is some arithmetic expression. The resulting range is then s1(LO1)+s1(e) .. s1(UP1)+s1(e) = s1(LO2) .. s1(UP2).

This means that the number of executions of BODY is the same in the transformed loop L2. If we use a suitable transformation t* in BODY, which compensates for the translation t of the values of the loop variable, we obtain a loop which is semantically equivalent to the given loop L1. This leads to the following scheme (for upward counting loops):

L1:
```
   -- PRE
FOR i in LO1..UP1 LOOP BODY END LOOP
   -- POST
```

L2:
```
   -- PRE
FOR i in t(LO1)..r(t(UP1))
LOOP BODY(i ↦ t*(i)) END LOOP
   -- POST
```

r(•) is a function which reduces (simplifies) arithmetic expressions.

In L2 a translation t(•) is applied to LO and UP and a second translation t*(•) to all occurrences of the loop variable i in BODY. The idea is that t*(•) neutralizes t(•) and that therefore the BODY of L2 is executed for the same values of the loop variable i as the BODY of L1. That this is really the case is shown below.

We apply the loop transformations only in such cases in which r(t(UP1)) has the form "v" or "–v". Since UP2 = r(t(UP1)) has this simple form we get the hypothetical invariant

HI = $POST^{v}_{i}$, if r(t(UP1)) has the form "v ", or

HI = $POST^{v}_{(-i)}$, if r(t(UP1)) has the form "-v" .

L2 is never really executed but is only used to determine HI and a corresponding proof rule. On the level of proof rule and proof we assume the usual mathematical sets of numbers, i.e. when applying t and t* we do not have to watch for range violations and can apply the usual laws of arithmetic.

The transformation t depends on UP and v and is a mapping $E \times Var \rightarrow (E \rightarrow E)$, i.e. $t(UP,v) \in E \rightarrow E$, where E is the set of arithmetic expressions. When UP and v are known from the context we also write t(e) instead of t(UP,v)(e). t(UP,v) is applied to both LO1 and UP1. In order to find t for a given expression UP1 and a given variable v we determine the syntactic transformation necessary to semantically neutralize all terms apart from v or -v.

E.g. if UP = m+10 we obtain t(UP, m)(e) = e -10 and for UP = n*a +b we obtain t(UP, n)(e) = (e-b)/a and t(UP, b)(e) = e - n*a . We obtain the corresponding t* by modifying t(e) analogously wrt e.

Not all such transformations lead to a translation of s1(LO1) .. s1(UP1). For UP1 = 2*n we obtain t(UP1,n)(e) = e/2 . If LO1 = 0 then the original range is 0..2*n and the transformed range is 0..n whose lengths are different for n >0. On the other hand, if we rewrite 2*n as n+n, we could transform the range 0..2*n into -n..n which has the same length.

This means that the transformation of the range is only possible if UP1 has a suitable form. In this paper we limit the possible transformations to the cases in the following table:

Table 1. Definition of the mappings t and t*

	Form of UP			
	o1 v	e1 o2 v	o1 v o2 e1	e1 o2 v o3 e2
t(UP,v)(e)	e	e - e1	e $o2^{-1}$ e1	e - e1 $o3^{-1}$ e2
r(t(UP,v)(UP))	o1 v	o2 v	o1 v	o2 v
t*(UP,v)(e)	e	e + e1	e o2 e1	e + e1 o3 e2
t*(UP,v)(t(UP,v)(e))	e	e - e1 + e1	e $o2^{-1}$ e1 o2 e1	e - e1 $o3^{-1}$ e2 + e1 o3 e2

where $v \notin free(e1) \cup free(e2)$, $o1 \in \{+, -, \varepsilon\}$, $o2, o3 \in \{+, -\}$, $+^{-1} = -$, $-^{-1} = +$, and e1 and e2 are parenthesized expressions. If e.g. UP = v-a+b we assume that UP has been transformed into v-(a-b) or an equivalent parenthesized form. As already mentioned in sect. 3.1 these restrictions seem not severe from a practical point of view.

Since the expressions in table 1 operate over the mathematical sets of numbers the usual algebraic rules apply. It is easy to see that the last row implies that $t^*(UP,v)(t(UP,v)(e))$ is semantically equivalent to e.

In order to show the equivalence of L1 and L2 we need one last property: $t(UP,v)(\bullet)$ and $t^*(UP,v)(\bullet)$ must be constant throughout the FOR-loop. This is guaranteed by the restriction (r2) in sect. 2. A consequence of this is: there is a $k \in \mathbb{Z}$ such that $s(t(UP,v)(e)) = s(e) + k$ for any arithmetic expression e and for any state s during the execution of the loop, where k can be derived from table 1. Analogously, we have $s(t^*(UP,v)(e)) = s(e) - k$.

With these properties we can now show that in L1 and in L2 BODY is executed for the same sequence of values of the iteration expression. For L1 we obtain $BODY^{s1(LO1)..s1(UP1)}$. For L2 we obtain

$$
\begin{aligned}
& \{BODY^{i}_{t^*(i)}\}^{s1(t(LO1))..s1(t(UP1))} \\
= \; & BODY^{s1(t^*(s1(t(LO1)))) \,..\, s1(t^*(s1(t(UP1))))} && \text{-- } s(t(e)) = s(e) + k \\
= \; & BODY^{s1(t^*(s1(LO1)+k)) \,..\, s1(t^*(s1(UP1)+k))} && \text{-- } s(t^*(e)) = s(e) - k \\
= \; & BODY^{s1(s1(LO1)+k)-k \,..\, s1(s1(UP1)+k)-k} && \text{-- } s(a+b) = s(a) + s(b),\ s(s(e)) = s(e) \\
= \; & BODY^{s1(LO1) \,..\, s1(UP1)}
\end{aligned}
$$

3.3 Determination of Common Conjuncts

According to the observation in (4) we present a refinement of the basic strategy by exempting common invariant conjuncts from RCPV. Common conjuncts often occur in programs with nested loops. One example is example 17 in [FKW02; KW99b]. A second example is the algorithm (7), which computes the ∞-norm p of the matrix a of size m×n, which is defined as: $p = \langle \text{Max } k: 1 \le k \le m: \langle \Sigma\ c: 1 \le c \le n: |a(k,c)| \rangle \rangle$ [GL89].

```
   -- PREo: {m,n} ≥ 1 ∧ p = 0
FOR i IN 1..m LOOP
   s := 0;
   -- PREi: s = 0 ∧
      --      p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   FOR j IN 1..n LOOP
      s := s+abs(a(i,j));                                          (7)
    END LOOP;
      -- POSTi: s = ⟨Σ c: 1..n: |a(i,c)|⟩ ∧
      --        p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   IF s>p THEN p := s; END IF;
      -- p = ⟨Max k: 1..i: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
END LOOP;
   -- POSTo: p = ⟨Max k: 1..m: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
```

PREi and POSTi have one conjunct in common, in which the upper bound must not be replaced by the loop variable to obtain an HI. This HI is an invariant of the inner loop.

We determine common conjuncts as follows

(a) transform PRE and POST into normal form NF

(b) determine the syntactically common conjuncts $C = C1 \wedge \ldots \wedge Cn$

(c) determine those Ci for which
$noWrite(BODY, free(Ci)) \vee [Ci \Rightarrow wp(BODY, Ci)]$ holds;
noWrite(S, M) means that no variable in the set of variables M is written to in the statement S.
Let C_{com} be the conjunction of these Ci.

The condition in (c) seems to be unnecessarily complex because, from a theoretical point of view, $trans(BODY, Ci) \equiv [Ci \Rightarrow wp(BODY, Ci)]$ is necessary and sufficient. From a practical point of view we have to bear in mind that a theorem prover may not be able to prove a theorem. On the other hand, noWrite(BODY, free(Ci)) can often be checked more easily, especially in the absence of function calls. Furthermore it is sufficient but not necessary. Therefore, using the combination of both conditions may classify more Ci as an invariant versus using either one alone.

The details of the normal form are given in [KW00].

If the method finds any invariant common conjuncts the normalized POST can be written as $POST' \wedge C_{com}$, where POST' does not contain any conjunct of C_{com}. The hypothetical invariant is then

$$HI \equiv POST'^{r(t(UP))}_{i} \wedge C_{com} \quad \text{or}$$
$$HI \equiv POST'^{r(-t(UP))}_{(-i)} \wedge C_{com}.$$

3.4 Adaptation of the Proof Rule

The bound transformation and the common conjuncts must now be considered in the proof rule for the FOR-loop. There are four factors which influence the adaptation of the proof rule:

a) direction of the loop: upwards / downwards

b) bound modification in BODY: bounds are modified / bounds are not modified

c) form of the transformed final bound: v / -v

d) occurrence of the loop variable in POST: $i \in free(POST)$ / $i \notin free(POST)$

The proof rule for the case

(upwards, not modified, v, $i \notin free(POST)$)

is given in (8).

$$\begin{aligned}
&[\, PRE \Rightarrow LO, UP \in Ti \,] \;\wedge \\
&[\, PRE \wedge LO > UP \Rightarrow POST' \,] \;\wedge \\
&[\, PRE \wedge LO \le UP \Rightarrow wp(\, BODY^{i}_{LO}, POST'^{r(t(UP))}_{t(LO)} \,) \,] \;\wedge \\
&[\, t(LO) \le i < t(UP) \wedge POST'^{r(t(UP))}_{i} \wedge C_{com} \Rightarrow wp(\, BODY^{i}_{t^*(i)+1}, POST'^{r(t(UP))}_{i+1} \,) \,]
\end{aligned} \tag{8}$$

The proof rules for the other cases are given in [KW00].

3.5 Algorithm for the Application of the Method

We are now ready to put the pieces together and present the application of the method as an algorithm, which works for both upwards and for downwards counting AFLs.

```
-- INPUT: PRE, POST, i, LO, UP, BODY, UpwardsCounting?
AFLCorrect?: enum(proof, open) := open;
FinalBound: expression;

IF UpwardsCounting?
THEN FinalBound := UP; ELSE FinalBound := LO; END IF;

IF FinalBound is suitable (see table 1)
THEN  C_com: expression := true;
      HI: expression;
      Post': expression := POST;

      IF there is a common conjunct c with
         noWrite(BODY, free(c)) ∨ [c ⇒ wp(BODY,c)]
      THEN C_com := ⟨∧ c: c is common conjunct:
                      noWrite(BODY, free(c)) ∨
                      [c ⇒ wp(BODY, c)]⟩;
           POST' := con(set(POST) - set(C_com));
      END IF;

   -- create the set T of all possible translations
   -- t(FinalBound,v),
   -- where v ∈ free(FinalBound) ∩ free(POST);
   FOR EACH t ∈ T DO
      -- r(t(FinalBound)) = v  ∨ r(t(FinalBound))= -v
      IF r(t(FinalBound))= v
      THEN POST' := POST'^v_i;
      ELSE POST' := POST'^v_(-i);
      END IF;
      IF the AFL can be proved using POST' and C_com in the
         appropriate rule in sect. 3.4
      THEN AFLCorrect? := proof; EXIT;
      END IF;
   END FOR;
END IF;

-- OUTPUT: AFLCorrect?
```

The functions con(·) and set(·) are defined as follows:

$$\text{set}(C_1 \wedge \ldots \wedge C_n) = \{ C_1, \ldots, C_n \},$$

$$\text{con}(\{ C_1, \ldots, C_n \}) = C_1 \wedge \ldots \wedge C_n$$

The meaning of the three possible outcomes (proof, open, nontermination) has already been explained in section 3.1.

3.6 Examples

In the following example (9) the natural numbers in the range m .. m-n (for n≤0) are summed up.

```
   -- PRE: s = 0 ∧ m ≥ 0 ∧ n≤0
FOR i IN m .. m-n LOOP
   s := s + i;                                                    (9)
END LOOP;
   -- POST: s = ⟨Σ j: m..m-n: j⟩
```

UP is suitable, $C_{com} \equiv$ true, free(UP) ∩ free(POST) = {m, n},
2 transformations are possible:
t1(m-n, m)(e) = e+n and t2(m-n, n)(e) = e-m.
t1 does not yield an invariant, but t2 gives

HI ≡ s = ⟨Σ j: m..m+i: j⟩

which is an invariant of the transformed loop. The transformed loop and HI satisfy (8).

The second example is from [PS05] and computes the sum of squares of the first n natural numbers. An equivalent AFL is (10).

```
   -- PRE: n≥0 ∧ n≤1860 ∧ n=N ∧ x=0
FOR y IN 0..n LOOP
   x := y*y + x;                                                  (10)
END LOOP;
   -- POST: x=(2n^3+3n^2+n)/6 ∧ x∈int32 ∧ n≥0 ∧ n≤1860 ∧ n=N
```

Additionally to [PS05] we assume that x ∈ int32 and use n in POST instead of y, which may not be in scope. Since UP is a simple variable we obtain directly

HI ≡ $x = (2y^3 + 3y^2 + y)/6 \wedge x \in \text{int32} \wedge n \geq 0 \wedge n \leq 1860 \wedge n = N$

HI is an invariant and the loop (10) together with HI satisfies (8).

4 Conclusion

We have developed a method for the mechanical generation of invariants for a practically relevant class of FOR-loops. The method can be incorporated into automatic program provers and would lead to a simplification of program verification using such a tool. By extending the suitable forms of the final bound the applicability of the method could be extended to further classes of FOR-loops.

Acknowledgments. We are grateful for the very useful hints of an anonymous referee which led to a number of improvements of the paper.

References

ABB05 Ahrendt, Wolfgang; Baar, Thomas; Beckert, Bernhard; et al.: The KeY tool. Software Syst Model 4 (2005) 32..54. DOI 10.1007/s10270-004-0058-x

Bar00 Barnes, John: High Integrity Ada - The SPARK Approach -. Addison-Wesley, 2000.

Bas80 Basu, S. K.: A Note on Synthesis of Inductive Assertions. IEEE TSE 6, 1 (1980) 32..39

BCD06 Barnett, M; Chang, B-Y E.; DeLine, R. et al.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Springer, LNCS 4111, Berlin, 2006, pp. 364..387

BD84 Dunlop, D. D.; Basili, V. R.: A Heuristic for Deriving Loop Functions. IEEE TSE 10, 3 (1984) 275..285

BG91 Gonnet, G. H.; Baeza-Yates, R.: Handbook of Algorithms and Data Structures. Addison Wesley, Wokingham, 1991. ISBN-10: 0-201-41607-7

BMM01 Ball, T.; Majumdar, R.; Millstein, T.; Rajamani, S. K.: Automatic Predicate Abstraction of C Programs. ACM PLDI 2001, 203..213

Cap75 Caplain, Michel: Finding Invariant Assertions for Proving Programs. SIGPLAN Notices 10, 6 (1975) 165..171

CEG99 Ernst, M. D.; Cockrell, J.; Griswold, W. G.; Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. ICSE '99, 213..224

CEG00 Ernst, M. D.; Czeisler, A.; Griswold, W. G.; Notkin, D.: Quickly Detecting Relevant Program Invariants. ICSE 2000, 449..458

Col88 Collins, W. J.: The Trouble with FOR-Loop Invariants. ACM SIGCSE Bull. 20, 1 (1988) 1..4

Dij76 Dijkstra, Edsger W.: A Discipline of Programming. Prentice-Hall, Inc., 1976.

Ell81 Ellozy, H. A.: The Determination of Loop Invariants for Programs with Arrays. IEEE TSE 7, 2 (1981) 197..206

FKW02 Ein Vergleich der Programmbeweiser FPP, NPPV und SPARK. Ada-Deutschland Tagung 2002. Shaker Verlag, Aachen, 2002. p. 127..145. ISBN-10: 3-8265-9956-X

Flo67 Floyd, R. W.: Assigning Meaning to Programs. In: Schwartz, J. T. (ed.): Mathematical Aspects of Computer Science. AMS, 1967, pp. 19 .. 32. ISBN-10: 0-8218-1319-6

FQ02 Flanagan, C.; Qadeer, S: Predicate Abstraction for Software Verification. ACM POPL'02, 191..202

GDM85 Mili, A.; Desharnais, J.; Gagné, J.-R.: Strongest Invariant Functions: Their Use in the Systematic Analysis of While-Statements. Acta Informatica 22 (1985) 47..66

GL89 Golub, G.H.; Loan, C.F. van: Matrix Computations. John Hopkins Press, 1989

Gri81 Gries, D.: The Science of Programming. Springer, New York, 1981

Gri82 Gries, D.: A Note on a Standard Strategy for Developing Loop Invariants and Loops. Sci Comp Progr 2 (1982) 2007..214

Gumxx Gumm, H: New Paltz Program Verifier. http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html. Visited 2007.Feb.21

Hoa69 Hoare, C. A. R.: An Axiomatic Basis of Computer Programming. CACM 12, 10 (1969) 576..580, 583

Hoa72 Hoare, C. A. R.: A Note on the FOR Statement. BIT 12 (1972) 334..341

JK05 Kovács, L. I.; Jebelean, T.: An Algorithm for Automated Generation of Invariants for Loops with Conditionals. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2005

JKP03 Kovács, L. I.; Popov, N.; Jebelean, T.: Verification of Imperative Programs in Theorema. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2003

Kau99 Kauer, S.: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich Schiller University, 1999.Jan.27

KM76 Katz, S.; Manna, Z.: Logical Analysis of Programs. CACM 19, 4 (1976) 188..206

KR06 Rodríguez-Carbonell, E.; Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci Comp Progr 64 (2007) 54..75. Available online 28 Sept 2006 at http://www.sciencedirect.com/; visited 2007.Jan.22

KW99a Kauer, S.; Winkler, J. F. H.: FPP: An Automatic Program Prover for Ada Statements. Workshop "Objektorientierung und sichere Software mit Ada". Karlsruhe, 1999.Apr.21-22

KW99b Kauer, S.; Winkler, J. F. H.: A Comparison of the Program Provers NPPV and FPP. Report Math / Inf / 1999 / 28, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 1999

KW00 Kauer, S.; Winkler, J.F.H.: Automatic Generation of Invariants for FOR-Loops Based on an Improved Proof Rule. Report Math / Inf / 2000 / 26, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 2000

Mis78 Misra, J.: Some Aspects of the Verification of Loop Computations. IEEE TSE 4, 6 (1978) 478..486

MS03 Müller-Olm, M.; Seidl, H.: Computing Polynomial Program Invariants. October 2, 2003. 2007Feb11 from: http://www.informatik.fernuni-hagen.de/forschung/ informatikberichte/pdf-versionen/310.pdf

MSS04 Sankaranarayanan, S.; Sipma, H. B.; Manna, Z.: Non-Linear Loop Invariant Generation using Gröbner Bases. ACM POPL 2004, 318..329

MW77 Morris, J. H. Jr.; Wegbreit, B.: Subgoal Induction. CACM 20, 4 (1977) 209..222

Pai86 Paige, R.: Programming with Invariants. IEEE Software 3, 1 (1986) 56..69

PS05 Seidl, H.; Petter, M.: Inferring Polynomial Invariants with Polyinvar. Technische Universität München, Garching, Germany. 2007.Feb.12 from: http://www2.cs.tum.edu/ ~petter/papers/nsad05.pdf

Tam80 Tamir, M.: ADI: Automatic Derivation of Invariants. IEEE TSE 6, 1 (1980) 40..48

Tur49 Turing, A.: Checking a Large Routine. In: Williams, L. R.; Campbell-Kelly, M. (eds.): The Early British Computer Conferences. MIT Press, Cambridge, 1989, 70..72

Weg74 Wegbreit, Ben: The Synthesis of Loop Predicates. CACM 17,2 (1974) 102..112

Win97 Winkler, J.F.H.: The Frege Program Prover. 42. Int. Wiss. Koll., Ilmenau, 1997. Vol.1 116..121

Win98 Winkler, J.F.H.: New Proof Rules for FOR-loops. Report Math / Inf / 98 / 13, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 1998

An Invariant-Based Approach to the Verification of Asynchronous Parameterized Networks

Igor V. Konnov and Vladimir A. Zakharov

Faculty of Computational Mathematics and Cybernetics,
Lomonosov State University, Moscow, RU-119899, Russia
konnov@cs.msu.su, zakh@cs.msu.su

Abstract. A uniform *verification problem for parameterized systems* is to determine whether a temporal property is true for every instance of the system which is composed of an arbitrary number of homogeneous processes. To cope with this problem we combine an induction-based technique for invariant generation and conventional model checking of finite state systems. At the first stage of verification we try to select automatically an appropriate invariant process which is greater (with respect to some preorder relation) than any instance of the parameterized system. At the second stage, as soon as an invariant of the parameterized system is obtained, we verify it by means of conventional model checking tool for temporal logics. To demonstrate the feasibility of quasi-block simulation we implemented this technique and applied it to verification of Resource ReSerVation Protocol (RSVP).
Keywords: program verification, asynchronous networks, invariant generation, induction, simulation, model checking.

1 Introduction

Verification plays an important role in designing reliable computer systems. Two main approaches to program verification are testing and formal verification. Since the behavior of concurrent systems is usually very complicated and tends to be non-reproducible, many bugs are difficult to detect by conventional testing. Formal verification approach provides a more preferred solution. It assumes that one builds a mathematical model for the system to be analyzed, specifies the properties the system should comply with, and then applies some appropriate mathematical techniques to check that the model satisfies the properties. Formal verification is relatively inexpensive in comparison to exhaustive simulation; it receives an ample algorithmic support from various branches of mathematics and manifests its strength in areas where other verification methods are inadequate.

Formal methods fall into the following major categories: model checking, theorem proving, and equivalence checking. Model checking allows verification of computer system by checking that a model $M(P)$ (usually represented as transition system derived from hardware or software design P) satisfies a formal specification φ (usually represented as temporal logic formula). When $M(P)$ is a finite state model then one could find a rich variety of model checking

procedures (see [9]). In what follows we will assume that each system (process) P under consideration has only finite state and will not distinguish it from its model (transition system) $M(P)$.

The development of effective techniques for checking parameterized systems $\mathcal{F} = \{P_k\}_{k=1}^{\infty}$ is one of the most challenging problems in verification today. The parameter k may stand for any variable characteristics of the design P_k (say, the size of some data structures, stacks, etc.), but much attention is given to the cases when the concurrent systems P_k are the parallel compositions $p_1 \| p_2 \| \ldots \| p_k \| q$ of similar "user" processes $p_1, p_2, \ldots, p_k$ and a control process ("environment") q. Then the uniform verification problem for parameterized systems is formulated as follows: given an infinite family $\mathcal{F}$ of systems $P_k = p_1 \| p_2 \| \ldots \| p_k \| q$ and a temporal formula φ, check that each transition system $M(P_k)$ satisfies φ.

Though in [1] it was shown that the problem is undecidable in general, some positive results may be obtained for specific parameterized systems. For the most part three basic techniques, namely, *symmetry*, *abstraction*, and *induction* are employed to extend the applicability of conventional model checking to restricted families of parameterized systems.

The idea of exploiting symmetry for state set reduction was introduced in [6, 15, 16]. Symmetry-based reduction has been successfully applied to a number of case studies (see [4] for survey) and now it is implemented within a framework of many model-checkers [2, 19]. However, in many practical cases this approach run into obstacles, since the problem of finding orbit representatives is as hard as graph isomorphism problem. Some papers [14, 24] have demonstrated a considerable progress in automatic symmetry detection, but this problem still remains the main critical point of the symmetry-based reduction techniques.

Abstraction is likely to be the most important technique for coping with state explosion problem in model checking. A theoretical framework for abstraction technique has been developed in [5, 13, 20]. Abstraction has been widely applied in verification of parameterized systems. Only with the essential help of abstraction does it become possible to apply model checking to verify infinite state systems (see [7, 23]). But, unfortunately, most of the abstraction techniques require user assistance in providing key elements and mappings.

The common idea of the induction technique can be summarized as follows. Define some preorder $\preceq$ (a simulation or bisimulation) on transition systems and choose some class of temporal formulae $Form$ such that

1. the composition operator $\|$ is monotonic w.r.t. $\preceq$, i.e. $P_1 \preceq P_1'$ and $P_2 \preceq P_2'$ imply $P_1 \| P_2 \preceq P_1' \| P_2'$;
2. the preorder $\preceq$ preserves the satisfiability of formulae φ from $Form$, i.e. $P' \models \varphi$ and $M \preceq P'$ imply $M \models \varphi$.

Then, given an infinite family $\mathcal{F} = \{P_k\}_{k=1}^{\infty}$, where $P_k = p_1 \| p_2 \| \ldots \| p_k \| q$, find a finite transition system $\mathcal{I}$ such that

3. $P_n \preceq \mathcal{I}$ for some n, $n \geq 1$;
4. $p_i \| \mathcal{I} \preceq \mathcal{I}$.

A transition system $\mathcal{I}$ which meets the requirements 3 and 4 is called an *invariant* of the infinite family $\mathcal{F}$. Requirements 1, 3 and 4 guarantee that $P_k \preceq \mathcal{I}$ holds for every k, $k \geq n$. If a property is expressed by a formula φ from $Form$ then, in view of the requirement 2, to verify this property of the parameterized system $\mathcal{F}$ it is sufficient to model check $\mathcal{I}$ and P_k, $1 \leq k < n$, against φ. The latter may be done by means of any suitable model-checking techniques for finite state transition systems. This approach to the verification of parameterized networks was introduced in [22, 31] and developed in many papers (see [4] for a survey).

The central problem with induction technique is that of deriving a general method for constructing invariants automatically. In many cases invariants can be obtained by application of the following heuristics: if $P_{k+1} \preceq P_k$ holds for some k then P_k may be used as an invariant $\mathcal{I}$. This consideration captures formally the common belief that some "typical" instance of a parameterized system inherits the most essential features of the whole family. This idea was applied in [7, 8, 15] for developing fully automated approach for verifying parameterized networks.

The right choice of a preorder $\preceq$ is of prime importance for the successful application of this heuristic in practice. In [7] and [15] strong simulation and block bisimulation respectively were used as a preorder $\preceq$, but so far as we know no systematic study of other possible preoreders has been made (though bisimulation equivalences were studied in detail). It is clear that the weaker is preorder $\preceq$, the larger in number are the cases of parameterized systems for which this approach succeeds. In [21] we introduced a block simulation preorder (which is an amalgamation of block bisimulation [15] and visible simulation [18]) and applied this preorder to generate straightforwardly invariants of some parameterized systems. In this paper we continue this line of research. Since asynchronous composition of processes is not monotonic w.r.t. block simulation in general case, we extend this preorder and introduce a quasi-block simulation which is weaker than block simulation. We show that quasi-block simulation preserves the satisfiability of formulae from $ACTL^*_{-X}$ and that asynchronous composition of processes is monotonic w.r.t. quasi-block simulation. This suggests the use of quasi-block simulation for generating invariants in the induction-based verification techniques. We implemented this technique and applied it to verification of Resource ReSerVation Protocol (RSVP). Hitherto formal verification of RSVP has been studied in [12] with the help of CSP and in [30] in the framework of Coloured Petri Nets. We demonstrate that induction-based verification of RSVP can be performed by employing quasi-block simulation.

The paper is organized as follows. In Section 2 we define the basic notions, including asynchronous composition of labelled transition systems, block and quasi-block simulations on transition systems. In Section 3 we study some essential features of quasi-block simulation. The most important property of quasi-block simulation is its monotonicity w.r.t. parallel composition of labelled transition systems. In Section 4 we consider RSVP as a case study and demonstrate how to verify this protocol using induction technique based on quasi-block simulation for generating a suitable invariant. Section 5 concludes with some directions for future research.

2 Definitions

Definition 1. Labelled Transition System (LTS) *is a sextuple* $M = \langle S, S_0, A, R, \Sigma, L\rangle$ *where*

- S *is a finite set of states,*
- $S_0 \subseteq S$ *is the set of initial states,*
- A *is a set of visible actions (not including the invisible action* τ*),*
- $R \subseteq S \times A \cup \{\tau\} \times S$ *is a labelled transition relation,*
- Σ *is a nonempty set of atomic propositions,*
- $L : S \to 2^{\Sigma}$ *is an evaluation function on the set of states.*

Any triple (s, a, t) from R is called a *transition*. We we will write $s \xrightarrow{a}_M t$ instead of $(s, a, t) \in R$ and often elide the subscript M when it is assumed. A *path* π of LTS M is a finite or infinite sequence $\pi = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{j-1}} s_j \xrightarrow{a_j} \ldots$ of transitions $(s_i \xrightarrow{a_i} s_{i+1})$.

Temporal specifications (or properties) of parameterized systems are expressed in temporal logics. The logics used in the framework of induction-based verification technique are usually the Full Branching Time Logic CTL^* or its sub-logics $ACTL^*$ and $ACTL^*_{-X}$. An important factor in deciding between them is a capability of a preorder $\preceq$ used in verification procedure to preserve the satisfiability of temporal formulae. We do not define the syntax and the semantics of these logics; they may be found in many textbooks, e.g. in [9].

Let M_1 and M_2 be two LTSs, $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i\rangle$, $i = 1, 2$, such that $\Sigma^1 \cap \Sigma^2 = \emptyset$. We call a *synchronizer* any pair $\Gamma = \langle \Delta, \overline{\ }\rangle$, where $\Delta \subseteq A^1$, and $\overline{\ } : \Delta \to A^2$ is an injection which binds some actions of M_1 and M_2. We write $\overline{\Delta}$ for the set $\{b \in A^2 \mid \exists a \in \Delta \ : \ \overline{a} = b\}$. When introducing a synchronizer we assume that some actions a are executed only synchronously with the co-actions $\overline{a}$. Thus, a pair $(a, \overline{a})$ forms a channel for communication between M_1 and M_2. One of this action (say, a) may be thought as an action of sending a message, whereas the other (co-action $\overline{a}$) is an action of receiving a message.

Definition 2. *The* (asynchronous) parallel composition *of LTS's* M_1 *and* M_2 *w.r.t. synchronizer* Γ *is an LTS* $M = M_1 \parallel_\Gamma M_2 = \langle S, S_0, A, R, \Sigma, L\rangle$ *such that*

- $S = S^1 \times S^2$, $S_0 = S^1_0 \times S^2_0$, $A = A^1 \cup A^2 \setminus (\Delta \cup \overline{\Delta})$, $\Sigma = \Sigma^1 \cup \Sigma^2$, $L(s, u) = L^1(s) \cup L^2(u)$
- *For every pair of states* $(s, u), (t, v) \in S$ *and an action* $a \in A$ *a transition* $((s, u), a, (t, v))$ *is in* R *iff one of the following requirements is met:*
  - $a \in A^1 \setminus \Delta$, $u = v$, $(s, a, t) \in R^1$ *(M_1 executes a),*
  - $a \in A^2 \setminus \overline{\Delta}$, $s = t$, $(u, a, v) \in R^2$ *(M_2 executes a),*
  - $a = \tau$, *and there exists* $b \in \Delta$ *such that* $(s, b, t) \in R^1$ *and* $(u, \overline{b}, v) \in R^2$ *(M_1 and M_2 communicate),*

Let φ be a temporal formula. Denote by Σ_φ the set of all basic propositions involved in φ. Given an LTS $M = \langle S, S_0, A, R, \Sigma, L\rangle$, one may separate those

transitions of M that either are visible (i.e. marked with an action $a \neq \tau$) or affect the basic propositions of φ:

$$Observ(M, \Sigma_\varphi) = \{(s,a,t) | (s,a,t) \in R \text{ and } (a \neq \tau \vee L(s) \cap \Sigma_\varphi \neq L(t) \cap \Sigma_\varphi)\}.$$

On the other hand, one may also distinguish some set of transitions that seemed "significant" for an observer. Any set $E \subseteq R$ of transitions which includes all visible transitions will be called a set of *events* of M. If $Observ(M, \Sigma_\varphi) \subseteq E$ then the set of events E will be called *well-formed* w.r.t. φ.

Definition 3. *A* finite block *from a state s_1 w.r.t. a set of events E is a finite path $B = s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{a} s_{m+1}$ such that $(s_m, a, s_{m+1}) \in E$ and $(s_i, \tau, s_{i+1}) \notin E$ for all $i : 1 \leq i < m$. An* infinite block *from a state s_1 is an infinite sequence $B = s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_k \xrightarrow{\tau} \cdots$ such that $(s_i, \tau, s_{i+1}) \notin E$ for all $i \geq 1$.*

We write $MAXF(E, s)$ and $MAXI(E, s)$ for the set of all finite and infinite blocks, respectively, from a state s w.r.t. a set of events E.

Definition 4. *Let $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i \rangle$, $i = 1, 2$, be two LTSs, let Σ_0 be a subset of $\Sigma^1 \cap \Sigma^2$, and let E^1 and E^2 be some sets of events of M_1 and M_2. Then a binary relation $H \subseteq S^1 \times S^2$ is called a* quasi-block simulation (qb-simulation) *on M_1 and M_2 w.r.t. Σ_0, E^1, E^2, iff for every pair $(s_1, t_1) \in H$ meets the following requirements:*

1. $L^1(s_1) \cap \Sigma_0 = L^2(t_1) \cap \Sigma_0$,
2. *For every finite block $B' = s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{a} s_{m+1} \in MAXF(E^1, s_1)$ there is a block $B'' = t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_n \xrightarrow{a} t_{n+1} \in MAXF(E^2, t_1)$ such that $(s_{m+1}, t_{n+1}) \in H$, and $(s_i, t_j) \in H$ holds for every pair i, j, $1 \leq i \leq m$, $1 \leq j \leq n$.*
3. *For every infinite block $B' = s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{\tau} \cdots \in MAXI(E^1, s_1)$ there is an infinite block $B'' = t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_n \xrightarrow{\tau} \cdots \in MAXI(E^2, t_1)$, such that $(s_i, t_j) \in H$ holds for every pair i, j, $1 \leq i$, $1 \leq j$.*

We write $M_1 \preceq^{qb}_{\Sigma_0} M_2$ iff there exist two sets of events E^1 and E^2 of LTSs M_1 and M_2 and a binary relation $H \subseteq S^1 \times S^2$ such that H is a qb-simulation on M_1 and M_2 w.r.t. Σ_0, E^1, E^2, and for every initial state $s_0 \in S^1_0$ there exists an initial state $t_0 \in S^2_0$ such that $(s_0, t_0) \in H$. If both E^1 and E^2 are well-formed w.r.t. φ then we say that the qb-simulation $M_1 \preceq^{qb}_{\Sigma_\varphi} M_2$ is also well-formed w.r.t. φ. A *block simulation* ($M_1 \preceq^{b}_{\Sigma_0} M_2$ in symbols) is a qb-simulation w.r.t. Σ_0, $Observ(M_1, \Sigma_0)$, $Observ(M_2, \Sigma_0)$.

Block simulation is similar to block bisimulation which was defined in [15] for the purpose of checking correctness properties for parameterized distributed systems composed of similar processes connected in ring network. It is also close to visible simulation introduced in [3] and studied in [26]. Quasi-block simulation is an extension of block simulation. The necessity of this extension stems from the fact that asynchronous composition of LTSs (unlike synchronous one) is not monotonic w.r.t. block simulation.

3 The Basic Features of Quasi-block Simulation

Proposition 1. $M_1 \preceq^{b}_{\Sigma_0} M_2 \Longrightarrow M_1 \preceq^{qb}_{\Sigma_0} M_2$.

This statement follows immediately from the definitions above. Moreover, qb-simulation can be reduced to block simulation.

Consider two LTSs $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i \rangle$, $i = 1, 2$, such that $M_1 \preceq^{qb}_{\Sigma_0} M_2$ w.r.t. sets of events E^1 and E^2. Denote by ε an auxiliary visible action such that $\varepsilon \notin A^1 \cup A^2$, and build the LTSs $\widetilde{M}_i = \left\langle S^i, S^i_0, A^i \cup \{\varepsilon\}, \widetilde{R}^i, \Sigma^i, L^i \right\rangle$, $i = 1, 2$, such that $(s, a, t) \in \widetilde{R}^i$ iff either $a \neq \varepsilon$ and $(s, a, t) \in R^i$, or $a = \varepsilon$ and $(s, \tau, t) \in E^i$. Thus, ε marks all those invisible transitions that are included in the sets of events E^1 and E^2.

Theorem 1. $M_1 \preceq^{qb}_{\Sigma_0} M_2 \Longleftrightarrow \widetilde{M}_1 \preceq^{b}_{\Sigma_0} \widetilde{M}_2$.

Theorem 1 may have a considerable utility in checking qb-simulation, since it provides a way of taking an advantage of efficient simulation-checking algorithms [11, 17] that are applicable to block simulation. Quasi-block (unlike visible or block simulations) is preserved under asynchronous compositions of LTSs.

Theorem 2. *Let* $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i \rangle$, $i = 1, 2, 3, 4$, *be four LTS's such that* $(\Sigma^1 \cup \Sigma^2) \cap (\Sigma^3 \cup \Sigma^4) = \emptyset$, $A^1 = A^2 = A'$, $A^3 = A^4 = A''$, *and* $A' \cap A'' = \emptyset$. *Let* Σ' *and* Σ'' *be the distinguished sets such that* $\Sigma' \subseteq (\Sigma^1 \cup \Sigma^2)$ *and* $\Sigma'' \subseteq (\Sigma^3 \cup \Sigma^4)$. *Let* $\Gamma = (\Delta, \overline{\ })$ *be a synchronizer such that* $\Delta \subseteq A'$, *and* $\overline{\ } : \Delta \to A''$. *Then* $M_1 \preceq^{qb}_{\Sigma'} M_2$ *and* $M_3 \preceq^{qb}_{\Sigma''} M_4$ *implies* $M_1 \parallel_\Gamma M_3 \preceq^{qb}_{\Sigma' \cup \Sigma''} M_2 \parallel_\Gamma M_4$.

Yet another simulation which has close relationships with quasi-block one is stuttering simulation. It was introduced in [3] and enjoys wide applications in the framework of partial order reduction technique (see [9, 18]).

Let $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i \rangle$, $i = 1, 2$, be two LTSs, and $\Sigma_0 \subseteq \Sigma^1 \cap \Sigma^2$. A relation $H \subseteq S^1 \times S^2$ is called a *stuttering simulation* w.r.t. Σ_0 iff every pair $(s', s'') \in H$ complies with the following requirements:

1. $L^1(s') \cap \Sigma_0 = L^2(s'') \cap \Sigma_0$.
2. For every path π', $\pi' = {s'}_1 \xrightarrow{a_1} {s'}_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{k-1}} {s'}_k \xrightarrow{a_k} \ldots$, $s'_0 = s'$ there is a path π'', $\pi'' = {s''}_1 \xrightarrow{a_1} {s''}_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{k-1}} {s''}_k \xrightarrow{a_k} \ldots$, $s''_0 = s''$ and partitions $P'_1 P'_2 \ldots$, $P''_1 P''_2 \ldots$ of π' and π'', such that for every $i \geq 1$ the sub-paths P'_i and P''_i match, i.e. $(s', s'') \in H$ holds for every pair of states $s' \in P'$ and $s'' \in P''$.

We write $M_1 \preceq^{st}_{\Sigma_0} M_2$ to indicate the existence of stuttering simulation between M_1 and M_2.

Theorem 3. $M_1 \preceq^{qb}_{\Sigma_0} M_2 \Longrightarrow M_1 \preceq^{st}_{\Sigma_0} M_2$.

A proof of this theorem is straightforward.

Since stuttering simulation preserves the satisfiability of temporal formulae from $ACTL^*_{-X}$, Theorem 3 brings us to the following conclusion.

Theorem 4. *Suppose that a qb-simulation* $M_1 \preceq^{qb}_{\Sigma_\varphi} M_2$ *is well-formed w.r.t. a* $ACTL^*_{-X}$*-formula* φ*, and* $M_2 \models \varphi$*. Then* $M_1 \models \varphi$ *as well.*

As it may be seen from the definition, stuttering simulation does not take into account any actions, but even in the case when $A^1 = A^2 = \emptyset$ it is weaker than qb-simulation (see Example 2 in Appendix 1). The fact that qb-simulation is stronger than stuttering simulation implies that the former is easy for checking and more feasible for practical applications in the framework of induction-based verification techniques.

4 Applying Quasi-block Simulation to the Verification of Asynchronous Networks

There are very few papers (in fact, the authors of [4] were not aware of any) where the induction-based verification technique is applied to asynchronous networks. In [15] parameterized systems composed of identical asynchronous processes which are arranged in a ring topology and communicate by passing a boolean token were considered. It has been shown that for several classes of indexed CTL^*_{-X} properties a *cutoff* effect takes place, i.e. the verification of the whole parameterized system can be reduced the model checking of finitely many instances of the system. In a certain sense, a cutoff plays a role of an invariant for such systems. In [10] the results of Emerson and Namjoshi were extended from rings to other classes of asynchronous networks. Nevertheless, many interesting classes of parameterized asynchronous systems do not fall into this category.

To the best of our knowledge the only paper where induction-based verification is applied to asynchronous networks is that of Clarke, Grumberg, and Jha [8]. In this paper they represented parameterized systems by network grammars, and used regular languages to express state properties. To generate invariants they developed an *unfolding heuristics*: given a parameterized system $\{P_k\}_{k=1}^{\infty}$ to find n such that $h(P_{n+1}) \preceq h(P_n)$, where $\preceq$ is a strong simulation and h is some appropriate abstraction. Much attention has been paid to the development of effective technique for constructing required abstractions.

To demonstrate that quasi-block simulation makes it possible to get rid of abstraction in induction based invariant generation we apply this approach to the verification of Resource ReSerVation Protocol (RSVP).

4.1 Resource Reservation Protocol

Resource reservation protocol (RSVP) [28] is designed to reserve resources in a network. The nodes of a network that hold these resources are called *producers* (or *senders*). Resource reservation procedure is launched by the *consumers* of resources (they are also called *receivers*). For instance, a consumer may reserve a download rate on the path from a producer to play video files stored on the producer without visible latency. A primary feature of RSVP is its scalability: RSVP scales to very large multicast groups because it uses receiver-orientated

reservation requests that merge as they progress up the multicast tree. The reservation for a single receiver does not need to travel to the source of a multicast tree (producer); it travels only until it reaches a reserved branch of the tree. While RSVP is designed for multicast applications, it may also make unicast data flows. RSVP is a unidirectional protocol. Resource reservation requests are sent upstream, from consumers to producers, whereas data is sent downstream, from producers to consumers. Any node in a network is allowed to be both a producer and a consumer. The protocol supports multiple reservation sessions, i.e. one consumer may reserve some resources in one session and other resources in another session. So, at the same time a host may be both a consumer in one session and a producer in another one. RSVP does not perform its own routing; instead it uses underlying routing protocols to determine where it should carry reservation requests. As routing changes paths to adapt topology changes, RSVP adapts its reservation to the new paths wherever reservations are in place.

Producers and consumers communicate by sending messages. We will restrict our consideration to the messages of the following 8 types: *path*, *resv*, *path_teardown*, *resv_teardown*, *path_refresh*, *resv_refresh*, *path_error*, *resv_error*. A *path* message is sent downstream by a producer to allocate available communication paths, whereas *resv* messages are sent along such paths upstream by consumers as requests for reservation of resources. A *path_teardown* message is sent by a producer when it breaks sending data. A *resv_teardown* message is sent by a consumer when it abandons receiving data. From time to time *path_refresh* messages are sent downstream to make sure that communication paths are still active, and *resv_refresh* messages are sent upstream as reservation acknowledgements. A *path error* message is sent upstream whenever an error occurs during path allocation. A *resv_error* message is sent downstream when a reservation error occurs.

The outline of communication in the protocol is as follows. Producers send *path* messages downstream along available routes that have been created by some routing protocol. In response to these messages consumers send upstream *resv* messages as reservation requests. Intermediate nodes (we will call them *routers*) check, whether reservation requests may be satisfied; if such is the case then they send reservation request upstream. The important feature of RSVP is that reservation requests are merged when sent upstream. When a reservation messages are delivered to a producer it sends data along the selected routes to the subscribed consumers. At any time a producer or a consumer may send a teardown message to cancel a communication session. The nodes refresh periodically the sessions by exchanging *path_refr* and *resv_refr* messages.

A model of RSVP. Some properties of RSVP have been already verified in [12, 30]. S.J. Creese and J. Reed in [12] used CSP to construct a formal model of some fragment of RSVP to check the property of reservation merging in a binary multicast tree. In [30] a model of RSVP with one producer, one router and one consumer has been studied by means of Coloured Petri Nets; this approach extended substantially the set of properties that were verified. We do not study

RSVP *in corpore* either and restrict our consideration to some formal scale-down model of RSVP. The principal simplifications are as follows.

Topology. Following [12] we consider binary trees only. In this case only one producer per session is allowed. The routers and consumers form a binary tree where the routers are placed in the intermediate nodes and the consumers are placed in the leaves. The only producer is attached to the router in the root of the tree.

Multicast vs. unicast. When dealing with a unicast messaging one have to attach a destination (and possibly source) address to every message. In this case one should either to consider models with potentially infinite state space (as address size grows with the number of processes), or to consider models with a bounded number of consumers. In our model the producer uses multicast messages only. A router does not necessarily send messages to both descendants anyway.

Reservation decisions. In RSVP each intermediate node uses Admission Control to check, whether the node has sufficient available resources‘to supply the requested quality of service. In our current model it is assumed that the routers have an unbound amount of resources, which leads to a successfull reservation on every request. We also take no account of Policy Control which is intended to determine whether the consumer has a a permission to make the reservation.

Failures. We assume that all channels and processes are reliable: no messages could be lost and no routers could be stuck.

Other abstractions. When building our model we do not touch such issues as reservation confirmations, policy control, and timeouts.

The model of RSVP we deal with is is a family of parameterized systems generated by the following network grammar G:

$$\begin{aligned} P &\to p \parallel_1 T \\ T &\to r \parallel_2 T \parallel_3 T \\ T &\to r \parallel_2 c \parallel_3 c \end{aligned}$$

where terminals p (producer), c (consumer), and r (router) are finite state processes. We denote by $\mathcal{L}(G)$ the set of networks generated by the grammar G.

As G generates only tree networks, it is more suitable to use traditional algebraic notation for writing network grammar terms. Thus, we will write $p(T)$, $r(T,T)$, $r(c,c)$ for the terms in the righthand side of the grammar rules. For instance, a term $p(r(r(c,c),r(c,c)))$ specifies a model composed of one producer, three routers, and four consumers; the network of this model is a tree of height 3. Sometimes a behaviour of a producer is of no importance for the analysis of a model. In these cases we restrict our consideration to tree networks derived from non-terminal T only. Such networks are specified by the terms of the form $r(t_1,t_2)$. In what follows when studying qb-simulation $r(t_1,t_2) \preceq^{qb}_{\Sigma(r)} r(t'_1,t'_2)$

between reduced models $r(t_1, t_2)$ and $r(t'_1, t'_2)$ we will assume that $\Sigma(r)$ is the set of atomic propositions of the topmost router r. Similarly, we assume that $\Sigma(p)$ is the set of atomic propositions of the producer p.

The models of RSVP were implemented in the model description language PROMELA [27] in the framework of the open-source software verification tool SPIN [29]. The usage of Promela gave us an advantage of exploiting a simulator and model checker from SPIN.

4.2 Finding Invariants

We apply the induction technique to verify the infinite family of parameterized finite state systems generated by the network grammar G. The key point of our approach is that of finding an invariant of $\mathcal{L}(G)$. We made an attempt to find such invariant by analyzing the networks derived from non-terminal symbol T.

Denote by T_N the set of tree networks of the height N derived from T.

Proposition 2. *If the following qb-simulations are valid*

$$r(r(c,c),c) \preceq^{qb}_{\Sigma(r)} r(c,r(c,c)) \tag{1}$$

$$r(r(c,c),r(c,c)) \preceq^{qb}_{\Sigma(r)} r(c,r(c,c)) \tag{2}$$

$$r(c,r(c,r(c,c))) \preceq^{qb}_{\Sigma(r)} r(r(c,c),r(c,c)) \tag{3}$$

$$r(r(c,r(c,c)),c) \preceq^{qb}_{\Sigma(r)} r(r(c,c),r(c,c)) \tag{4}$$

$$r(r(c,r(c,c)),r(c,r(c,c))) \preceq^{qb}_{\Sigma(r)} r(r(c,c),r(c,c)) \tag{5}$$

then any network t from T_3 is qb-simulated by the model $r(r(c,c),r(c,c))$, i.e. $t \preceq^{qb}_{\Sigma(r)} r(r(c,c),r(c,c))$.

Proof. Assumption (1) allows us to rotate a tree. Applying assumptions (1) and (2) to the subtrees of a network from T_3 we conclude that for any $t \in T_3$ there exists such a $t' \in \{r(c,r(c,r(c,c))), r(r(c,r(c,c)),c), r(r(c,r(c,c)),r(c,r(c,c)))\}$ that $t \preceq^{qb}_{\Sigma(r)} t'$. In view of the assumptions (3), (4) and (5) the latter is reduced to qb-simulaion $t \preceq^{qb}_{\Sigma(r)} r(r(c,c),r(c,c))$. All these reductions can be made due to the monotonocity Theorem 2. This completes the proof. □

As it may be seen from Proposition 2 the model $Inv = r(r(c,c),r(c,c))$ is a keystone of our construction.

Proposition 3. *If the assumptions (1)–(5) are valid, then the model Inv qb-simulates every network t from T_N, $N \geq 3$, i.e. $t \preceq^{qb}_{\Sigma(r)} Inv$.*

Proof. By induction on the height N of tree networks. By Proposition 2, if qb-simulations (1)–(5) are valid then $t \preceq^{qb}_{\Sigma(r)} Inv$ holds for every network t from T_3. Suppose that Inv qb-simulates every network from T_N . Let t be an arbitrary network T_{N+1}. Consider a tree network t' which is obtained from t by

substituting the network Inv instead of every subtree of height 3 in t. Clearly, the height of t' is N. Furthermore, by Theorem 2, if Inv qb-simulates every network from T_3 then $t \preceq^{qb}_{\Sigma(r)} t'$. Hence, $t \preceq^{qb}_{\Sigma(r)} Inv$. □

By combining Theorem 2 and Proposition 3, we arrive at the following conclusion.

Corollary 1. *If the assumptions (1)–(5) are valid then the model $p(Inv)$ is an invariant of the family of models generated by the network grammar G, i.e. $p(t) \preceq^{qb}_{\Sigma(r)\cup\Sigma(p)} p(Inv)$ holds for every network t of height $N, N \geq 3$, derived from non-terminal T.*

Thus, to certify that the finite model $p(Inv)$ is an invariant of the infinite family of models $\mathcal{L}(G)$ it is sufficient to make certain that the assumptions (1)–(5) are true.

4.3 Checking Quasi-block Simulation

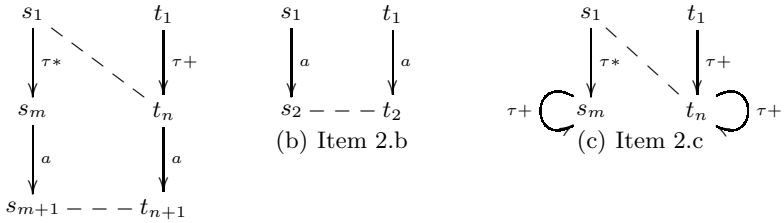
When checking quasi-block simulation between finite models we rely on Proposition 1. Since $M_1 \preceq^{b}_{\Sigma} M_2$ implies $M_1 \preceq^{qb}_{\Sigma} M_2$, we reduce quasi-block simulation checking of the assumptions (1)–(5) to block simulation checking of the corresponding models.

To check that a pair of states (s, u) in some LTSs M_1, M_2 complies with the definition of block simulation one should check each pair in every finite block outgoing from the states s and u. To reduce the complexity of checking procedure we introduce a new concept of *semi-block simulation*. Let $M_i = \langle S^i, S^i_0, A^i, R^i, \Sigma^i, L^i \rangle$, $i = 1, 2$, be two LTSs. Let Σ_0 be a subset of $\Sigma^1 \cap \Sigma^2$, and E^1 and E^2 be some sets of events of M_1 and M_2.

Definition 5. *A binary relation $H \subseteq S^1 \times S^2$ is called a* semi-block simulation *on M_1 and M_2 w.r.t. Σ_0, E^1, E^2, iff every pair $(s_1, t_1) \in H$ meets the following requirements:*

1. $L^1(s_1) \cap \Sigma_0 = L^2(t_1) \cap \Sigma_0$,
2. *For every finite block $B' = s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{a} s_{m+1} \in MAXF(E^1, s_1)$ there is a block $B'' = t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_n \xrightarrow{a} t_{n+1} \in MAXF(E^2, t_1)$ such that:*
   (a) if $n > 1$ then $(s_1, t_n) \in H$ and $(s_{m+1}, t_{n+1}) \in H$;
   (b) if $n = 1$ then $(s_{m+1}, t_{n+1}) \in H$;
3. *(divergency) if a τ-cycle is reachable from state s_1 by some τ-path, i.e. $MAXI(E_1, s_1) \neq \emptyset$, then there exists such a τ-cycle γ reachable from state t_1 by some τ-path and a state $t_n \in \gamma$ that: $(s_1, t_n) \in H$.*

The items of this definition are illustrated in Fig. 4.3. We write $M_1 \preceq^{sbs}_{\Sigma_0} M_2$ iff there exist two sets of events E^1 and E^2 of LTSs M_1 and M_2 and a binary relation $H \subseteq S^1 \times S^2$ such that H is a semi-block simulation on M_1 and M_2 w.r.t. Σ_0, E^1, E^2, and for every initial state $s_0 \in S^1_0$ there exists an initial state $t_0 \in S^2_0$ such that $(s_0, t_0) \in H$.



(a) Item 2.a (b) Item 2.b (c) Item 2.c

Fig. 1. Semi-block simulation.

Theorem 5. $M^1 \preceq^{sbs}_{\Sigma_0} M^2 \Longleftrightarrow M^1 \preceq^{bs}_{\Sigma_0} M^2$.

We implemented a naive algorithm for checking semi-block simulation which works iteratively according to the definition above. It adds all pairs with the same labelling of states to the relation and then removes those pairs that do not satisfy the definition. However, in this case at the first iteration the relation may have a very large size close to $|M_1| \times |M_2|$. To avoid such effect we use a lazy approach. All pairs in the relation are partitioned into two groups: positive pairs (that satisfy the definition) and negative pairs (that do not satisfy it). The computation of a semi-block simulation begins with considering the initial pairs as positives. Next, the positive pairs are checked one by one following the definition. The computation continues while some pairs are brought into consideration or change their status from positive to negative. Since the set of negative pairs grows monotonically the computation will eventually terminate.

A semi-block simulation checking algorithm has been implemented in C++ using `std::map` to store relation. Our simulation-checking tool built the required block simulations that validate the assumptions (1)–(5). When checking $r(r(c, r(c, c)), r(c, r(c, c)))$ versus Inv we had ran out of memory (2 Gigabytes). To bypass the memory limitation we implemented a relation storage as a minimized automaton from Spin (see [25]). We found that this not only drastically decreases memory consumption, but also speed-up the computation.

The results of our experiments are depicted in Table 1. The program was running on 2.4 GHz AMD Opteron provided by Laboratory of Computer Systems, Moscow State University.

The checking of $r(r(c, r(c, c), r(c, r(c, c))$ vs Inv takes a considerable amount of time. As checking of each pair is performed proportionally fast and computation slows up after about 50 millions of positives added, we guess two reasons of such a degradation. First, we used state enumeration in minimized DFA, which looks to be less efficient than insertion, deletion and lookup operations. Second, minimized DFA hash tables and splay trees should be tuned up on large problems.

Table 1. Validating the assumptions (1)–(5). Results.

Assumption on M_1 and M_2	#States of M_1	#States of M_2	#Positive pairs	#Negative pairs	Time	Memory	Valid
(1)	1732	1277	15902	905	2 sec	22M	yes
(2)	24993	1277	223304	4437	31 sec	39M	yes
(3)	14672	24993	1425766	5443	7 min	43M	yes
(4)	21659	24993	3.8×10^6	9002	19 min 30 sec	44M	yes
(5)	3.8×10^6	24993	3.5×10^8	116172	76 hours	49M	yes

4.4 Using Invariants for Checking Properties of RSVP.

As it follows from Theorem 4 and Corollary 1, to check the whole infinite family of networks $\mathcal{L}(G)$ against any $ACTL^*_{-X}$-specification φ it is sufficient to demonstrate that the invariant $p(r(r(c,c),r(c,c)))$ and networks $p(r(c,r(c,c)))$, $p(r(r(c,c),c))$, $p(r(c,c))$, whose height is less than 3, satisfy φ.

Our main efforts were focused on the computation of block simulation and thus finding a desired invariant. So, we have not checked a lot of properties. One property we checked on the model $p(r(r(c,c),r(c,c)))$ (one producer, three routers, and four consumers) is the requirement that the consumer does not receive *path tear acknowledge* while it does not send *path teardown* message. This specification is expressed in Linear Temporal Logic as:

$$\varphi_1 = \mathsf{G}\neg producer!path\_tear \rightarrow \mathsf{G}\neg producer?path\_tear\_acknowledge \quad (6)$$

The second checked property is the property of reservation merging, which had been checked in [12]. In terms of our model it is written as an absence of reservation request while router is in "reserved" state:

$$\varphi_2 = \mathsf{G}(router_1.reserved \rightarrow \neg router_1.parent!resv) \quad (7)$$

By applying Spin model-checker we found that specification φ_1 is satisfied on $p(r(r(c,c),r(c,c)))$ and φ_2 is satisfied on $r(r(c,c),r(c,c))$. Due to Proposition 3 and Corollary 1 we conclude that properties are satisfied on any tree of length greater than 1:

$$p(w) \models \varphi_1, \text{ where } w \in \bigcup_{i=2}^{\infty} T_i \quad (8)$$

$$w \models \varphi_2, \text{ where } w \in \bigcup_{i=2}^{\infty} T_i \quad (9)$$

5 Conclusions and Directions for Future Research

There is a number of tasks to be solved next to make a good "reputation" for quasi-block simulation. Certainly, we have to find out some other practical case studies that could indicate convincingly the advisability of using quasi-block simulation in the verification of parameterized systems. Thus, we are about to extend our condensed version of RSVP by introducing to the model bounds on the resources and adding some elements from Admission Control module. We are also interested in checking fault tolerance properties under the assumption that some routers in communications paths may fail. This line of investigation depends to a large extent also on how much effectively quasi-block simulation can be checked. We assume that Theorems 1, 5 could give an essential prerequisite for constructing efficient checking procedures. So far we use an explicit representation of simulation relations (with a minor usage of symbolic techniques for storing them). We think that further improvements may be achieved with the application of a game-theoretic approach in conjunction with symbolic computations. Of some interest is also a search of some parameterized system which can be served as indicative counter-example to reveal the limitations of quasi-block simulations and to outline the ways for its further improvement.

References

1. Apt K.R., Kozen D. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986, p. 307–309.
2. Bosnacki D., Dams D., Holenderski L. A heuristic for symmetry reductions with scalarset. In *Procedings of FME2001*, Lecture Notes in Computer Science, 2021, 2001, p. 518–533.
3. Brown M.C., Clarke E.M., Grumberg O. Characterizing finite Kripke structures in propositional temporal logics. *Theoretical Computer Science*, v. 59, 1988, p. 115–131.
4. Calder M., Miller A. Five ways to use induction and symmetry in the verification of networks of processes by model-checking. in *Proceedings of AvoCS 2002 (Automated Verification of Critical Systems)*, 2002, p. 29–42
5. Clarke E.M., Grumberg O., Long D.E. Model checking and abstraction. In *Proceedings of Principles of Programming Languages*, 1992.
6. Clarke E.M., Filkorn T., Jha S. Exploiting symmetry in temporal logic model checking. In *Proceedings of CAV'93*, Lecture Notes in Computer Science,697, 1993, p. 450–461.
7. Clarke E.M., Grumberg, O., and Jha, S. Verifying parameterized networks using abstraction and regular languages, In *Proceedings of the 6-th International Conference on Concurrency Theory*, 1995.
8. Clarke E.M., Grumberg, O., and Jha, S. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997, p. 726–750.
9. Clarke E.M., Grumberg O., Peled D.A. Model checking. MIT Press, 1999.
10. Clarke E., Talupur M., Touili T., Veith H. Verification by network decomposition. In *Proceedings of CONCUR'04*, Lecture Notes in Computer Science, 3170, 2004, p. 276–291.

11. Cleaveland R., Sokolsky O. Equivalence and Preorder Checking for Finite-State Systems, In *Handbook of Process Algebra*, Elsevier, 2001, p. 391–424.
12. S.J. Creese and J. Reed. Verifying End-to-End Protocols Using Induction with CSP/FDR. In IPPS/SPDP Workshop, 1999, p. 1243–1257.
13. Dams D., Grumberg O., Gerth R. Abstract interpretation of reactive systems: abstractions preserving $ACTL^*$, $ECTL^*$ and CTL^*. In *IFIP Working Conference and Programming Concepts, Methods and Calculii*, 1994.
14. Donaldson A.F. Miller A. Automatic symmetry detection for model checking using computational group theory. In *Proceedings of the 13th International Symposium on Formal Methods Europe (FME 2005)*, Lecture Notes in Computer Science, 3582, 2005, p. 481–496.
15. Emerson E.A., Namjoshi K.S. Reasoning about rings. In *Proceedings 22th ACM Conf. on Principles of Programming Languages, POPL'95*, 1995, p.85–94.
16. Emerson E.A., Sistla A.P. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2), 1996, p.105–131.
17. Etessami K., Schuller R., Wilke T. Fair simulation relations, parity games, and state space reduction for Buchi automata. In *Proceedings of 28th International Collquium "Automata, Languages and Programming"*, Lecture Notes in Computer Science, 2076, 2001, p. 694–707.
18. Gerth R., Kuiper R., Peled D., Penczek W. A partial order approach to branching time logic model checking. *Information and Computation*, 150(2), 1999, p.132–152.
19. Ip C.N., Dill D.L. Verifing systems with replicating components in murϕ. *Formal Methods in System Design*, 14, 1999, p.273–310.
20. Kesten Y., Pnueli A. Verification by finitary abstraction. *Information and Computation*, 163, 2000, p.203–243.
21. Konnov I.V., Zakharov V.A. An approach to the verification of symmetric parameterized distributed systems. *Programming and Computer Software* 31(5), 2005, p. 225–236.
22. Kurshan R.P., MacMillan K.L. Structural induction theorem for processes. In *Proceedings of the 8-th International Symposium on Principles of Distributed Computing, PODC'89*, 1989, p. 239–247.
23. Lesens D., Saidi H. Automatic verification of parameterized networks of processes by abstraction. In *Proceedings of the 2-nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, 1997.
24. Manku G.S., Hojati R., Brayton R.K. Structural symmetries and model checking. In *Proceedings of International Conference on Computer-Aided Verification (CAV'98)*, 1998, p. 159–171.
25. Puri A., Holzmann G.J. A Minimized automaton representation of reachable states. Software Tools for Technology Transfer, 1999, Vol. 3, No. 1, pp. 270–278.
26. Penczek W., Gerth R., Kuiper R., Szreter M. Partial order reductions preserving simulations. In *Proceedings of the CSP'99 Workshop*, 1999, p. 153–171.
27. Promela Reference Manual. http://spinroot.com/spin/Man/promela.html
28. Request for Comments 2205: Resource ReserVation Protocol. http://tools.ietf.org/html/rfc2205.
29. Spin Model Checker. http://spinroot.com
30. M. Villapol. Modelling and Analysis of the Resource Reservation Protocol using Coloured Petri Nets. PhD Thesis, Institute for Telecommunications Research and Computer Systems Engineering Centre, University of South Australia, 2003.
31. Wolper P., Lovinfosse. Properties of large sets of processes with network invariants. Lecture Notes in Computer Science, 407, 1989, p. 68–80.

Automated Polynomial Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema *

Laura Kovács

Research Institute for Symbolic Computation, Linz
kovacs@risc.uni-linz.ac.at

Abstract. The current paper presents an overview on the polynomial loop invariant generation method using algebraic techniques (symbolic summation and polynomial algebra), implemented in *Mathematica*, and used for imperative program verification in the *Theorema* system. For a subfamily of loops, called P-solvable imperative loops, we present completeness aspects of the approach in generating polynomial equations as invariants. The application of the method is demonstrated in examples working on numbers.

1 Introduction

One of the most rigorous ways to imperative program verification is by use of simple formal calculations based on versions of Hoare logic, e.g. the Floyd-Hoare-Dijkstra method involving weakest precondition calculations [15, 11, 8]. Such calculations allow us to capture the semantics of small program components and in the process derive specifications that can be used to assess whether a given program fragment has the required functionality.

It is well known that generation of loop invariants is in fact the challenging part of the Floyd-Hoare-Dijkstra method. This paper discusses an approach for automatically generating polynomial equations as loop invariants by advanced techniques from algorithmic combinatorics and polynomial algebra, using also the symbolic manipulation capabilities of the computer algebra system *Mathematica* on which the *Theorema* system (www.theorema.org) [4] resides.

This work is built on already published theoretical results and practical experiments regarding verification of imperative programs in *Theorema* [22, 23]. More specifically, it is an overview of how the generation of polynomial invariants for imperative loops can be automated using algebraic techniques for imperative program verification in *Theorema*. Our goal in this paper is to present and

* The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project. The *Theorema* system is supported by FWF (Austrian National Science Foundation) - SFB project F1302.

exemplify the main ideas and requirements that are involved in the invariant generation process; for technical details of the presented algorithms and properties we refer to our earlier works [22, 23, 20].

Polynomial identities found by an automatic analysis are useful for program verification, as they provide non-trivial valid assertions about the program, and thus significantly simplify the verification task. Finding valid polynomial identities (i.e. invariants) has applications in many classical data flow analysis problem [24], e.g. constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

In our work, a subfamily of imperative loops, called *P-solvable* loops (standing for polynomial solvable) has been identified, for which the value of each program variable can be expressed as a linear combination of algebraic exponential terms (i.e. the exponential base is an algebraic number) in the loop counter with polynomial coefficients in the loop counter, with the property that the exponentials are polynomially related. Hence, the loop variables are polynomials in the loop counter and algebraic exponentials. For such loops with only assignment statements, our method is complete in generating a finite set of polynomial relations among the loop variables, from which any polynomial invariant can be derived, whereas for the case of P-solvable loops with conditionals the completeness of our approach is still under study.

The key steps of our invariant generation method are:
(i) assignment statements from a P-solvable loop body are extracted yielding a system of recurrence equations, describing the behavior of the loop variables that are changed at each iteration;
(ii) advanced methods from algorithmic combinatorics implemented in *Mathematica* [26, 18] are used to determine the closed form solution of the recurrence equations of each loop variable, i.e. solutions that are function of the loop counter. The closed forms might contain algebraic exponential terms in the loop counter;
(iii) Algebraic (polynomial) dependencies among these algebraic exponential terms in the loop counter are computed using algebraic and combinatorial methods implemented in *Mathematica* [19].
As a result of these steps, every loop variable is expressed as a polynomial in terms of the loop counter and polynomially related algebraic exponentials. The coefficients of the polynomials are determined by the initial values (when the loop is entered) of the loop variables;
(iv) Finally, loop counters and the polynomially related exponential terms are eliminated using the Gröbner basis algorithm [3] implemented in *Mathematica*, and a finite set of polynomial identities among the program variables as invariants are determined. From this finite set, any polynomial identity serving a loop invariant can be derived.

The approach has been implemented in *Mathematica*, successfully applied on a number of programs working on numbers, and used for *imperative program verification* in *Theorema* [20]. In this imperative programming environment, the constructs of the programming language are: assignments, blocks, conditionals,

`For` and `While` loops (annotated with loop invariants and termination terms) and procedure calls. Programs are considered as procedures, without return values and with input, output and/or transient parameters.

The design of a framework for imperative program verification in an expressive logic like *Theorema* is driven by two main reasons. On the one hand, we wanted to automatize the weakest precondition strategy in order to generate verification conditions, thus proof obligations, in the *Theorema* syntax. The automatically inferred polynomial invariants are used in this process. On the other hand, we want to apply the *Theorema* provers to prove the verification conditions, producing in this way useful case studies for the development of the existing *Theorema* provers.

The rest of the paper is organized as follows: Section 2 gives a short overview on related work for invariant generation, followed by Section 3 containing the presentation of some theoretical notions that are used further in the paper. In Section 4 we present our method for polynomial invariant generation, and illustrate it on concrete examples. Section 5 concludes with some ideas for the future work.

2 Related Work

There have been many attempts to the automated generation of loop invariants, starting with the works [9, 12, 17], but with a limited success, for cases where only few arithmetic operations (mainly additions) among program variables were involved.

Recently, due to the increasing power of computer algebra and computational logic, the problem of automated invariant generation became again a challenging research topic. Several methods were applied to perform successful invariant generation, e.g. abstract interpretation [1, 2, 30, 6], quantifier elimination [5, 16], polynomial algebra [25, 29, 16, 27].

In [29], non-linear invariants are generated using forward propagation, by reducing the non-linear invariant generation problem to a numerical *constraint solving* problem, where the constraints describe properties of the unknown coefficients of the polynomial loop invariant with *a priori fixed degree*. A similar approach is used in [16], where a method for invariant generation using quantifier-elimination is proposed, by generating (and solving) constraints on the coefficients of the *parameterized* polynomial invariant with *fixed degree*. In [25], backward propagation is performed for non-linear programs without branch conditions, by computing a polynomial ideal that represents the weakest precondition for the validity of a *generic polynomial relation (with fixed degree)* at the target program point. In all these works, at the end of the invariant generation procedure, one or more polynomial invariants of an a priori fixed degree are obtained, but one has to further make a guess whether more polynomial invariants are needed and what are their degrees. In contrast, the method presented in this paper does not need an a priori assumption on the number of polynomials or on their degrees, and it is proved complete in generating polynomial identities

for P-solvable loops with only assignments from which any polynomial invariant can be derived.

A very interesting approach based on a fix-point algebraic procedure for the generation of all polynomial invariants is presented in [27, 28], by computing iteratively the Gröbner basis of a certain polynomial ideal. The method works when the list of (conditional) assignments present in the loop constitutes a *solvable mapping* - as it is defined in the paper. It appears that "solvable mappings" are similar to the notion of P-solvable loops introduced by us. However, we are also detecting automatically the algebraic dependencies between the (rational or non-rational) algebraic exponential parts of the closed forms, an operation which is also used in the above cited work, but whose solution is obtained only for rational exponential terms.

The approach presented in [28] is *complete* in generating all polynomial invariants for *simple loops* (loops with conditionals and solvable mapping-assignments), a property which in our work is proved so far only for the case of P-solvable loops with assignments. Using the examples presented in related works as well, our practical experiments show that our approach is also complete in finding all polynomial invariants for P-solvable loops with conditionals.

3 Preliminaries

We present some relevant definitions and properties of recurrence equations and polynomial ideals which are needed later on. For additional details, the interested reader might consult [7, 10, 19]. In what follows, we assume that $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{Q}$, $\mathbb{R}$, etc.), and by $\bar{\mathbb{K}}$ we denote its algebraic closure.

Definition 1. *A* Gosper-summable recurrence $f(n)$ *in* $\mathbb{K}$ *is a recurrence of the form:*

$$f(n+1) = f(n) + h(n+1) \ (n \geq 1), \tag{1}$$

where $h(n)$ *is a hypergeometric term [13] in* $\mathbb{K}$*, e.g.* $h(n)$ *can be a product of factorials, binomials, pochhammers, rational-function terms and exponential expressions in the summation variable* n *(all these factors can be raised to an integer power).*

The closed-form solution of a Gosper-summable recurrence can be exactly and algorithmically computed [13]; for doing so, we use the recurrence solving package `zb`, implemented in *Mathematica* by the RISC Combinatorics group [26].

Example 1. Given the Gosper-summable recurrence,

$$f(n+1) = f(n) + n * 2^n, \ n \geq 0, \qquad \text{with initial value:} f(0)$$

its closed form solution is:

$$f(n) = f(0) + 2 + 2^n(n-2).$$

Definition 2. *A* C-finite recurrence $f(n)$ *[31, 10] in* $\mathbb{K}$ *is a (homogeneous) linear recurrence with constant coefficients, i.e. it is of the form:*

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \ldots + a_{r-1} f(n+r-1) \ (n \geq 1), \qquad (2)$$

where r *is the* order *of the recurrence, and* $a_0, \ldots, a_{r-1}$ *are constants from* $\mathbb{K}$*, with* $a_0 \neq 0$*.*

By writing x^i *for each* $f(n+i)$*,* $i = 0, \ldots, r$*, the corresponding* characteristic polynomial $c(x)$ *of* $f(n)$ *is:*

$$c(x) = x^r - a_0 - a_1 x - \cdots - a_{r-1} x^{r-1}. \qquad (3)$$

A crucial and elementary fact about C-finite recurrences is that they always admit a closed form solution [10]:

Proposition 1. *The closed form of a C-finite sequence* $f(n)$ *in* $\mathbb{K}$ *is:*

$$f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n, \qquad (4)$$

where $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ *are the distinct roots of the characteristic polynomial of* $f(n)$*, and* $p_i(n)$ *is a polynomial in* n *whose degree is less than the multiplicity of the root* θ_i $(i = 1, \ldots, s)$*.*

Another important property that our invariant generation algorithm relies on is that an *inhomogenous* C-finite recurrence, i.e.

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \ldots + a_{r-1} f(n+r-1) + g(n) \ g(n) \neq 0,$$

with the property that $g(n)$ is a linear combination of terms $n^d \theta^n$, $\theta \in \bar{\mathbb{K}}$, $d \geq 0$, can always be transformed into an equivalent homogenous C-finite recurrences, thus its closed form can be exactly computed.

For obtaining the closed-form solutions of C-finite recurrences we use the `SumCracker` package, a *Mathematica* implementation by the RISC Combinatorics group [18].

Example 2. Given the C-finite (inhomogenous) recurrence:

$$f(n+1) = 2 * f(n) - 2 * \left(2 - \frac{1}{2^n}\right) - \frac{1}{2^{n+1}}, \quad \text{with initial value } f(0),$$

its closed form solution is:

$$f(n) = 2^n * f(0) + 4 - 2^{-n} - 3 * 2^n$$

Definition 3. *Let* $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ *be algebraic numbers, and their corresponding exponential sequences* $\theta_1^n, \ldots, \theta_s^n \in \bar{\mathbb{K}}$*.*
An algebraic dependency *(or* algebraic relation*) [19] of these sequences is a polynomial* p *such that*

$$p(\theta_1^n, \ldots, \theta_s^n) = 0, \quad (\forall n \geq 1). \qquad (5)$$

For automatically determining *all* (i.e. the ideal of) algebraic dependencies among exponential sequences, we use the `Dependencies` package [19] implemented in *Mathematica* by the RISC combinatorics group.

Example 3. Algebraic Dependencies.

- The algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = \frac{1}{2}$ is:
$$\theta_1^n * \theta_2^n - 1 = 0$$
- The algebraic dependency among the exponential sequences of $\theta_1 = \frac{1+\sqrt{5}}{2}, \theta_2 = \frac{1-\sqrt{5}}{2}$ is:
$$\left(\theta_1^n\right)^2 * \left(\theta_2^n\right) - 1 = 0$$
- There is no algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 3$.

Since algorithmic manipulations of ideals are needed, we use the Buchberger algorithm for Gröbner basis computation [3]. A Gröbner basis is a basis for an ideal that has special properties that make it possible to answer algorithmically questions about ideals, such as ideal membership of a polynomial, equality and inclusion of ideals, etc. A detailed presentation of the Gröbner bases theory can be found in [3].

4 Automated Polynomial Invariant Generation

The assignment statements from a loop body form a system of recurrence equations describing the behavior of loop variables. In other words, a recurrence equation of a loop variable allows the "indirect" computation of the loop variable values at any loop iteration, by first determining the values of the loop variable from previous loop iterations, and then apply the recurrence equation for getting the current value of the loop variable. When the number of iterations is large, this might take too long. Thus a *solution to the recurrence* would be more suitable for getting the value of the loop variable for any loop iteration, that is a *closed form* solution of the loop variable which does not require the computation of previous values of the loop variable.

Hence, finding closed form solutions for recurrence equations can be considered a major challenge in reasoning about imperative loops, for automatically inferring invariant relations among the loop variables.

Our algorithm for polynomial invariant generation combines computer algebra and algorithmic combinatorics, in such a way that at the end of the invariant generation process all valid polynomial assertions of a *P-solvable loop* are automatically obtained. The definition of P-solvable loops is available in our earlier conference papers [21, 23].

Informally, an imperative loop is a **P-solvable imperative loop** iff the closed form solution of the loop variables are linear combinations of geometric sequences $\theta^n \in \bar{\mathbb{K}}$ with polynomial coefficients $p(n) \in \mathbb{K}$ (where $n \in \mathbb{N}$ is the

loop counter and the coefficients of $p(n)$ are determined by the initial values of the loop variables), with the property that there exist algebraic dependencies among all θ^n.

The class of P-solvable loops includes simple loops in which the expressions in the assignment statements are affine mappings. Moreover, as presented in equation (4), any closed form solution of a loop variable defined by a C-finite recurrence is a linear combination of terms of the form $p_i(n)\theta_i^n$. Hence, by imposing the restriction of having algebraic dependencies among the exponential sequences θ_i^n, a loop with only C-finite assignments is P-solvable.

Since finding closed form expressions of recurrences in the general case is undecidable, it is necessary to distinguish among the types of recurrence equations that occur in practical imperative programs with loops. At the current stage of our work, we consider loops with the property that their assignment statements are either Gosper-summable, C-finite recurrences or can be handled by the technique of generating functions [14].

4.1 P-solvable Loops with Assignments Only

We give only a brief theoretical illustration of our algorithm, more details can be found in [22]. We denote by n the loop counter, and by $x_1, \ldots, x_m$ $(m \geq 1)$ the recursively changed loop variables.

Algorithm 4.1 "receives" as input a P-solvable loop with assignment statements only, and starts first with extracting and solving the recurrence equations of the loop variables, determining this way the solution system of closed forms (solutions that are linear combinations of exponential terms with polynomial coefficients in the loop counter). Next, it computes the set A of generators for the ideal of algebraic dependencies among the exponential terms. Finally, from A and the polynomial system of closed forms denoted by I, a set J of generators for the ideal of *all polynomial relations* among the loop variables is computed by elimination using Gröbner bases, w.r.t. a suitable elimination ordering.

Algorithm 4.1 P-solvable Loops with Assignments Only
Input: An imperative P-solvable loop with only assignment statements
Output: The generators for the polynomial ideal of the loop's polynomial invariants

1 Extract the recurrence equations of the loop variables;

I. Recurrence Solving.

2 Identify the type of recurrences and solve them. The closed form system is:

$$\begin{cases} x_1(n) = p_{1,1}(n)\theta_1^n + \cdots + p_{1,s}(n)\theta_s^n \\ x_2(n) = p_{2,1}(n)\theta_1^n + \cdots + p_{2,s}(n)\theta_s^n \\ \vdots \\ x_m(n) = p_{m,1}(n)\theta_1^n + \cdots + p_{m,s}(n)\theta_s^n \end{cases}, \text{ where } \begin{array}{l} \theta_j \in \bar{\mathbb{K}},\ p_{i,j} \in \bar{\mathbb{K}}[n], \\ j = 1, \ldots, s,\ i = 1, \ldots, m \end{array}$$

3 Identify the algebraic dependencies $t_k(\theta_1,\ldots,\theta_s)=0$ $(k>1)$.

Denote $A=\{t_k(\theta_1,\ldots,\theta_s)\mid k>1\}$, thus $\langle A\rangle \trianglelefteq \bar{\mathbb{K}}[y_0,\ldots,y_s]$

4 Introduce the notations: $y_0=n, y_1=\theta_1^n,\ldots,y_s=\theta_s^n$, and we have:

$$\begin{cases} x_1(n) = q_1(y_0,y_1,\ldots,y_s) \\ x_2(n) = q_2(y_0,y_1,\ldots y_s) \\ \vdots \\ x_m(n) = q_m(y_0,y_1,\ldots,y_s) \end{cases}, \text{ where } q_i\in\bar{\mathbb{K}}[y_0,y_1,\ldots,y_s],\ i=1,\ldots,m$$

II. Polynomial Invariant Generation.

5 Consider $I=\{x_1-q_1(y_0,\ldots,y_s),\ldots,x_m-q_m(y_0,\ldots,y_s)\}\cup A$.

Thus $I\subset\bar{\mathbb{K}}[y_0,y_1,\ldots,y_s,x_1,\ldots,x_m]$;

6 Compute the Gröbner basis J of I with an order $y_0>\cdots>y_s>x_1>\cdots>x_m$;
7 *return* $J=I\cap\mathbb{K}[x_1,\ldots,x_m]$

Remark 1. At step 2 of Algorithm 4.1, for solving recurrences we use the Gosper algorithm, handling C-finite recurrences or the technique of generating functions.

The principle of Algorithm 4.1 is presented in the following program fragment:

Example 4.

$x:=10; y:=10;$
$\texttt{While}[y>0,$
$x:=2*x;$
$y:=\frac{1}{2}*y-1]$

$$\overset{\Rightarrow}{Step}{}_1 \quad \begin{cases} k=0,\ldots,\mathbf{K} \\ x(k+1)=2*x(k) \\ y(k+1)=\frac{1}{2}*y(k)-1 \end{cases} \quad \overset{\Rightarrow}{Step}{}_2 \quad \begin{cases} k=0,\ldots,\mathbf{K} \\ x(k) \ {}_{C-\overline{finite}} \ 2^k*x(0) \\ y(k) \ {}_{C-\overline{finite}} \ \frac{1}{2^k}*(y(0)-2)+2 \end{cases}$$

$$\overset{\Rightarrow}{Step}{}_{3,\,4} \quad \begin{cases} k=0,\ldots,\mathbf{K} \\ a(k)=2^k,\ b(k)=2^{-k} \\ \begin{cases} x(k) & = & a(k)*x(0) \\ y(k) & = & b(k)*(y(0)-2)+2 \\ 0 & \overset{=}{_{Alg.Dep.}} & a(k)*b(k)-1 \end{cases} \end{cases}$$

$\overset{\Rightarrow}{Step}{}_{5,\,6,\,7}$ Polynomial Invariant: $2*x+x*y-120=0$,

where $\mathbf{K}$ denotes the unknown bound of the loop iteration counter k.

4.2 P-solvable Loops with Conditionals

We consider a generalization of Algorithm 4.1, namely we present an invariant generation algorithm of P-solvable loops with conditionals. The key idea is to

do first program transformation (see Prop. 2), namely transform the outer P-solvable loop with conditionals into nested P-solvable loops with assignments only (i.e. inner loops), and then apply Algorithm 4.1 to get the closed form solution system of each inner loop. Finally, by "merging" the closed form solution systems, using that initial values e.g. of the second inner loop are given by the final values of the first inner loop, and eliminating the variables in the inner loop indexes, we obtain polynomial relations for the outer loop.

Proposition 2. *Transformation Rule of Loops with Conditionals [22].*
The P-solvable loop with conditionals and assignments

$$\{I\} \quad \texttt{While}[b, c_1; \texttt{If}[b_1 \ \texttt{Then}\ c_2 \ \texttt{Else}\ c_3]; c_4] \quad \{I \wedge \neg b\} \tag{6}$$

is equivalent to the P-solvable loop with nested P-solvable loops containing only assignments:

$$\begin{array}{lll} & \texttt{While}[b, & \\ \{I\} & \texttt{While}[b \wedge b_1', c_1; c_2; c_4]; & \{I \wedge \neg b\} \\ & \texttt{While}[b \wedge \neg b_1', c_1; c_3; c_4]] & \end{array} \tag{7}$$

where I is a loop invariant and b_1' represents condition b_1 modified by the assignment statement c_1.
(For further purposes, we denote $S_1 = c_1; c_2; c_4$ and $S_2 = c_1; c_3; c_4$.)

What remains is to determine the relation between the polynomial invariants of the P-solvable loop (6) and the polynomial identities obtained from the variable elimination step performed on the "merged" closed form systems of the inner loops from (7). For doing so:

(i) first we showed that the polynomial relations among the variables from an arbitrary iteration of the P-solvable outer loop (7) are determined by:
  – "merging" the closed forms of the inner loops, by considering the two possibilities:
    1. the initial values of the second inner loop are given by the final values of the first inner loop;
    2. the initial values of the second inner loop are given by the final values of the first inner loop;
  – next, by eliminating the variables in the inner loop counters, we obtain polynomial relations among the loop variables that are candidate polynomial invariants for the outer loop;
  – finally, keep only the set of those polynomial relations whose conjunction is valid for both possible iteration traces S_1 and S_2 of (6), and thus they are polynomial invariants of (6);

(ii) next, we showed that any iteration of the P-solvable outer loop (7) admits the same set of polynomial relations among the loop variables;

(iii) hence for generating the polynomial invariants of (6), we consider only the first iteration of the P-solvable outer loop (7) whose polynomials are determined by step (i).

More details and proofs of the above steps can be found in [22]. Now we can formulate the algorithm for polynomial invariant generation for P-solvable loops with conditionals. W.l.o.g, we may assume that we have a P-solvable loop with only one conditional.

Algorithm 4.2 P-solvable Loops with Conditionals
Input: An imperative P-solvable loop with conditional and assignment statements
Output: Polynomial invariants of the loop

1 By Prop. 2, transform the initial loop into a nested loop system as in (7);

2 Apply Algorithm 4.1 for determining the merged closed form systems of the inner loops as described at steps (i).1 and (i).2;

3 Eliminate variables in the inner loop indexes, and obtain set F of possible polynomial invariants of the outer loop;

4 From F keep only the set G of those polynomials whose conjunction is valid for S_1 and S_2, i.e.:

$$G = \{p \in F \mid wp(S_1, p) \in \langle G \rangle \wedge wp(S_2, p) \in \langle G \rangle\} \subset F$$

where $wp(S_i, p)$is the weakest precondition of S_i w.r.t. p;

5 *return* G

For a step-by-step illustration of Algorithm 4.2, let us consider the following loop example:

Example 5.

$$\begin{array}{ll} & r := a-1;\ q := 1;\ p := 1/2; \\ (A) & \texttt{While}[(2*p*r \geq err), \\ & \texttt{If}[2*r-2*q*p \geq 0 \\ & \quad \texttt{Then}\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2, \\ & \quad \texttt{Else}\ r := 2*r;\ p := p/2] \end{array}$$

Step 1: Loop Transformation

$$\begin{array}{ll} (A) & \texttt{While}[(2*p*r \geq err), \\ (B) & \quad \texttt{While}[(2*p*r \geq err) \wedge (2*r-2*q*p \geq 0), \\ & \qquad r := 2*r-2*q-p;\ q := q+p;\ p := p/2]; \\ (C) & \quad \texttt{While}[(2*p*r \geq err) \wedge \neg(2*r-2*q*p \geq 0) \\ & \qquad r := 2*r;\ p := p/2]] \end{array}$$

Step 2/(i).1 - 1: Extracting system of recurrences for each inner loop

While loop (B):
$i = \overline{0, \mathbf{I}}$
$$\begin{cases} p(i+1) = p(i)/2 \\ q(i+1) = q(i) + p(i) \\ r(i+1) = 2*r(i) - 2*q(i) - p(i) \end{cases}$$

While loop (C):
$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$
$$\begin{cases} p(j'+1) = p(j')/2 \\ q(j'+1) = q(j') \\ r(j'+1) = 2*r(j') \end{cases}$$

where $\mathbf{I}, \mathbf{J}$ represent the unknown bounds of the iteration number of each loop counter i, j, respectively.

Step 2/(i).1 - 2: Solving recurrences for each inner loop

While loop (B):

$i = \overline{0, \mathbf{I}}$

$$\begin{cases} p(i) & _{C-\overline{finite}} & \frac{1}{2^i} * p(0) \\ q(i) & _{\overline{Gosper}} & q(0) + 2 * p(0) - \frac{1}{2^{i-1}} * p(0) \\ r(i) & _{C-\overline{finite}} & 2^i * (r(0) - 2 * q(0) - 2 * p(0)) - \\ & & \frac{1}{2^{i-1}} * p(0) + 2 * q(0) + 4 * p(0) \end{cases}$$

While loop (C):

$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$

$$\begin{cases} p(j') & _{C-\overline{finite}} & \frac{1}{2^j} * p(\mathbf{I}) \\ q(j') & = & q(\mathbf{I}) \\ r(j') & _{C-\overline{finite}} & 2^j * r(\mathbf{I}) \end{cases}$$

Step 2/(i).1 - 3: Introduction of extra variables with their algebraic dependencies

While loop (B):

$i = \overline{0, \mathbf{I}}$

$x(i) = 2^i, y(i) = 2^{-i}$

$$\begin{cases} p(i) & = & y(i) * p(0) \\ q(i) & = & q(0) + 2 * p(0) - 2 * y(i) * p(0) \\ r(i) & = & x(i) * (r(0) - 2 * q(0) - 2 * p(0)) - \\ & & 2 * y(i) * p(0) + 2 * q(0) + 4 * p(0) \\ 0 & _{\overline{Alg.Dep}} & x(i) * y(i) - 1 \end{cases}$$

While loop (C):

$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$

$u(j') = 2^{j'}, v(j') = 2^{-j'}$

$$\begin{cases} p(j') & = & v(j') * p(\mathbf{I}) \\ q(j') & = & q(\mathbf{I}) \\ r(j') & = & u(j') * r(\mathbf{I}) \\ 0 & _{\overline{Alg.Dep}} & u(j') * v(j') - 1 \end{cases}$$

Step 2/(i).1 - 4: Merged Closed Form System for Loops (B);(C)

While loop (C);(B)

$$\begin{cases} p = y * v * p(0) \\ q = q(0) + 2 * v * p(0) - 2 * y * v * p(0) \\ r = x * (u * r(0) - 2 * q(0) - 2 * v * p(0)) - 2 * y * v * p(0) + 2 * q(0) + 4 * v * p(0) \\ 0 = u * v - 1 \\ 0 = x * y - 1 \end{cases}$$

Step 2/(i).2: Merged Closed Form System for Loops (C);(B)

Similarly, we obtain:

While loop (C);(B)

$$\begin{cases} p = v * y * p(0) \\ q = q(0) + 2 * p(0) - 2 * y * p(0) \\ r = u * (x * (r(0) - 2 * q(0) - 2 * p(0)) - 2 * y * p(0) + 2 * q(0) + 4 * p(0)) \\ 0 = u * v - 1 \\ 0 = x * y - 1 \end{cases}$$

Step 3: Elimination of loop counters and new variables
After eliminating the variables x, y, u, v by Gröbner bases computation, the set F of polynomial relations :

$$F = \{q^2 - q(0)^2 + 2 * p * r - 2 * p(0) * r(0) = 0, q^2 - q(0)^2 + 2 * p * r - 2 * p(0) * r(0) = 0\}$$

Step 4: Polynomial invariant for the outer loop

$$G = \{q^2 - q(0)^2 + 2 * p * r - 2 * p(0) * r(0) = 0\}$$

that is, by initial values substitution:

$$G = \{a - 2 * p * r = q^2\}$$

5 Conclusions

We have tested our algorithm on a number of interesting number theoretic examples that needed polynomial invariants in order to be verified [22]. In all cases, our method implemented in *Mathematica*, by combining advanced symbolic summation techniques with computer algebra, succeeded with the generation of all polynomial invariants.

Note that the generated invariant equalities do not depend on the pre- and/or postconditions of the program. However, currently we investigate the possibility for invariant inequality generation by using quantifier elimination together with manipulation of pre- and postconditions.

Moreover, we are also interested in generalizing the framework to programs on nonnumeric data structures.

Acknowledgement. The author wishes to thank prof. Deepak Kapur (University of New Mexico), prof. T. Jebelean (RISC-Linz) and M. Kauers (RISC-Linz) for their help and comments.

References

1. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In *Proc. of SAS 1996*, volume 1102 of *LNCS*, pages 323–335, 1996.
2. N. Bjørner, A. Browne, and Z. Manna. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
3. B. Buchberger. Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985.
4. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

5. M. A. Colon, S. Sankaranarayanan, and H. B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Proc. of CAV 2003*, volume 2725 of *LNCS*, pages 420–432, 2003.
6. P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Proc. of VMCAI'05*, pages 1–24, 2005.
7. D. Cox, J. Little, and D. O'Shea. *Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2nd edition, 1998.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. B. Elspas, M. W. Green, K. N. Lewitt, and R. J. Waldinger. Research in Interactive Program - Proving Techniques. Technical report, Stanford Research Institute, 1972.
10. G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. AMS, 2003.
11. R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.
12. S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. In *IEEE Transactions on Software Engineering*, pages 68–75, 1975. 1(1):68-75.
13. R. W. Gosper. Decision Procedures for Indefinite Hypergeometric Summation. *Journal of Symbolic Computation*, 75:40–42, 1978.
14. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 2nd edition, 1989.
15. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.
16. D. Kapur. Automatically Genearting Loop Invariants using Quantifier Elimination. In *Proc. of ACA*, 2004.
17. M. Karr. Affine Relationships Among Variables of Programs. *Acta Informatica*, 6:133–151, 1976.
18. M. Kauers. SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *Journal of Symbolic Computation*, 41:1039–1057, 2006.
19. M. Kauers and B. Zimmermann. Computing the Algebraic Relations of C-finite Sequences and Multisequences. *Journal of Symbolic Computation*, 2007. In press.
20. L. Kovacs and T. Jebelean. An Algorithm for Automated Generation of Invariants for Loops with Conditionals. *IEEE Computer Society*, pages 245–249, 2005. Proc. of SYNASC'05.
21. L. Kovacs and T. Jebelean. Finding Polynomial Invariants for Imperative Loops in the Theorema System. In *Proc. of Verify'06, IJCAR'06*, pages 52–67, 2006.
22. L. Kovacs, T. Jebelean, and D. Kapur. Using Symbolic Summation and Polynomial Algebra for Imperative Program Verification in Theorema. *Mathematics and Computers in Simulation (MATCOM)*, 2007. in press.
23. L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Proc. of ISOLA 2006*, 2006.
24. M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*, 2006.
25. M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *Proc. of SAS 2002*, volume 2477 of *LNCS*, 2002. pp. 4-19.
26. P. Paule and M. Schorn. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *Journal of Symbolic Computation*, 20(5-6):673–698, 1995.
27. E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC 04*, 2004.

28. E. Rodriguez-Carbonell and D. Kapur. Generating All Polynomial Invariants in Simple Loops. *J. of Symbolic Computation*, 42(4):443–476, 2007.
29. S. Sankaranaryanan, H. B. Sipma, and Z. Manna. Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL 2004*, 2004.
30. A. Tiwari, H. Ruess, H. Saidi, and N. Shankar. A Technique for Invariant Generation. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 113–127, 2001.
31. D. Zeilberger. A Holonomic System Approach to Special Functions. *Journal of Computational and Applied Mathematics*, 32:321–368, 1990.

Using Widenings to Infer Loop Invariants Inside an SMT Solver, or: A Theorem Prover as Abstract Domain

K. Rustan M. Leino and Francesco Logozzo

Microsoft Research, Redmond, WA, USA
{leino,logozzo}@microsoft.com

Abstract. This paper presents a new technique for combining the inference power of abstract interpretation with the precision and flexibility of an automatic satisfiability-modulo-theories theorem prover.

0 Introduction

Reasoning about the correctness of programs is hard. The hard parts include a program's use of loops and recursion, whose correctness relies on invariants. In this paper, we present a technique that combines the automatic discovery of invariants and the precise reasoning about programs using those invariants. More precisely, we build in a widening operator into a satisfiability-modulo-theories (SMT) solver, which allows the SMT solver to apply the widening operator when the verification process discovers the need to further approximate loop invariants. An important consequence of this design is that it lets the SMT solver and the abstract-interpretation machinery that typically surrounds the widening operator to share in a common implementation. We also show how the basic technique can be extended to infer invariants in a goal-directed way.

To put this into more context, let us consider some common approaches to program verification.

0.0 Some Approaches to Program Verification

One approach to program verification is *verification-condition generation*. It encodes a program's proof obligations as logical formulas, *verification conditions* (VCs), which are then passed to a mechanical theorem prover. If the prover can establish the validity of the formulas, then the program has been shown to be correct. For many verification tasks, an SMT solver makes a good choice as the theorem prover, because it is automatic (*i.e.*, it works without user interaction), supports common theories (*e.g.*, arithmetic, functions and equality, plus support of quantifiers), and, in the event that it cannot establish the validity of the formula, outputs counterexamples that can be turned into error messages about the program (*cf.* [10]). In its classical form, verification-condition generation assumes the program to contain assertional specifications, in particular preconditions, postconditions, and loop invariants.

Another approach to program verification is *abstract interpretation* [5], a general static-analysis technique that can infer invariants of a program and be applied to verify properties of the program. In the classical setting, verification through abstract interpretation works in two phases. In the first phase, the program is analyzed to infer invariants. In the second phase, those invariants are propagated to all program points and it is

checked that the proof obligations at each program point are never violated. Analogous to the way an SMT solver uses various theories, an abstract interpreter uses *abstract domains*, each of which supports a *widening operator* that abstracts over details in order to compute invariants as post-fix points in the abstract-domain lattice.

Advantages of the VC generation approach include the precision offered by the theorem prover, and the beautiful Nelson-Oppen way of combining decision procedures for theories [19]. Another advantage is the flexibility offered by quantifiers when encoding verification conditions: these let a verification tool introduce function symbols and give a (possibly partial) axiomatization of them, without necessitating any changes to the theorem prover. A drawback of the approach is that it does not automatically infer the invariants that are needed to make the proofs go through.

Advantages of the abstract interpretation approach include a uniform view of inference and checking, and control over how to represent and approximate the properties of interest. Another advantage is that it presents a generalized setting for talking about static approximation (*i.e.*, the abstract domain) and dynamic approximation (*e.g.*, the widening operator). A drawback of the approach is that its implementation is not straightforward: it needs the design of an abstract domain, transfer functions that consider the effect of program statements on elements of the abstract domain, and convergence operators like widening. In addition, deciding on the direction of the analysis (forward, backward, or some combination thereof) impacts precision, and the forming of initial abstract-domain elements (a process called *seeding*) during an abstract interpretation usually has to be done in an *ad hoc* manner. Finally, there is no "user-programmable abstract domain"; for example, in the abstract interpretation literature, quantifiers have been considered just for restricted classes of problems, *e.g.*, [20, 8].

A hybrid approach to program verification combines the two approaches above, integrating an abstract interpreter with a theorem prover. For instance, one can apply the first phase of an abstract interpreter to infer loop invariants, instrument the program with the inferred invariants, and then continue as in the classical VC generation approach [1]. The major drawback of this hybrid approach is the "duplication of knowledge": despite the fact that the theorem prover and the abstract interpreter share many algorithms (for instance, the closure/decision procedures for linear arithmetic), most of the code and "knowledge" must be duplicated in (i) the abstract domains and the theorem prover theories; (ii) the transfer functions and the background axioms; and (iii) symbolic constants that may be used in the transfer functions and the single-assignment variables introduced during VC generation [7, 2]. A second drawback is that there is no feedback from the theorem prover to the abstract interpreter that would allow it to refine its inference.

The second drawback of the simple hybrid approach is addressed by our previous technique of "*loop invariants on demand*", a powerful integration between the theorem prover and the analyzer that focuses the refinement of loop invariants using counterexamples generated by the theorem prover [11]. Our previous technique works like this. First, we run a "cheap" static analysis on the program to infer simple loop invariants. Second, we instrument the program with those invariants and some special predicate symbols (loop predicates), which can seen as a kind of second-order predicate for loop invariants. Third, we generate verification conditions and pass them to an automatic

```
// assume M to be an array of length l and 0 ≤ p < l
M[p] := 0;
while (M[p] < 10) {
  M[p] := M[p] + 1;   (*)
}
assert M[p] = 10;
```

Fig. 0. Discovering the loop invariant $0 \leq M[p] \leq 10$ in a classical setting involves the use of the reduced product of an abstract domain for tracking the shape of the memory, and a numerical domain for tracking the value of the memory location $M[p]$. With our technique, we exploit the knowledge already inside the theorem prover to infer the right invariant.

theorem prover. If the theorem prover can complete the proof with this information, then we are done. Otherwise, the theorem prover returns a monome that represents a path in the program that (may) lead to an error. We analyze the monome, extract the loop predicates, project the monome onto the entry points of the loops, and run a more precise static analysis just on these loops. We then obtain more precise loop invariants, and pass these back to the theorem prover, iterating the process until a proof is found, no further refinement of the loop invariants is possible, or some bound on the number of iterations is reached.

The main advantage of "loop invariants on demand" is that it provides an effective way of combining backward and forward analyses of the code through a form of dynamic trace partition [16, 9]. However, it still does not solve the issue of "duplication of knowledge" and code, since the theorem prover and static analyzer are two separate entities. Furthermore, (i) the static analyzer is (still) unaware of quantifiers and (ii) there are many low-level details, for instance the different representations of variables and functions in the abstract interpreter and the theorem prover, that make the implementation tricky.

In this paper, we introduce a new technique for performing a goal-directed inference of loop invariants directly inside an SMT solver. The starting point is a VC encoding along the lines of "loop invariants on demand". Given an SMT solver, we then define a widening operator over first-order formulae. This affords us a uniform handling of variables, constants, and functions, but more importantly it factors out the "duplication of knowledge" issue. Furthermore, it exploits the theorem prover's precision in handling symbolic expressions and its flexible support of quantifiers. We illustrate the main ideas behind our technique with the next two examples. The examples show that we can make use of quantifiers, and that the goal-directed technique provides a generic framework for seeding.

0.1 Running Examples

Example 0. Our first running example is the program in Fig. 0. It shows a loop that updates an element of an array. The program involves the essence of a pointer program, where the array M represents memory and p represents a pointer into memory (like `*p` in C). Proving the correctness of the program involves the use of the loop invariant $0 \leq M[p] \leq 10$. Discovering such an invariant, in a classical setting, requires an abstract domain that is (at least) the combination of an alias analysis and a numerical

// assume A to be an array of length l
$s := 0;\ i := 0;$
while $(i < l)\ \{\ s := A[i] + s;\ \ i := i + 1;\ \ \}$
assert $s = \sum_{k=0}^{<l} A[k];$

Fig. 1. Proving the assertion involves discovering the loop invariant $0 \leq i \leq l \wedge s = \sum_{k=0}^{<i} A[k]$. While the first invariant can be discovered using standard numerical abstractions, the relation between s and the values in the array requires the design of an *ad hoc* abstract domain. Furthermore, the choice of the abstract domain must be fixed before running the analysis. Our technique leads to a dynamic and postcondition-driven choice of the abstract domain.

analysis, because (i) it must capture the shape of the memory, to infer that p always denotes the same index (*i.e.*, same memory location of) M, and that only the location $M[p]$ is modified, whereas all the others remains unchanged; and (ii) it must correctly approximate the integer value stored at location $M[p]$. Examples of such domains are in [15, 4].

Standard VC-generation handling of arrays takes care of the aliasing issues: assignment (*) in Fig. 0 is treated as $M := \mathsf{store}(M, p, M[p] + 1)$, the term $M[p]$ is represented by some equivalence class, say β, in the SMT solver's theory of functions and equality, and the array property

$$(\forall\, m, i, j, v \bullet\ i = j\ \Rightarrow\ \mathsf{store}(m, i, v)[j] = v\ ) \tag{0}$$

gets used to figure out the change of the value of $M[p]$. With the technique of this paper, incorporating a widening operator into the theory of arithmetic then performs the necessary inference. For example, using the theory of linear inequalities enriched with a widening with thresholds $\nabla_{threshold}$, [3], we can infer, after two iterations of the loop, the invariant $\{\beta = 1\}\nabla_{threshold}\{\beta = 0\} = \{0 \leq \beta \leq 10\}$, which the theorem prover uses to prove the program correct. □

Example 1. Our second running example is shown in Fig. 1. We want to prove that the assertion after the loop always holds. This is tricky, because we have to infer the loop invariant $s = \mathsf{sum}(A, 0, i)$. $\mathsf{sum}(A, 0, i)$ is a predicate that, intuitively, axiomatizes the sum of the first i elements of the array A.

One way of doing it is by providing an *ad hoc* abstract domain and the relative transfer functions. Let us call this domain ArrSum. The abstract elements in ArrSum are maps from variables to terms of the form $\mathsf{sum}(a, 0, n)$ (the sum of the first n elements of the array a). While the domain operations over ArrSum are straightforward, the transfer functions must be carefully designed and coded to handle, for instance, the assignments that involve array elements. However, ArrSum is quite a specialized construction, and even if implemented in an analyzer, it is not clear *when* the analyzer must use it for analyzing a piece of code. For example, in the code of Fig. 1, the need for keeping track of the sum of the elements of the array arises only when the assertion is hit, *i.e.*, after the loop. A forward static analysis alone may not figure it out until the loop exit is reached. For instance, when analyzing the code using Octagons, the static analyzer cannot prove the postcondition, so (i) it must recognize that the problem is because Octagons cannot express such a property; (ii) it must choose the right abstract

domain, *i.e.*, ArrSum; and (iii) it must analyze the method from scratch using ArrSum. This wastes both time and precision, since there is no communication between the first and second analyses.

With the technique in this paper, we do not need to hard-code the transfer functions or specify *a priori* the abstract domain. In a sense, the technique is able to dynamically choose the right domain and transfer functions by means of quantifier instantiation. For instance, the following partial axiomatization of the sum of the array:

$$(\forall\, a, l, h \bullet\ h \leq l \ \Rightarrow\ \mathsf{sum}(a, l, h) = 0\ ) \tag{1}$$
$$(\forall\, a, l, h \bullet\ l \leq h \ \Rightarrow\ \mathsf{sum}(a, l, h+1) = \mathsf{sum}(a, l, h) + a[h]\ ) \tag{2}$$

in combination with our technique is enough to infer the right loop invariant and hence to prove the program correct [12]. Roughly, the theorem prover first attempts to prove the assertion without looking inside the body of the loop. It fails, but its state now contains (i) the postcondition involving $\mathsf{sum}(A, 0, n)$; and (ii) a predicate encoding the need for a stronger loop invariant. We apply the widening as described later in the paper which causes the instantiation of the two quantifiers above, in particular obtaining the loop invariant $s = \mathsf{sum}(A, 0, i)$, which is enough for the theorem prover to complete the proof. □

1 Preliminaries

We describe our technique for a simple imperative language. We prescribe the generation of verification conditions for this language using the standard approach of translating it into an intermediate language and then applying weakest preconditions [13, 1]. We adapt this process to instrument the verification conditions with information that will be used by our invariant-generation machinery.

1.0 Source Language

We consider the source language whose grammar is given in Fig. 2. The source language includes support for specifications via the $\mathbf{assert}\ E$ statement: if the expression E evaluates to false, then the program fails. The assignment statement $E_0 := E_1$ sets the memory location represented by E_0 to the value of the expression E_1. The statement $\mathbf{havoc}\ x$ non-deterministically assigns a value to x. Sequential composition, conditionals, and loops are the usual ones. Note that we assume loops to be uniquely determined by labels ℓ taken from a set $\mathcal{W}$: Given a label ℓ, the function $\mathsf{LookupWhile}(\ell)$ returns the while loop associated with that label.

1.1 Single-Assignment Form

We translate the source language of Fig. 2 into an intermediate form so as to get rid of (i) loops and (ii) assignments. To achieve (i), the translation replaces an arbitrary number of iterations of a loop with something that describes the effect that these iterations have on the variables, namely the loop invariant. To achieve (ii), the translation uses a variant of static single assignment (SSA) [0,7], introducing new variables (*inflections*)

$$
\begin{array}{lll}
\mathit{Stmt} ::= & \mathbf{assert}\ \mathit{Expr}; & \text{(assertion)} \\
\mid & \mathit{Expr} := \mathit{Expr}; & \text{(assignment)} \\
\mid & \mathbf{havoc}\ x; & \text{(set } x \text{ to an arbitrary value)} \\
\mid & \mathit{Stmt}\ \mathit{Stmt} & \text{(sequential composition)} \\
\mid & \mathbf{if}\ (\mathit{Expr})\ \{\mathit{Stmt}\}\ \mathbf{else}\ \{\mathit{Stmt}\} & \text{(conditional)} \\
\mid & \mathbf{while}^{\ell}\ (\mathit{Expr})\ \{\mathit{Stmt}\} & \text{(loop)}
\end{array}
$$

Fig. 2. The grammar of the source language.

$$
\begin{array}{lll}
\mathit{Cmd} ::= & \mathbf{assert}^{L}\mathit{Expr} & \text{(assert)} \\
\mid & \mathbf{assume}\ \mathit{Expr} & \text{(assume)} \\
\mid & \mathit{Cmd}\ ;\ \mathit{Cmd} & \text{(sequence)} \\
\mid & \mathit{Cmd}\ \Box\ \mathit{Cmd} & \text{(non-deterministic choice)}
\end{array}
\qquad L ::= \mathit{init} \mid \mathit{rec} \mid \epsilon
$$

Fig. 3. The intermediate language. We label assertions with strings so as to distinguish between assertions to be checked at the entry point of a loop ($\mathbf{assert}^{init}$), after each loop iteration ($\mathbf{assert}^{rec}$), and user-supplied assertions from the source language ($\mathbf{assert}$, ϵ being the empty string).

that stand for the values of program variables at different source-program locations and that within any one execution path has only one value.

The statements of the intermediate language are given by the grammar in Fig. 3. Our intermediate language is that of passive commands, *i.e.*, assignment-free and loop-free statements [7].

The $\mathbf{assert}$ and $\mathbf{assume}$ statements first evaluate the expression Expr. If it evaluates to true, then the execution continues. If the expression evaluates to false, the $\mathbf{assert}$ statement causes the program to fail (the program *goes wrong*) and the $\mathbf{assume}$ statement causes the program to *block* (which implies that there is no possibility of the program subsequently going wrong). The intermediate language also supports sequential composition and non-deterministic choice.

The translation from a source-language program S to an intermediate-language program is given by the following function (id denotes the identity map):

$$\mathsf{translate}(S) \ = \ \mathbf{let}\ (\mathrm{C}, m) := \mathsf{tr}(S, id)\ \mathbf{in}\ \mathrm{C}$$

The verification condition is constructed by taking the weakest precondition (see Fig. 4) of the intermediate-language program, and adding an antecedent Q that contains the program's precondition and various background axioms (such as (0), (1), and (2)):

$$Q \ \Rightarrow\ \mathsf{wp}(\mathsf{translate}(S),\ \mathit{true})$$

The definition of the function tr is in Fig. 5. The function takes as input a program in the source language and a renaming function from program variables to their pre-state inflections, and it returns a program in the intermediate language and a renaming function from program variables to their post-state inflections. The rules in Fig. 5 are described as follows.

The translation of an $\mathbf{assert}$ just renames the variables in the asserted expression to their current inflections. One of the goals of the passive form is to get rid of the assignments. Given an assignment $x := \mathrm{E}$ in the source language, we generate a fresh

$$\mathsf{wp}(\mathbf{assert}^L E,\ \phi) \ =\ E \wedge \phi \qquad \mathsf{wp}(C_0\ ;\ C_1,\ \phi) \ =\ \mathsf{wp}(C_0,\ \mathsf{wp}(C_1,\ \phi))$$
$$\mathsf{wp}(\mathbf{assume}\ E,\ \phi) \ =\ E \Rightarrow \phi \qquad \mathsf{wp}(C_0 \,\square\, C_1,\ \phi) \ =\ \mathsf{wp}(C_0,\ \phi) \wedge \mathsf{wp}(C_1,\ \phi)$$

Fig. 4. Weakest preconditions of intermediate-language statements.

variable for x (intuitively, the value of x after the assignment), apply the renaming function to E, and output an **assume** statement that binds the new variable to the renamed expression. For instance, a statement that assigns to y in a state where the current inflection of y is y_0 is translated as follows, where y_1 is a fresh variable that denotes the inflection of y after the statement:

$$\mathsf{tr}(y := y + 4,\ [y \mapsto y_0]) \ =\ (\mathbf{assume}\ y_1 = y_0 + 4,\ [y \mapsto y_1])$$

An array assignment $a[\mathrm{E}_0] := \mathrm{E}_1$ is, as usual, treated as a shorthand for $a := \mathsf{store}(a, \mathrm{E}_0, \mathrm{E}_1)$. The translation of **havoc** x just binds x to a fresh variable, without introducing any assumptions about the value of this fresh variable. The translation of sequential composition yields the composition of the translated statements and the post-renaming of the second statement. The translations of the conditional and the loop are trickier.

For the conditional, we translate the two branches to obtain two translated statements and two renaming functions. Then we consider the set of all the variables on which the renaming functions disagree (intuitively, they are the variables modified in one or both the branches of the conditional), and we assign them fresh names. These names will be the inflections of the variables after the conditional statement. Then, we generate the translation of the true (resp. false) branch of the conditional by assuming the guard (resp. the negation of the guard), followed by the respective translated statement and an **assume** statement that equates the fresh variables with the inflections at the end of the true (resp. false) branch. Finally, we use the non-deterministic choice operator to complete the translation of the whole conditional statement.

For the loop, we first identify the assignment targets of the loop body (defined in Fig. 6), generate fresh names for them, and translate the loop body. Then, we generate a fresh predicate symbol indexed by the loop identifier (J_ℓ), which intuitively stands for the invariant of the loop $\mathsf{LookupWhile}(\ell)$. We also generate a fresh predicate symbol F_ℓ, which stands for the continuation of the loop; we will make this point clearer in Section 3. If the label ℓ is clear from context, as it is in our single-loop examples, we omit it. We output a sequence that is made up by: (i) an assertion that the loop invariant holds at the loop entry point; (ii) an assumption that the loop invariant holds after an arbitrary number of executions of the loop body (intuition: we have performed an arbitrary number of loop iterations and the renaming function n gives the current inflections of the variables); and (iii) a non-deterministic choice between two cases: (a) the loop condition evaluates to $true$, we execute a further iteration of the body, we check that the loop invariant holds ($\mathbf{assert}\ J_\ell\langle\mathsf{range}(n_C)\rangle$), and we stop checking ($\mathbf{assume}\ \mathit{false}$); or (b) the loop condition evaluates to false and the loop execution terminates.

Please note that we tacitly assume a total order on variables, so that, *e.g.*, the sets $\mathsf{range}(m)$ and $\mathsf{range}(n)$ can be isomorphically represented as lists of variables. We will use the list representation in our examples.

$$\mathsf{tr} \in \mathit{Stmt} \times (\mathtt{Vars} \to \mathtt{Vars}) \to \mathit{Cmd} \times (\mathtt{Vars} \to \mathtt{Vars})$$

$\mathsf{tr}(\mathbf{assert}\ \mathrm{E};,\ m) = (\mathbf{assert}\ m(\mathrm{E}),\ m)$
$\mathsf{tr}(\mathtt{x} := \mathrm{E};,\ m) = (\mathbf{assume}\ \mathtt{x}' = m(\mathrm{E}),\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathtt{a}[\mathrm{E}_0] := \mathrm{E}_1;,\ m) = (\mathbf{assume}\ \mathtt{a}' = \mathsf{store}(a, m(\mathrm{E}_0), m(\mathrm{E}_1)),\ m[\mathtt{a} \mapsto \mathtt{a}'])$
where $\mathtt{a}'$ is a fresh variable
$\mathsf{tr}(\mathbf{havoc}\ \mathtt{x};,\ m) = (\mathbf{assume}\ \mathit{true},\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathrm{S}_0\ \mathrm{S}_1,\ m) =$ **let** $(\mathrm{C}_0, n_0) := \mathsf{tr}(\mathrm{S}_0, m)$ **in**
let $(\mathrm{C}_1, n_1) := \mathsf{tr}(\mathrm{S}_1, n_0)$ **in**
$(\mathrm{C}_0\ ;\ \mathrm{C}_1,\ n_1)$
$\mathsf{tr}(\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\},\ m) =$
let $(\mathrm{C}_0, n_0) := \mathsf{tr}(\mathrm{S}_0, m)$ **in**
let $(\mathrm{C}_1, n_1) := \mathsf{tr}(\mathrm{S}_1, m)$ **in**
let $\mathtt{V} := \{\mathtt{x} \in \mathtt{Vars} \mid n_0(\mathtt{x}) \neq n_1(\mathtt{x})\}$ **in**
let $\mathtt{V}'$ **be** fresh variables for the variables in $\mathtt{V}$ **in**
let $\mathrm{D}_0 :=$ **assume** $m(\mathrm{E})$; C_0 ; **assume** $\mathtt{V}' = n_0(\mathtt{V})$ **in**
let $\mathrm{D}_1 :=$ **assume** $\neg m(\mathrm{E})$; C_1 ; **assume** $\mathtt{V}' = n_1(\mathtt{V})$ **in**
$(\mathrm{D}_0 \,\square\, \mathrm{D}_1,\ m[\mathtt{V} \mapsto \mathtt{V}'])$
$\mathsf{tr}(\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\},\ m) =$
let $\mathtt{V} := \mathsf{targets}(\mathrm{S})$ **in**
let $\mathtt{V}'$ **be** fresh variables for the variables in $\mathtt{V}$ **in**
let $n := m[\mathtt{V} \mapsto \mathtt{V}']$ **in**
let $(\mathrm{C}, n_C) := \mathsf{tr}(\mathrm{S}, n)$ **in**
let J_ℓ **be** a fresh predicate symbol **in**
let F_ℓ **be** a fresh predicate symbol **in**
$(\mathbf{assert}^{\mathit{init}}\ J_\ell\langle\mathsf{range}(m)\rangle$;
assume $J_\ell\langle\mathsf{range}(n)\rangle \wedge F_\ell$;
(**assume** $n(\mathrm{E})$; C ; $\mathbf{assert}^{\mathit{rec}}\ J_\ell\langle\mathsf{range}(n_C)\rangle$; **assume** false
$\square$ **assume** $\neg n(\mathrm{E})$), $n)$

Fig. 5. The function that translates from the source program to our intermediate language.

For example, applying translate to the program in Fig. 0 results in the intermediate-language program shown in Fig. 7.

2 Widening in an SMT Solver

To prove the program in Fig. 0 correct, we use an automatic theorem prover, and more precisely a *satisfiability modulo theories* (SMT) solver. We transform the program into passive form and generate the weakest preconditions, as described in the previous section, and then pass the resulting formula to the theorem prover. In order to prove the assertion after the loop, we need to provide a loop invariant, *i.e.*, a value for the predicate J_ℓ. Our goal is to have the SMT solver automatically find a sound yet precise approximation for J_ℓ. For this aim, we need to (i) make explicit the assumptions about the theorem prover; (ii) define a widening operator inside the theorem prover; and (iii) define a strategy for the application of the widening operator.

$$\begin{array}{l}
\mathsf{targets} \in \mathit{Stmt} \to \mathsf{P}(\mathtt{Vars}) \\
\mathsf{targets}(\mathbf{assert}\ \mathrm{E};) \ = \ \emptyset \\
\mathsf{targets}(\mathtt{x} := \mathrm{E};) \ = \ \mathsf{targets}(\mathtt{x}[\mathrm{E}_0] := \mathrm{E}_1;) \ = \ \mathsf{targets}(\mathbf{havoc}\ \mathtt{x};) \ = \ \{\mathtt{x}\} \\
\mathsf{targets}(\mathrm{S}_0\ \mathrm{S}_1) \ = \ \mathsf{targets}(\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\}) \ = \ \mathsf{targets}(\mathrm{S}_0) \cup \mathsf{targets}(\mathrm{S}_1) \\
\mathsf{targets}(\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}) \ = \ \mathsf{targets}(\mathrm{S})
\end{array}$$

Fig. 6. The assignment targets, that is, the set of variables assigned in a source statement.

$$\begin{array}{l}
\mathbf{assume}\ M_0[p] = 0\ ; \ \ \mathbf{assert}^{init}\ J\langle M_0\rangle\ ; \ \ \mathbf{assume}\ J\langle M_1\rangle \wedge F\ ; \\
(\ \mathbf{assume}\ M_1[p] + 1 \leq 10\ ; \ \ \mathbf{assume}\ M_2 = \mathsf{store}(M_1, p, M_1[p] + 1)\ ; \\
\quad \mathbf{assert}^{rec}\ J\langle M_2\rangle\ ; \ \ \mathbf{assume}\ \mathbf{false}\ ; \\
\square \\
\quad \mathbf{assume}\ 10 \leq M_1[p]\ ; \\
)\ ; \\
\mathbf{assert}\ M_1[p] = 10\ ;
\end{array}$$

Fig. 7. The intermediate-language program obtained as a translation of the source-language program in Fig. 0. J is a predicate symbol corresponding to the loop invariant, F is a predicate standing for the semantics of the program bottom up from the exit point up to the loop entry point.

2.0 The SMT Solver

An SMT solver decides validity, w.r.t. to some set of theories, of first-order formulae with equalities. We assume an SMT solver to be a triple $\langle E, \{\, T_i \mid 0 \leq i\}, \vdash\rangle$, where E is a system for tracking equivalence relations (*e.g.*, an e-graph [18]), T_i is a set of first-order theories (*e.g.*, linear arithmetic or the fact-generating "theory" of quantifiers) and $\vdash$ is the decision procedure.

2.1 The Widening

A widening is an operator for extrapolating the limit of a (possibly infinite) sequence. The simplest (and least precise) widening is the one that extrapolates the sequence with the greatest element of the abstract domain. Stated differently, it is the one that answers "I do not know". For an abstract domain whose elements can be viewed as a set of constraints, a traditional schema for defining a widening (i) considers two successive sets of constraints in the sequence, say C_n and C_{n+1}; and (ii) keeps those of the constraints in C_n that are "stable", that is, that also hold in C_{n+1}. In other words, in its computation of the limit, this schema looks at consecutive elements of the sequence and removes from the limit any constraints that are not stable. For instance, the traditional definition of the widenings on the numerical domains Polyhedra [6], Octagons [17], and Intervals [5] are instances of this schema, [14]. We seek the definition of such a kind of widening inside an SMT solver.

First, we require that all the theories be extended with a widening specific to the theory. We write T_i^{∇} to denote the extension of a first-order theory T_i that uses widenings. The extension is always possible because, as mentioned earlier, one can always use the "I do not know" widening. However, one may consider using more precise widenings for certain theories.

$$\Sigma \nabla \Upsilon = \textbf{for each } c \in \Sigma \textbf{ do}$$
$$\qquad \textbf{if } \Upsilon \vdash c \textbf{ then keep } c \textbf{ else keep } \bigwedge_{0 \le i} c \nabla_i (\Upsilon \downarrow i)$$

Fig. 8. The widening over first-order formulae. $\Upsilon \downarrow i$ denotes the projection of the formulae in Υ over the language understood by the theory T_i^∇. ∇_i is the widening of the theory T_i^∇.

for each $\ell \in \mathcal{W}$ **do** $J_\ell :=$ **false** ;
repeat
  Try to prove the formula ;
    0. if error on $\textbf{assert}^{init}\ J_\ell\langle \mathbf{x_0} \rangle$ then $J_\ell := J_\ell \vee \textbf{rename}(\pi_{\mathbf{x_0}}(\Sigma), \mathbf{x_0}, \mathbf{x})$;
    1. if error on $\textbf{assert}^{rec}\ J_\ell\langle \mathbf{x_2} \rangle$ then $J_\ell := J_\ell \nabla \textbf{rename}(\pi_{\mathbf{x_2}}(\Sigma), \mathbf{x_2}, \mathbf{x})$;
    2. if error on $\textbf{assert}\ P$ then report "**error**" and exit ;
    3. if no error then report "**correct**" and exit ;

Fig. 9. The driver algorithm for the SMT solver. Σ denotes the state of the theorem prover, $:=$ denotes destructive assignment, π denotes projection, the list $\mathbf{x_0}$ stands for the inflections of the (list of) variables $\mathbf{x}$ at the entry point of loop ℓ, and the list $\mathbf{x_2}$ stands for the inflections of the (list of) variables $\mathbf{x}$ at the end of the body of loop ℓ.

Example 2. Consider the theory of linear inequalities and suppose the implementation uses a list of linear constraints. The widening of two sets of linear constraints, say ∇_{la}, can be implemented by converting the constraints into the dual representation as vertices and generators, and then applying the original Cousot-Halbwachs widening on polyhedra [6]. Alternatively, since the conversion into the dual form can be expensive, cheaper solutions can be used, *e.g.*, projecting the constraints onto octagons and then applying the widening of this domain [17]. □

Second, we define the widening inside the SMT solver as in Fig. 8. Let Σ and Υ be two sets of formulae. Intuitively Σ is the set of formulae (whose conjunction represents) "the current approximation of the loop invariant" and Υ is the set of formulae (whose conjunction represents) "after one more iteration". If a formula $c \in \Sigma$ holds in Υ, then it is stable over loop iterations, so we keep it. Otherwise, for each theory T_i^∇, we apply the widening of c and (the projection over the) theory of Υ.

2.2 The Fixpoint Computation Strategy

Having defined a widening operator, we now define how to apply it. Stated differently, we need an algorithm that drives the theorem prover to the application of the widening, so as to compute fixpoints. The driver is defined in Fig. 9. For now, we assume that the solver reaches case 2 only if there is no other path in the formula for which cases 0 and 1 apply; we will remove this assumption in the next section.

In essence, the algorithm invokes the SMT solver using successively weaker definitions of each inferred loop invariant. If the SMT solver complains that it cannot prove the current approximation of a loop invariant to hold, the algorithm weakens the loop invariant and reruns the SMT solver. The algorithm assigns to the loop predicate J_ℓ its current invariant for loop ℓ; to represent the program variables in this formula, we will use the pre-inflected forms of the program variables.

In more detail, the algorithm initializes all the loop predicates to $\mathbf{false}$. In other words, the fixpoint computation starts from bottom. Then we ask the SMT solver to prove the formula. We consider a case for each of the four different outcomes.

Outcome 0: the solver complains about the "$init$" assertion of a loop ℓ. Intuitively, the current loop invariant is too strong to accommodate all paths that lead to the entry of the loop. So we weaken the loop invariant J_ℓ to include the prover's knowledge at the entry point of the loop ℓ. This is done by (i) projecting the state Σ of the prover onto the inflections of the variables $\mathbf{x}$ at the entry point: $\pi_{\mathbf{x_0}}(\Sigma)$; (ii) changing the variables back to their pre-inflected form, that is, renaming the $\mathbf{x_0}$ back to $\mathbf{x}$; and (iii) joining the result to the previous J_ℓ. Note and recall, the free variables of the predicate J_ℓ are written in the pre-inflected form $\mathbf{x}$.

Outcome 1: the solver complains about the "rec" assertion of a loop ℓ. This means that the prover is unable to prove that the current loop-invariant approximation is maintain by the loop body. Here, we do not use the logical disjunction as in case 0, because that would not guarantee termination of our analysis. Instead, we use the widening operator ∇ of Fig. 8.

Outcome 2: the solver complains about an assertion that originates from the source program. Since the iterations of our algorithm only weakens the approximated loop invariants, further iterations of the algorithm would never mask this failing assertion. Thus, we report an error and exit. Note, however, that the condition may or may not be an error, because of imprecision in the theorem prover's decision procedures (for example, for quantifiers) or because of precision lost in widening operations.

Outcome 3: the solver finds the current formula to be valid. Then the program is correct (since our analysis and the SMT solver are sound), and we can report it to the user and exit.

A note: For outcome 2, it would be nice if the theorem prover were as specific as possible about the counterexample it has found, since this will allow a more concrete error message. But for outcomes 0 and 1, we are interested in as general a counterexample as possible, since that will reduce the number of iterations of our algorithm.

2.3 Application to the First Example

We briefly sketch how our technique works on the code in Fig. 0. First, the program is translated into the single-assignment form in Fig. 7. Second, we generate the weakest preconditions from the passive program and add formula (0) as an antecedent; call the result ϕ. Third, we feed ϕ to the prover. For the sake of simplicity, herein we reason on the passive program and not on the formula ϕ, the two representations being isomorphic for our purposes. We apply the driver in Fig. 9. We set J_ℓ to $\mathbf{false}$ and run the prover. It returns, complaining about the assertion $\mathbf{assert}^{init}\ J\langle M_0\rangle$. Case 0 applies, and $\pi_{\mathbf{M_0}}(\Sigma)$ yields $M_0[p] = 0$, so we update J to be the predicate $M[p] = 0$. We run the prover again, and this time it complains about the recursive assertion $\mathbf{assert}^{rec}\ J\langle M_2\rangle$. Case 1 applies, so we instantiate the widening operator in Fig. 8. The state of the theorem prover up to this point includes the constraints

$$\{\ \text{formula (0)},\ 0 \le M_1[p],\ M_1[p] + 1 \le 10,\ M_2 = \mathsf{store}(M_1, p, M_1[p] + 1)\ \}$$

Projecting this state onto M_2 yields $1 \leq M_2[p] \leq 10$, because by instantiating the quantifier (0), it follows that $M_2[p] = \mathsf{store}(M_1, p, M_1[p] + 1)[p] = M_1[p] + 1$. From renaming, we get $1 \leq M[p] \leq 10$. The widening is then $\{M[p] = 0\} \nabla_{la} \{1 \leq M[p] \leq 10\}$. $M[p]$ is represented in the e-graph of the theorem prover with some symbolic name β. Thus, if we assume that the widening of linear arithmetic, ∇_{la} is with thresholds, we have

$$\{\beta = 0\} \; \nabla_{la} \; \{1 \leq \beta \leq 10\} \; = \; \{0 \leq \beta \leq 10\}$$

Consequently, the predicate J is updated to $0 \leq M[p] \leq 10$, the prover is run again, and this time it can prove the formula to be valid, so that it reaches case 3 and exits.

3 Goal-Directed Seeding

In the previous section, we saw how to exploit the symbolic capabilities and the quantifiers in theorem provers for inferring non-trivial loop invariants involving array updates, and hence memory manipulation. However, we want to go further, and in particular we want to be able to handle more properties, expressed as (partial) axiomatizations. Let us consider the example in Fig. 1. We would like to use the axioms (1) and (2) to get the correct loop invariant, so as to prove the user-supplied assertion correct. The technique of the previous section is not powerful enough to do so, essentially because the algorithm in Fig. 9 is a *forward* algorithm. It does not infer any invariant about s, since the need for an invariant that correlates s and sum will be discovered only *after* the loop. Hence, the theorem prover cannot prove the user assertion, so the algorithm reaches case 2, and thus ends up reporting a spurious warning.

To overcome this problem, we must be able to refine the information about loop invariants using information coming from the future. In other words, we need to tag entry points of loops with information about what will happen after the loop, so that, if needed, we can exploit this information to refine the loop invariant. We do it in two steps. First, we annotate loop assumptions with a predicate symbol F_ℓ, *i.e.*, a placeholder that we use to encode the execution of the program *after* the loop. Second, we refine case 2 of Fig. 9, so that now, when an assertion is violated, instead of immediately reporting an error and exiting, we (i) check the counterexample to see if it contains any F_ℓ, *i.e.*, witnesses of loops whose invariants can be refined with information coming from the continuation of the loop; (ii) try to refine the loop invariants J_ℓ for the affected loops; and (iii) give another change to the theorem prover with the new facts.

3.0 Enrich

We refine the case 2 of the driver as in Fig. 10. $\mathsf{Enrich}(\ell)$ is a procedure that rolls back the loop invariant inferred for the loop ℓ, and gives a new change to the fixpoint computation for inferring a new invariant, by having some additional knowledge coming from the "future".

The pseudo-code for Enrich is in Fig. 11. The first step of Enrich is to "un-negate" the failed assertion. If an $\mathbf{assert}\ P$ fails, then the monome μ contains the negation of P. Since P may be arbitrarily complex, in particular if it contains conjuncts (that

$2'$. if error on $\mathbf{assert}\ P$
  if want to try again
    $\mathbf{let}\ \mu\ \mathbf{be}$ the counterexample generated from the theorem prover
    Chose an $F_\ell \in \mu$
      $\mathsf{Enrich}(\mu, \ell)$
  else
    report "**error**" and exit ;

Fig. 10. The refinement of the case 2 that introduces a goal-directed inference.

$\mathsf{Enrich}(\mu, \ell) =$
  a. un-negate the failing assertion in μ. Call the result Σ ;
  b. roll back to a pre-widening value for J_ℓ ;
  c. $J_\ell := J_\ell \wedge (\mathbf{rename}(\pi_\Sigma(\mathbf{x_1}), \mathbf{x_1}, \mathbf{x}) \nabla J_\ell)$;

Fig. 11. The definition of function Enrich. The list $\mathbf{x_1}$ stands for the inflections of the (list of) variables $\mathbf{x}$ after an arbitrary number of loop iterations.

negated became disjuncts), "un-negating" it may be difficult. We believe it is still possible, however—for example, the theorem prover may be able to take a snapshot of its state before it attempts to prove an assertion.

The second step is to roll back the updates of J_ℓ to before we applied the widening. Intuitively, this means that we want to go back to the state just before analyzing the loop ℓ. Hence, we pick the last update J_ℓ before the application of case 1 in the algorithm above. This rolling back is always possible, provided that the implementation keeps the sequence of updates to J_ℓ (or the disjuncts added via applications of case 0).

The third step essentially pushes (backwards) the failed assertion P to the entry point of the loop, so that facts about "the future" become known during the analysis of the loop body, giving case 1 of the driver in Fig. 9 a chance to infer stronger invariants than in its previous attempts.

3.1 Application to the Second Example

Let us consider the code in Fig. 1. We omit showing the steps of single assignment, inflections, and weakest preconditions, in order to concentrate on how the driver algorithm works. We start the iterations from bottom, so at a first approximation we have $J = \mathbf{false}$. Then, we try to verify the code, the prover complains about the "$init$" assertion, so case 0 applies and we update J as follows:

$$\begin{aligned} J &:= \mathbf{false} \vee \{s = i = 0 \leq l\} \\ &= \{s = i = 0 \leq l\} \end{aligned}$$

We give the theorem prover another chance, and now it complains about the "rec" assertion. We proceed as above and we get the new approximation $J = \{0 \leq i \leq l\}$. Unfortunately, the widening lost all the information about s, so it is not possible to prove the assertion at the end of the program. The driver reaches the case $2'$, in a state μ containing $\{s_0 = i_0 = 0 \leq i_1 \leq l,\ l \leq i_1,\ s_1 \neq \mathsf{sum}(0, l, A),\ F\}$, where s_0 and i_0 are the inflections of s and i at the entry point of the loop and s_1 and i_1 are

the inflections of the same variables after an arbitrary number of iterations. Since μ contains the placeholder F, we apply the Enrich procedure to the (only) loop in the example.

Step a removes the negation of the assertion, and adds the assertion itself. The new state Σ contains:

$$\{\text{ axioms},\ s_0 = i_0 = 0 \leq i_1 \leq l,\ l \leq i_1,\ s_1 = \mathsf{sum}(0, l, a) = \mathsf{sum}(0, i_1, A),\ F\ \}$$

Note, that from the equalities $i_1 = l$ and $s_1 = \mathsf{sum}(0, l, a)$ and formula (1), the SMT solver can deduce the equality $s = \mathsf{sum}(0, i_1, A)$.

Step b rolls back J to the last state before the application of a widening:

$$J = \{s = i = 0 \leq l\}$$

This means that we, once again, have as much information as possible about s from the execution paths that reach the loop. Step c applies the widening to get the new value for J:

$$\begin{aligned} J &:= \{0 \leq i \leq l,\ i = l,\ s = \mathsf{sum}(0, l, A) = \mathsf{sum}(0, l, A)\} \nabla J \\ &= \{0 \leq i \leq l,\ s = \mathsf{sum}(0, i, A)\} \end{aligned}$$

which, together with the two axioms (1) and (2) let the theorem prover conclude the proof.

4 Conclusions

We have presented a technique that moves the inference of a program's loop invariants into the theorem prover that is attempting to prove the program correct. This has the benefits of making the inference more demand driven, reducing the duplication of work in the implementations of the abstract interpreter and theorem prover, and providing an easy way to extend abstract interpreters with information specified in quantifiers.

The work is still at an early stage. For example, we have not yet implemented the technique, we have not formally proved any soundness theorems, and we have not evaluated the complexity of our widening operator, which would benefit from an efficient way to enumerate the constraints $c \in \Sigma$ in the prover. Nevertheless, we are encouraged by the ease with which our second example goes through; in that example, the loop-invariant inference makes use of the partial axiomatization of sum that is specified by quantifiers (previously not incorporated in abstract interpreters), and it automatically seeds the abstract domain with an appropriate application of function sum (in contrast to the typical mode of requiring each abstract domain to design its own seeding). We hope that the discussion at the workshop will lead to greater insights.

Acknowledgments We are grateful to the referees for their careful readings and useful comments.

References

0. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th POPL*, pages 1–11. ACM, January 1988.
1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87. ACM, September 2005.
3. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207. ACM, June 2003.
4. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI '05*, volume 3385 of *LNCS*. Springer, January 2005.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252. ACM, January 1977.
6. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97. ACM, 1978.
7. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *28th POPL*, pages 193–205. ACM, January 2001.
8. Sumit Gulwani and Ashish Tiwari. Static analysis of heap-manipulating low-level software. Technical report, Microsoft Research, MSR-TR-2006-160, 2006.
9. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS '98*, volume 1503 of *LNCS*, pages 200–215. Springer, 1998.
10. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*. Springer, 2000.
11. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS '05*, volume 3780 of *LNCS*. Springer, November 2005.
12. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs. Manuscript KRML 175, May 2007. Submitted.
13. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *FTfJP 1999*, Technical Report 251. Fernuniversität Hagen, May 1999.
14. Francesco Logozzo. Approximating module semantics with constraints. In *SAC 2004*, pages 1490–1495. ACM, March 2004.
15. Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *VMCAI '07*, volume 4349 of *LNCS*. Springer, January 2007.
16. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP 2005*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.
17. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
18. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Technical Report CSL-81-10, Xerox PARC, June 1981.
19. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
20. Francesco Ranzato and Francesco Tapparo. An abstract interpretation perspective on linear vs. branching time. In *APLAS '05*, volume 3780 of *LNCS*. Springer, November 2005.

About one Algorithm of Program Polynomial Invariants Generation

Michael S. Lvov

Kherson State University, 40-Rokiv Zovtnja, 27, 73000, Kherson, Ukraine

Abstract. We consider the problem of polynomial invariants automatic generation for programs, data algebras of which are the domains of integrity or fields (programs, defined above fields). The set of invariants in optional control point of such program P forms polynomial ideal I_p. It is noted if in conditions of program statements it is used the equalities and disequalitites, the problem of construction of the basis of this ideal is algorithmically unsolvable. Thus the conditions in program statements are to be ignored, considering computations to be non-deterministic. The basic theorem of this work is the theorem of algorithmic solvability of membership problem $g \in I_p$ and construction of vector space bases of polynomial invariants of bounded degree for non-deterministic programs, defined above field. The proof per se uses only the Noetherian feature of rings of multivariable polynomials above field. Based upon this theorem it is laid down some algorithms of solution of stated problems, using the Gröbner bases.

Keywords: Programs statistic analysis, polynomial invariants, invariants automatic generation problem, Gröbner bases.

1 Introduction

Methods of programs static analysis at present is being studied very intensive. One of the important problems is the problem of program invariants automatic generation. Invariants of program are used particularly in methods of programs verification. We would like to note the existence and effectiveness of algorithms of program invariants generation severely depend on data domain, that is on data algebra features. The investigations of program invariants automatic generation problem for different data algebras have being performed since the seventies in Institute of Cybernetics of the National Academy of Sciences of Ukraine [1]. The main results are described in [2]. In present paper it is described the method of construction of polynomial invariants in programs, data algebra of which is the domain of integrity or the field [3]. In more details it is stated the methods and main problems of program invariants generation for indicated data algebras in [4].

The method of a solution of the problems considered in our work is based on Noetherian feature of rings of multivariable polynomials. The same idea was used by M. Müller-Olm, H. Seidl in [9] at a solution of the same problems for

polynomially defined programs. We shall note, that in this paper the program conditions of type $f(X) \neq 0$, where - $f(X)$ - polynomials from variables of the program are considered. The same authors [8] have offered a method of computation of polynomial program invariants in the linearly-defined (affine) programs containing recursive calls of procedures. In both papers it is used the graph model of the program.

S.Sankaranarayanan, H. Sipma, Z. Manna [7] have offered a method of computation of polynomial invariants of loops as polynomial forms [template polynomials] with the use of algorithm of computation of Gröbner bases. As model of the program they use transition systems.

E. Rodriguez-Carbonell, D. Kapur [10] have considered algebraic bases of search problem of polynomial invariants of loops. The basic result of this paper is the algorithm of computation of all polynomial invariants for loops with so-called solvable assignment statements. In particular, affine operators with positive real eigenvalues are solvable. The same authors in [11] have offered a method of generation of polynomial invariants of loops, including the enclosed loops, and also considering program conditions as in the form of polynomial equalities and disequalities. This method is realized in program system. In this paper a great number of examples is considered, and also the tables of an operating time of algorithm depending on technical parameters of analyzed programs are described.

L. Kovàcs, T. Jebelean [12] describe the algorithm of searching of invariants of loops which is realized in program system (Theorema System). The algorithm of this system is based on the several approaches described in paper . The execution of algorithm explicitly is described on examples.

2 Program Models

For problems of program invariants automatic generation in [1, 2] it is proposed the iterative methods of static analysis, called the methods of upper and lower approximation. As a program model the $U - Y$ program schemes above memory are used, being interpreted above data algebras. In this model the program is represented by initial oriented graph with marked final vertexes. The vertexes of this graph are called states but edges are called transitions. The states of program are its control points. Programs transitions are marked by conditions and computing statements (definitions are below). This model represents unstructured program.

In [3] the algorithms of proof and generation of polynomial invariant equalities of bounded degrees for considered class of programs are described. In [3] as a program model it is used the regular expressions in algorithmic algebra of Gluskov, which represents the structured programs. The difference of program models from [2] and [3] is not important, as for the transformation $U - Y$ program scheme in equivalent regular expression of algorithmic algebra of Glushkov can be used the algorithm of analysis of finite automata.

Let's give an example of the source text of the program in the form of procedure in language Pascal, and its representations in the form of regular expression in Glushkov's algebra.

Exmaple 1. The source program, and its representation in the form of regular expression.

```
Procedure GCD(a, b: Integer; var d: Integer);
begin
   While a <> b do
   If a < b then b := b-a else a := a-b;
   d := a
end;
```

Computing statements: y_1: b:= b-a, y_2: a := a-b, y_3: d := a, e: identical statement.

Conditions: u_1: a<>b, u_2: a<.b.

Program regular expression:

$$\underset{u_1}{\{}\ y_1 \underset{u_2}{\vee} y_2\ \}^* y_3.$$

The program system announced in [5], as the source text of the program uses its description in the form of procedure in language Pascal. It generates the dictionary of computing statements, the dictionary of conditions, lists of formal parameters and local variables, as well as regular expression of the program. We use the techniques of symbolic computations both for transformations of programs, and for the proof and generation of polynomial invariant equalities. Therefore we prefer to deal with algebraic models of programs. The regular expression of the program can be received from its $U - Y$ *scheme* by means of the analysis algorithm of graph being considered as a finite automata.

So, as a program model we will use the representation of program in a way of regular expression in algorithmic algebra of Glushkov. We would like to show the main operations of this algebra:

$$P^* Q,\ \underset{u}{P \vee Q}\ ,\ \underset{u}{\{}\ P\ \}\ ,\ \{\ P\ \underset{u}{\}} \tag{1}$$

Here P, Q - are programs, but u - conditions. The semantics of these operations can be defined with the help of statements of language Pascal:

$P^* Q \sim P; Q$ – *consecutive execution.*

$\underset{u}{P \vee Q} \sim$ If u then P else Q.

$\underset{u}{\{}\ P\ \} \sim$ While u do P

$\{\ P\ \underset{u}{\}}. \sim$ Repeat P until u

The atomic programs are so called computing statements (definition is below).

Thus, the semantics of program is defined by semantics of computing statements and conditions that is by program interpretation above some data domain.

3 Programs Interpreted Above Data Algebras

Let A be algebra with operations signature $\sum =< \sigma_1, \ldots, \sigma_k >$ and support A. Let further $X =< x_1, \ldots, x_n >$ be vector of variables, which we interpret as the variables of the program. For short we will call it X program memory, and vectors $a =< a_1, \ldots a_n >$, $a_i \in A$ – memory states. The set $U = A^n$ forms the space – universe of program memory states. Computing statement is a mapping $F : U \to U$, determined by the appropriation coordinate-wise.

$$x_1 := f_1(X), \ldots, x_n := f_n(X). \tag{2}$$

The computing statement we will write down in form of vector $X := F(X)$, and vector of coordinate functions (computations) – in form $F =< f_1, \ldots, f_n >$. We would like to note that in such indications the computing statement is interpreted as simultaneous assignment to the vector the variable values that is the right hand sides of coordinate -wise assignment statements.

Let's specify this definition for our case. We shall consider, that algebra A - either a domain of integrity, or a field. For example, it can be a ring of integers Z, a field of rational numbers Q, algebraic and transcendental extensions of field Q, finite fields. We shall consider, that all arithmetical operations above elements of a field A are fulfilled precisely. In program system [5] computations with rational numbers of unlimited length, for example, are used.

We will indicate that program P is defined linearly (it is interpreted above vector space), if A is some field (domain of integrity), but all its computing statements look like

$$X := F * X + b, \tag{3}$$

where F – linear operator, acting in vector space U, and $b \in U$.

We will indicate that program P is defined polynomially (it is interpreted above polynomial ring), if A – is some field (domain of integrity), but all its computing statements look like:

$$X := F(X), \tag{4}$$

where $f_i \in A[X]$, that is polynomials with coefficients from A.

We will indicate that program P is defined rationally (it is interpreted above the field of rational expressions), if A – is some field, but all its computing statements look like:

$$X := F(X). \tag{5}$$

where $f_i \in A(X)$, that is rational expression with coefficients from A.

The conditions are determined as predicates in some special conditions language L. In any case this language includes predicates of equality and disequality $(=, \neq)$.

Thus, the programs, being interpreted above linear spaces on each step of computations make linear or affine transformations of memory states universe U, programs, interpreted above polynomial rings – polynomial transformations U, but programs, being interpreted above the field of rational expressions – rational transformations U.

4 Program Invariants

Polynomial $g(X) \in A[X]$ is called polynomial invariant of the program P if

$$\Box \{\text{True}\} P \{g(X) = 0\}.$$

It means at any initial memory state $a =< a_1, \ldots a_n, >$ if the program P completes the execution, for final memory state b (that is such b that $a \{P\} b$ the equality is correct $g(b) = 0$.

Example 2. Program invariance equality.

```
Procedure ModDiv(a, b: Integer; var q, r: Integer);
begin
  r := b;
  q := 0;
  While r >= b do begin
  r := r - a;
  q := q + 1
  end
  {Invariance equality: a = b*q + r, Invariant: a - b*q - r}
end;
```

Thus, the polynomial invariance equalities look like $l(X) = r(X)$, where $l(X), r(X) \in A[X]$. Corresponding invariant looks like $l(X) - r(X)$.

Let's give algebraic definitions and statements necessary for further description.

Set $I \subseteq A[X]$ is called ideal of ring $A[X]$, if

1. for each elements $f, g \in I$ difference $f - g \in I$ and
2. for each elements $f \in I, g \in A[X]$ product $g * f \in I$

The basis of ideal $I \subseteq A[X]$ is called the finite set of polynomials $b_1(X), \ldots, b_k(X)$ of ring $A[X]$ such as that each polynomial $p \in I$ can be represented as

$p(X) = c_1(X) * b_1(X) + \ldots + c_x(X) * b_k(X)$, where $c_i(X) \in A[X]$.

The Hilbert's Theorem about the basis of ideal of polynomials commutative ring states, that each ideal $I \subseteq A[X]$ has finite basis. If $b_1, \ldots, b_k$ – is basis of ideal I, this fact is written as $I = (b_1, \ldots, b_k)$.

Let $I_0 \subset I_1 \subset \ldots \subset I_k \subset \ldots$ – be increasing sequence of ideals of a ring $A[X]$. In this case this sequence is finite.

This feature is called the Noetherian feature of ring $A[X]$. The Hilbert's Theorem about the basis and Noetherian feature of $A[X]$ – are the equivalent statements. The ring with this feature is called Noetherian ring.

The base of an ideal does not possess property of uniqueness. The Gröbner base of ideal $I \subseteq A[X]$ is a representation of base in the form of, possessing many remarkable algebraic and algorithmic properties, including property of uniqueness. Therefore the techniques of construction of Gröbner bases are widely used in problems of the constructive theory of polynomial ideals.

Ideal $I \subseteq A[X]$ is called prime, if from $p * q \in I$ follows $p \in I$ or $q \in I$.

Ideal $I \subseteq A[X]$ is called radical, if from $p^k \in I$ follows $p \in I$.

Radical ideal I can be represented in form of intersection of finite number of prime ideals: $I = J_1 \cap J_2 \cap \ldots \cap J_l$.

It is easy to show that the set of all polynomial invariants of rationally defined program P forms the radical ideal of the ring $A[X]$, which we will mark via I_p.

Full statement of a problem of generation of polynomial invariants consists in the following:

Let I_p – be an ideal of polynomial invariants of the program P. To present I_p in the form of intersection of prime ideals $I_p = J_1 \cap J_2 \cap \ldots \cap J_l$ and to construct Gröbner bases of ideals $J_1, J_2, \ldots, J_l$.

Unfortunately, this problem is difficult. We shall be limited with its partial solution.

Later we will describe also the set of all elements I_p, degrees of which are bounded by some constant M. The set of such polynomials forms vector space, to be marked via $I_p^{(M)}$. It is clear that $I_p^{(M)} \subset I_p$.

5 Theoretical Results

Theorem 1. *If $ch(A) = 0$ and language of conditions L includes predicates of equality and disequality $(=, \neq)$, the problem of construction of base I_p in class of linearly determined programs is algorithmically unsolvable.*

The proof is given in [5]. It is clear that from this theorem insolubility of examined problem for all classes of programs follows, described in the present work. Thus in further results the conditions in programs have to be ignored, considering the programs to be non-deterministic. The signature (1) of operations with programs at this is reduced to

$$P * Q, P \vee Q, \{P\}. \tag{6}$$

The main result [2] of the present work consists of the following:

Theorem 2. *Let P be a program, interpreted above the polynomial ring $A[X]$, I_p - ideal of invariants P and $I_p^{(M)} = \{g \,|g \in I_p, deg\ \ g \leq M\}$.*

Then:

a) Membership problem $g \in I_p$ is algorithmically solvable.

b) The problem of vector space base construction $I_p^{(M)}$ is algorithmically solvable.

Proof. Let's generate the dictionary of computing statements of the program P. For this purpose to each computing statement $X := F_i(X)$ of the program P we shall put in correspondence a symbol y_i. Thus, the dictionary of computing statements consists of pairs $(y_i, X := F_i(X))$ (see an example 1). Let $y_1, \ldots, y_m$ - be numerals of all computing statements of program P. Then the regular expression in the alphabet $Y = \langle y_i, \ldots, y_m \rangle$ represents program P.

a). The proof of the statement a) we shall conduct by induction on structure of regular expression P in the alphabet Y.

Let $w = y_{i_1} * y_{i_2} * \ldots * y_{i_k}$ - be a word in the alphabet Y. We shall speak, that $w \in P$, if w - is a word of the regular language, determined by regular expression P. (In other words, P supposes the computations described by a word w). Actually program P associates with set of words belonging to it. Each word P describes one of possible paths of computations in program P.

Let's introduce the following symbols. Let $w \in P$. Through $g(w), g \in A[X]$, we shall designate a polynomial $g\left(F_{i_k}\left(F_{i_{k-1}}\left(\ldots\left(F_{i_1}(X)\right)\ldots\right)\right)\right)$. Through $g(P)$ we shall designate set of polynomials of view $g(w), w \in P$. An ideal of a ring $A[X]$, generated by set of polynomials $g(P)$, we shall indicate through $(g(P))$.

We shall show, that for each program P it is exised and can be effectively constructed the finite subset of words $W = W(g, P) \subseteq P$ such, that $(g(P)) = (g(W))$. Thus, the set $g(W)$ forms finite basis of an ideal, generated by set $g(P)$. We shall notice, that g - an invariant of program P in only case when, $g(w) = 0$ for any $w \in P$. Therefore check of an invariance g is reduced to check of $g(w) = 0$, for all $w \in W$.

1. (induction base) $P = y_i$. Then $W(g, P) = \{y_i\}$.
2. (induction step)

a). Let $P = P_1 \vee P_2$.
Then

$$W(g, P_1 \vee P_2) = W(g, P_1) \cup W(g, P_2). \tag{7}$$

b). Let $P = P_1 * P_2$.

On an induction hypothesis, for the program P_2 the finite set of words $W(g, P_2)$ can be constructed. Let $W(g, P_2) = \{v_1, \ldots, v_l\}$. As from $w \in P$ follows $w = w_1 * w_2$, where $w_1 \in P_1$, $w_2 \in P_2$ we have: $g(w) = g(w_1 * w_2) = g(w_2(w_1))$, when

$$W(g, P_1 * P_2) = W(g(v_1), P_1) \cup \ldots \cup W(g(v_l), P_1). \tag{8}$$

On an induction hypothesis for the program P_1 and polynomials $g(v_i), i = 1, \ldots, l$ corresponding finite sets of words $W(g(v_1), P_1)$ can be constructed. Therefore finite set $W(g, P) = W(g, P_1 * P_2)$ also can be constructed.

c). Let $P = \{P_1\}$.

Then $P = e \cup P_1 \cup P_1^2 \cup \ldots$, where e - a indication of an identical computing statement. We shall consider the sequence of ideals:

$$(g) \subseteq (g, g(P_1)) \subseteq \ldots \subseteq (g, g(P_1), \ldots, g(P_1^m)) \subseteq \ldots$$

As the ring $A[X]$ is a Noetherian ring, this sequence is stabilized on some number m_0, i.e. exist such natural m_0, that at any $m \geq m_0$

$$(g) \subseteq (g, g(P_1), \ldots, g(P_1^m)) = \left(g, g(P_1), \ldots, g\left(P_1^{m+1}\right)\right)$$

.

Obviously, this equality takes place, if $g\left(P_1^{m+1}\right) \subseteq (g, g(P_1), \ldots, g(P_1^m))$. We shall show, that m_0 - the least natural number such, that

$$g\left(P_1^{m_0+1}\right) \subseteq (g, g(P_1), \ldots, g(P_1^{m_0})). \tag{9}$$

Really, substituting in (8) instead of X each of polynomials $w(X), w \in P_1$, we receive:

$$g\left(P_1^{m_0+1} * w\right) \subseteq (g(w), g(P_1 * w), \ldots, g(P_1^{m_0} * w)). \tag{10}$$

It means that

$$g\left(P_1^{m_0+1} * P_1\right) \subseteq \left(g(P)_1, g(P_1 * P_1), \ldots, g(P_1^{m_0} * P)_1\right). \tag{11}$$

So

$$g\left(P_1^{m_0+2}\right) \subseteq \left(g(P_1), \ldots, g\left(P_1^{m_0+1}\right)\right) \subseteq (g, g(P_1), \ldots, g(P_1^{m_0})).$$

Thus, from equality

$$(g, \ldots, g(P_1^{m_0})) = \left(g, \ldots, g(P_1^{m_0}), g\left(P_1^{m_0+1}\right)\right) \tag{12}$$

follows equalities

$$(g, \ldots, g(P_1^{m_0})) = \left(g, \ldots, g\left(P_1^{m_0+k}\right)\right) \tag{13}$$

for any natural k. Therefore

$$W(g, P) = W(g, E) \cup W(g, P_1) \cup \ldots \cup W(g, P_1^{m_0}). \tag{14}$$

As the problem of membership $g \in (q_1, \ldots, q_l)$ is algorithmically solvable, formulas (7) – (14) describe recursive algorithm of construction $W(g, P)$. The statement is proved.

b). For the proof of the statement b) theorems we shall consider a polynomial $g(X) = a_0 X_1^M + \ldots + a_{\varphi(M)}$ of the degree M with indefinite coefficients. We shall construct for this polynomial set $W(g, P)$. As the ring $A\left[a_0, \ldots, a_{\varphi(M)}, X\right]$ also a Noetherian ring, this construction can be effected by means of algorithm of item a). Let $W(g, P) = \{g_1, \ldots, g_l\}, g_k \in A\left[a_0, \ldots, a_{\varphi(M)}, X\right]$. Then the system of equalities $g_1 \equiv g_2 \equiv \ldots \equiv g_l$ defines the system of the linear homogeneous equations above A in unknowns $a_0, \ldots, a_{\varphi(M)}$. The fundamental system of solutions of this system sets required base $I_p^{(M)}$.

Example 3. Let's consider the application of a method i.a) of theorem 2 for the proof of an invariance of equality $(x^2 - xy - y^2)^2 = 1$ of the program which computes the least number of Fibonacci surpassing N.

```
Procedure Fib(N: Integer; var x: Integer);
var y: Integer;
Begin
  x := 1;
  y := 1;
  While x < N do Begin
   x := x + y;
   y := x - y
  End
{Invariant: (x^2 - x * y - y^2)^2 - 1}
End;
```

Let's construct the dictionary of computing statements:
$y_1 : x := 1; y := 1.$
$y_2 : x := x + y; y := y.$
$y_3 : x := x; y := x - y.$
Let's construct a regular expression of program:
$P = y_1 * \{y_2 * y_3\}$.
The consequent application of computing statements in the loop can be converted in one statement $y_4 : x := x+y; y := x$. The regular expression of program at this action will be simplified:
$P = y_1 * \{y_4\}$.
The program invariant we will represent as

$$g(x, y) = (x^2 - xy - y^2 - 1)(x^2 - xy - y^2 + 1).$$

Further,$P = P_1 * P_2, P_1 = y_1, P_2 = y_4$.
We construct $W(P_2, g).P_2 = e \vee y_4 \vee y_4^2 \vee ....$ Then we find number of stabilization of ideals chain
$(g) \subseteq (g, (g(y_4)) \subseteq (g, (g(y_4), g(y_4^2)) \subseteq ....$
We shall compute
$g(y_4) = g(x+y, x) = ((x+y)^2-(x+y)x-x^2-1)((x+y)^2-(x+y)x-x^2+1) = (-x^2 + xy + y^2 - 1)(-x^2 + xy - y^2 + 1) = (x^2 - xy - y^2 + 1)(x^2 - xy - y^2 - 1).$
So, $g(x, y) = g(x + y, x)$. Thus, $m_0 = 0$ and $g(\{y_4\}) = (g)$. We have proved, that $g(x, y)$ -is the loop invariant. We shall note if g -is invariant of loop and y - body of loop, so $g(y)$ -is also invariant of loop. This can be used. For example, if the ideal of loop invariants I_C is generated by one polynomial $g(X)$, the chain of ideals will be broken at the first step. If not, we receive new invariant.
Let $g_1(x, y) = x^2 - xy - y^2 - 1, g_2(x, y) = x^2 - xy - y^2 + 1$. At transformations of a loop $g_1(x, y)$ passes in$-g_2(x, y)$, and $g_2(x, y)$ - in$-g_1(x, y)$. That is the ideal of invariants of this example is not prime. It is decomposable in intersection

of prime ideals, generated accordingly by polynomials g_1 and g_2. Further, we'll compute $W(P, g)$. We will receive that $W(P_2, g) = \{e\}$. So $W(P, g) = \{y_1\}$. For the proof of invariance g it is left to check that $g(y_1) \equiv 0$. Substituting $x = 1, y = 1$ in $g(x, y)$, we can be convinced in this: $g(1, 1) = 0$.

Example 4. Generation of invariants by method of indefinite coefficients. The procedure of this example turns the point (x, y) into the angle $\arctan(\frac{3}{4})$ until it gets in e -neighborhood of its initial position (a, b).

```
Procedure Rotation (a, b: Real; e: Real; var x, y: Real);
var u, v: Real;
begin
   x := a;
   y := b;
  Repeat
    u := (4/5)*x - (3/5)*y;
    v := (3/5)*x + (4/5)*y;
    x := u;
    y := v;
  until Sqr((x-a)) + Sqr(y-b) < e;
end;
```

The dictionary of computing statements we will form reducing the statement in body of loop to the form

$y_1 : x := \frac{4}{5}x - \frac{3}{5}y; y := \frac{3}{5}x + \frac{4}{5}y; y_2 : x := a; y := b.$

The program is represented by regular expression $P = y_2 * y_1 * \{y_1\}$. We will search invariant in form of homogenous polynomial of 2-nd degree from variables x, y, a, b.

$$g(a, b, x, y) = Ax^2 + Bxy + Cy^2 + Da^2 + Eab + Fb^2 \tag{15}$$

The computations described in the proof of the theorem 2b, for practical purposes are badly suitable. In practice it is possible to use the methods planned in these and the following examples. Let's notice, that computations along any path of programs fulfilled for concrete numerical values of input parameters, lead to linear equations from indefinite coefficients $A, B...F$.

We shall consider the computations along the path $w_0 = y_1 * y_2$. We shall consider

$a = 1, b = 0$. We'll receive:

$$g(w_0(1, 0)) = \frac{16}{25}A + \frac{12}{25}B + \frac{9}{25}C + D = 0 \tag{16}$$

$a = 0, b = 1$. We'll receive:

$$g(w_0(0, 1)) = \frac{9}{25}A - \frac{12}{25}B + \frac{16}{25}C + F = 0 \tag{17}$$

$a = 1, b = 1$. We'll receive:

$$g(w_0(1,1)) = \frac{24}{25}A - \frac{7}{25}B - \frac{24}{25}C - E = 0 \tag{18}$$

Eliminating D, E, F from (15) , we'll receive (19)

$$g = A(x^2 - \frac{16}{25}a^2 - \frac{9}{25}b^2 + \frac{24}{25}ab) + B(xy - \frac{12}{25}a^2 + \frac{12}{25}b^2 - \frac{7}{25}ab) + C(y^2 - \frac{9}{25}a^2 - \frac{16}{25}b^2 - \frac{24}{25}ab) \tag{19}$$

The equations (16)- (18) are linearly independent. However, the further computations show, that the equations received at other initial values a, b along a path w_0, depend on these ones. That is possibilities of receiving of new equations with use of a path w_0 are settled.

Let's consider the computations along the path $w_1 = y_1 * y_2 * y_2$. Let's consider $a = 1, b = 0$. From (19) we'll receive:

$$g(w_1(1,0)) = \frac{351}{625}A - \frac{132}{625}B + \frac{351}{625}C = 0 \tag{20}$$

Eliminating coefficient C from (19), we'll receive:

$$g = A(x^2 - a^2 - b^2 + y^2) + B(xy - \frac{8}{13}a^2 + \frac{28}{117}b^2 - \frac{25}{39}ab + \frac{44}{117}y^2) \tag{21}$$

Let's consider $a = 1, b = 1$. From (21) we'll receive:

$$g(w_1(1,1)) = -\frac{50}{39}B = 0 \tag{22}$$

.

Eliminating coefficient B from (21), we'll receive:

$$g = A(x^2 - a^2 - b^2 + y^2) \tag{23}$$

Finally, $g = x^2 - a^2 - b^2 + y^2$ - is invariant of program.

6 The Main Problems and Algorithms

From the theorem proof 2a) it is seen that the main algorithmic problem to be solved at realization of corresponding algorithm is the problem

$$g\left(P_1^{m+1}\right) \subseteq \left(g, g\left(P_1\right), \ldots, g\left(P_1^m\right)\right).$$

In other words the problem comes to recognize the membership of some polynomial to polynomial ideal, specified by its base. This problem is a classic one of polynomial ideals theory, and for its solution it is better to use the Gröbner bases of polynomial ideals. This approach is indicated in [5], where the beginning

of work is announced on realization of program system, solving the problems of theorem 2.

So, all ideals, which are built in algorithm of theorem 2a) should be represented by their Gröbner bases. In future we will indicate the fact, that ideal I is represented by Gröbner base $f_1, \ldots, f_k$, by equality $I = (f_1, \ldots, f_k)_{Gr}$. Gröbner base of ideal I we will mark via $Gr(I)$.

Our algorithm of theorem 2a) should be bases upon the solution of the following problems:

1. For $I = (f_1, \ldots, f_k)_{Gr}$, $J = (g_1, \ldots, g_l)_{Gr}$ to find $K = (f_1, \ldots, f_k, h_1, \ldots, h_m)_{Gr}$.
2. For $I = (f_1, \ldots, f_k)_{Gr}$, computing statement $X := H(X)$, to find $J = (g_1, \ldots, g_l)_{Gr}$, that is to find $Gr((f_1(H(X)), \ldots, f_k(H(X))))$.
3. For $I = (f_1, \ldots, f_k)_{Gr}$, $J = (g_1, \ldots, g_l)_{Gr}$ to recognize $I \subseteq J$.

The problems 1 and 3 are classic ones in the theory of polynomial ideals, the solutions of which in Gröbner bases terms are well known. Let's examine now the questions, related to the realization of theorem 2b) algorithm. Let's note that from the proof method follows that to search the invariants is possible specifying them as polynomial forms that is polynomials of "special" form with indefinite coefficients. For example it can be defined the general form of desired invariants as linear combination of several polynomials with indefinite coefficients.

$$g(X) = a_1 * g_1(X) + \ldots + a_k * g_k(X). \tag{24}$$

If, for example, it is required to search the linear invariants, it should be laid

$$g_1 = x_1, \ldots, g_n = x_n, g_{n+1} = 1. \tag{25}$$

It can be searched for example all invariants of form

$$g(X) = a_1 * x_1^2 + \ldots + a_n * x_n^2 etc.$$

Since the computing statements leave the indefinite coefficients motionless, the algorithms of theorem 2a) should be applied coordinate-wise to polynomial vector $(g_1(X) \ldots g_k(X))$. Gröbner bases should be built separately for polynomials $g_1(X) \ldots g_k(X)$, and the condition of break of increasing chains p. b) of theorem algorithm 2a) to be applied to all vector of ideals $(I_1, \ldots, I_n)$.

Thus, the construction of finite set of words from symbols y_i of computing statements at program analysis $P = \{P_1\}$ finishes at such value of m, at which all coordinates of vector of ideals are stabilized.

7 Invariants of Rationally Defined Programs

In [3] it is noted that algorithms of theorem 2 can be modified for rationally defined programs. With this purpose we can describe the algorithm of solution of problem 2 in case when in statement $X := H(X)$.

$$H(X) = (h_1(X), \ldots, h_n(X)), h_i(X) = r_i(X)/s_i(X),$$

at that right hand sides in this statement are irreducible fractions.

For this for set $X =< x_1, \ldots, x_n >$ we'll put in accordance the set $Z =< z_1, \ldots, z_n >$, the variables of which play the role of auxiliary. Then let's consider the set of polynomials

$$f_1(Z), \ldots, f_k(Z), x_1 * s_1(Z) - r_1(Z), \ldots, x_n * s_n(Z) - r_n(Z). \quad (26)$$

Let's build the Gröbner basis of this set, eliminating the variables of Z set. The received Gröbner base $g_1(X), \ldots, g_k(X)$ is a solution of problem 2. The similar arguments can be applied also in cases when the right hand sides of computing statements contains the algebraic irrationalities (for example, the square root functions).

Example 5. In this example we will consider computations of program invariants, summarizing a program of the previous example. Exactly, the procedure of point (a, b) rotation to free angle α is being analyzed.

This angle is specified by input parameter $t = tan(\frac{\alpha}{2})$.

```
(t = tan(alpha/2), alpha − angle of rotation).
Procedure Rotation (a, b:Real; t:Real; e:Real; var x, y:Real);
var u, v: Real;
    c, s: Real;
begin
    c := (1-sqr(t))/(1+sqr(t));      {c = cos(alfa)}
    s := (2*t)/(1 + sqr(t));         {s = sin(alfa)}
    x := a;
    y := b;
    Repeat
      u := c*x - s*y;
      v := s*x + c*y;
      x := u;
      y := v;
    until Sqr((x-a)) + Sqr(y-b) < e;
end;
```

Let's formulate the dictionary of computing statements:

$y_1 : c := (1 - t^2)/(1 + t^2), s := 2 * t/(1 + t^2), x := a; y := b. y_2 : x := c * x - s * y; y := s * x + c * y.$

The program is represented by regular expression $P = y_1 * y_2 * \{y_2\}$. As in previous example we will search invariant in form of

$$g(a, b, x, y) = Ax^2 + Bxy + Cy^2 + Da^2 + Eab + Fb^2 \quad (27)$$

Some approaches to computations of invariants are possible. At first, it is possible to apply a method of the previous example. For this purpose it is necessary to substitute various numerical values instead of variables a, b, t. Secondly, the computations similar to computations of the previous example, it is possible to do above a field $Q(t)$. At last, it is possible to compute invariants after application of statement y_1 , using a method i.6 of eliminations of variables for rationally defined computing statements. We shall describe in more detail this approach.

For computing statement $y_1 : c := (1 - t^2)/(1 + t^2, s := 2 * t/(1 + t^2), x := a; y := b$. let's make a system of polynomials (26). We'll receive:

$(c + 1)t^2 + (c - 1), st^2 - 2t + s, x - a, y - b.$

Then we eliminate the variable t. We'll receive:

$c^2 + s^2 - 1, x - a, y - b.$

Further computations we execute above field $Q(a, b, c)\,[s]\,/(c^2 + s^2 - 1)$. Let's consider computations along the path $w_0 = y_1 * y_2$. To the right hand sides of statement y_2 we substitute a instead of x, b instead of y. All results can be represented with a system of equalities:

$$\begin{cases} g = Ax^2 + Bxy + Cy^2 + Da^2 + Eab + Fb^2 \\ x = ca - sb \\ y = sa + cb \\ c^2 + s^2 = 1 \end{cases} \tag{28}$$

Let's substitute the values x, y in the first equality of system (28) and we shall lead a polynomial $g(a, b, c, s)$ to a standard form. We shall lead a polynomial by modulo c^2+s^2-1, substituting c^2 on $1-s^2$. We shall lead the received polynomial $g(a, b, c, s)$ to a standard form and we shall equate to zero its coefficients. We shall receive:

$B = 0, A - C = 0, C + D = 0, F + A = 0, E = 0.$

Having eliminating from (28) coefficients B, C, D, E, F , we'll receive:

$g(x, y, a, b) = A(x^2 + y^2 - a^2 - b^2).$

It is left to prove that $g = x^2 + y^2 - a^2 - b^2$ - is invariant. We'll put in a system (25) all necessary equalities for computations

$$\begin{cases} u^2 + v^2 = a^2 + b^2 \\ u = cx - sy \\ v = sx + cy \\ c^2 + s^2 = 1 \end{cases} \tag{29}$$

We'll substitute in first equality of (29) values u, v and lead the received equality by modulo $c^2 + s^2 - 1$. We'll receive: $x^2 + y^2 = a^2 + b^2$.

In conclusion we would like to note that the solution of problem of base I_p construction is unknown to us.

References

1. A.A. Letichevskiy. About one approach to program analysis // Cybernetics. - 1979.- 6.-.1-8.(in russian)

2. A.B. Godlevskiy, Y.V. Kapitonova, S.L. Krivoy, A.A. Letichevskiy. Iterative methods of program analysis// Cybernetics. – 1989. – 2, P.9–19. (in russian)
3. M.S. Lvov. Invariant equalities of small degrees in programs, defined above field// Cybernetics. – 1988. – 1. – P. 108 – 110. (in russian)
4. A.Letichevsky, M.Lvov. Discovery of invariant Equalities in Programs over Data fields. Applicable Algebra in Engineering, Communication and Computing. – 1993. – 4. – pp. 21–29.
5. S.M. Lvov. About realization of computations in analysis problems, defined above vector spaces// Programming problems – 2004.–2–3. Special issue. P. 95–101. (in russian)
6. M.S. Lvov. Invariant inequalities in programs interpreted above ordered fields// Cybernetics. – 1986. – 5. – P.22–27. (in russian)
7. Sriram Sankaranarayanan, Henny Sipma, Zohar Manna: Non-linear loop invariant generation using Gröbner bases. POPL 2004: 318-329
8. Markus Müller-Olm, Helmut Seidl: Precise interprocedural analysis through linear algebra. POPL 2004: 330-341
9. Markus Müller-Olm, Helmut Seidl: Computing polynomial program invariants. Inf. Process. Lett. 91(5): 233-244 (2004)
10. Enric Rodriguez-Carbonell, Deepak Kapur: Automatic generation of polynomial loop invariants: algebraic foundations. ISSAC 2004: 266-273
11. Enric Rodriguez-Carbonell, Deepak Kapur: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. 64(1): 54-75 (2007)
12. Laura Ildikö Kovàcs, Tudor Jebelean: An Algorithm for Automated Generation of Invariants for Loops with Conditionals. SYNASC 2005: 245-249

Reflexive Transitive Loop Invariants: A Basis for Computing Loop Functions

Ali Mili

College of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102
mili@cis.njit.edu

Abstract. The search for loop invariants in the form of inductive assertions is useful to prove the correctness of loops in Hoare's logic, but is only indirectly useful to derive the function of a loop. In this paper we introduce a class of binary loop invariants, and discuss their application to the derivation of loop functions.

Keywords

Function extraction, loop functions, loop invariants, relational calculus, refinement calculus, computing loop behavior.

1 Unary and Binary Loop Invariants

Loop invariants in the form of inductive assertions have been the subject of extensive research since their introduction by C.A.R. Hoare in [13]. Their primary function is to prove the correctness of a while loop with respect to a specification that takes the form of a precondition and postcondition. In this paper, we present a different type of loop invariant, and discuss its use in the derivation of loop functions.

1.1 An Illustrative Example

To give the reader some intuition on the difference between traditional inductive assertions and the proposed loop invariants, we present a simple example involving a loop that computes the sum of an array.

```
x:  xtype; i:  1..N+1;  a:  array [1..N] of xtype;

begin
x:= 0;  i:= 1;
while (i <> N+1) do
   begin x:= x+a[i]; i:= i+1 end
end
```

For this loop, we let the precondition and postcondition be defined as:

$$\phi(s_0, s) \equiv a = a_0 \wedge x = 0 \wedge i = 1, \psi(s_0, s) \equiv x = \sum_{k=1}^{N} a_0[k].$$

An adequate invariant for this specification is:

$$\chi(s_0, s) \equiv a = a_0 \wedge x = \sum_{k=1}^{i-1} a[k].$$

According to [13], in order to prove that the while statement is partially correct with respect to the specification $(\phi(s_0, s), \psi(s_0, s))$, it suffices to prove the following premises.

1. Initial condition: $\phi(s_0, s) \Rightarrow \chi(s_0, s)$.
2. Invariance (Inductive) condition: $\{\chi(s_0, s) \wedge t\}\ B\ \{\chi(s_0, s)\}$.
3. Exit (Final) condition: $\chi(s_0, s) \wedge \neg t \Rightarrow \psi(s_0, s)$.

We leave it to the reader to check that these three premises hold for the specification and the loop invariant at hand.

While traditional loop invariants capture a property that holds initially, and after an arbitrary number of executions of the loop body, the type of loop invariants we illustrate in this section capture a reflexive transitive relation that exists between two states s and s' that are separated by an arbitrary number of iterations (i.e. s' is obtained from s by application of an arbitrary number of loop body instances, assuming the loop condition returns true whenever is tested). In other words, whereas traditional loop invariants are inductive in terms of the current state, the loop invariants we introduce in this paper are doubly inductive, providing for two states, s and s', to vary, where s precedes s' (by an arbitrary number of iterations). Also, whereas traditional loop invariants are dependent upon the loop as well as its context (in terms of initial state, precondition/ postcondition, etc) the new loop invariants depend exclusively on the loop, and remain unchanged regardless of the context in which the loop is embedded.

For illustration, we consider the following reflexive transitive relation

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

and we argue that this relation defines a loop invariant in the following sense: the predicate χ defined by

$$\chi(s_0, s) \equiv (s_0, s) \in R$$

satisfies the second premise of Hoare's rule, i.e.

$$\{\chi(s_0, s) \wedge t\}\ B\ \{\chi(s_0, s)\}.$$

For lack of space, we do not present a proof of this claim here; the interested reader is referred to [21]. Of course, we did not introduce reflexive transitive loop invariants just to derive traditional invariant assertions; rather we introduced them because they allow us to generate the function of while loops, as we will discuss later in the paper. But we resolved to prove the formula

$$\{(s_0, s) \in R \wedge t\}\ B\ \{(s_0, s) \in R\}$$

only to justify that we can refer to R as a loop invariant, in the traditional sense.

1.2 Narrative Characterizations

In this subsection, we build on the intuition conveyed by the example above about the contrast between traditional loop invariants and the new form of loop invariants to elucidate the various dimensions of contrast between them, in preparation for giving a formal definition and formal characterizations of these concepts. Specifically, we list the following premises:

- *Different Arities.* Traditional inductive assertions have the form $\phi(s_0, s)$, where s_0 represents the initial state; whereas the proposed loop invariants have the form $(s, s') \in R$, where s and s' are both arbitrary states. Hence the former are unary predicates, whereas the latter are binary predicates. In the sequel, we refer to them respectively as *unary invariants* and *binary invariants.*
- *Different Dependencies.* A far more important distinction is that a unary loop invariant depends not only on the loop under consideration but also on the specification (precondition/ postcondition pair). By contrast, a binary loop invariant depends exclusively on the loop under consideration, regardless of its context. This is why, in the example presented above, we checked the second condition of Hoare's method, i.e.

  $$\{(s_0, s) \in R \wedge t\}\ B\ \{(s_0, s) \in R\}$$

  but we did not check the initial condition nor the final condition. The binary loop invariant does not depend on the precondition and the postcondition, hence is not subject to any equation that involves them. An immediate consequence of this difference, is that if we change the specification or the initialization of the loop we have to change its unary invariants, but we do not have to change its binary invariant.
- *Different Levels of Generality.* Specifically, we argue that binary loop invariants subsume unary loop invariants, in the following sense: from any binary loop invariant (R), we can generate a unary loop invariant (χ) that satisfies Hoare's invariance condition

  $$\{\chi() \wedge t\}\ B\ \{\chi()\}.$$

– *Different Methods.* Due to their unary nature, unary loop invariants can be derived and analyzed by induction on the trace of execution of the loop: if the invariant holds up to a point in the execution trace, then it holds after one more execution of the loop body. Because binary loop invariants involve two arguments (s and s'), a simple induction on the execution trace does not do them justice; then we recourse to an induction on the loop structure (if the function of the loop body satisfies this property, then the function of the loop satisfies that property).

We often encounter loop invariants that are not only reflexive and transitive, but also symmetric: for those, the roles of s and s' are interchangeable, anyone of them may precede the other, all we know is that they are an arbitrary number of iterations apart.

In the next section, we introduce some mathematical background, which we use in section 3 to formally define, using relational abstractions, traditional loop invariants and the new form of loop invariants. In section 4 we discuss characterizations of (strongest) unary loop invariants and binary loop invariants, to elucidate their subtle relations. Finally we discuss the application of binary loop invariants in section 5 and conclude with a summary and assessment of our findings as well as a review of some related work.

2 Mathematical Background

2.1 Elements of Relations

We represent the functional specification of programs by relations; without much loss of generality, we consider homogeneous relations, and we denote by S the space on which relations are defined. A relation R on set S is a subset of the Cartesian product $S \times S$, hence it is natural to represent general relations as $R = \{(s, s')|p(s, s')\}$, for some binary predicate p. Typically, set S is defined by some variables, say x, y, z; hence an element s of S has the structure $s = \langle x, y, z\rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ (resp. $x(s')$, $y(s')$, $z(s')$) to refer to the x-component, y-component and z-component of s (resp. s'). We may, for the sake of brevity, write x for $x(s)$ and x' for $x(s')$ (and do the same for other variables).

Constant relations 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. Operations on relations also include the *converse*, denoted by $\widehat{R}$ or $R^{\frown}$, 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 $dom(\widehat{R})$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined by $R\widehat{R}$. For any R, the nucleus of R is symmetric and reflexive on $dom(R)$. 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, $RL = L$.

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 Relations Based Refinement

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, of course, to refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [12, 23]. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit that, modulo traditional definitions of total correctness [9, 12, 17], 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.
- $R \sqsupseteq R'$ 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'. We admit without proof that any relation R can be refined by a deterministic relation, i.e. a function. We also admit without proof that the refinement relation is a partial ordering. In [3] 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: $RL \cap R'L = (R \cap R')L$. Under this condition, their least upper bound is referred to as the *join*, denoted by $\sqcup$, and defined by:

$$R \sqcup R' = \overline{RL} \cap R' \cup \overline{R'L} \cap R \cup (R \cap R').$$

- 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.

3 Relational Definitions

Because binary loop invariants involve two states, it is natural to represent them with (binary) relations; for the sake of uniformity, we will also use binary relations (specifically, vectors) to represent unary loop invariants. In this paper we assume implicitly that loops terminate for all elements of their space, i.e. they define total functions; we have shown in [20] that this hypothesis does not restrict generality (we can always redefine the space of the loop to be the set of states on which the loop terminates —Mili et al. [20] show that this does not exclude any states of interest).

3.1 Unary Loop Invariants

We consider a while loop of the form

```
w = while t do B
```

on space S, and we let $\chi()$ be a loop invariant of w. The invariance condition of $\chi()$ can be written as

$$\{\chi() \wedge t\}\ B\ \{\chi()\}.$$

We interpret this condition in logical terms as:

$$\forall s, s' : \chi(s) \wedge t(s) \wedge (s, s') \in B \Rightarrow \chi(s').$$

If we let V be the vector defined by $V = \{(s, s')|\chi(s)\}$ and T be the vector defined by $T = \{(s, s')|t(s)\}$, then we can rewrite this condition as:

$$\forall s, s' : (s, s') \in V \cap T \cap B \Rightarrow (s, s') \in \widehat{V}.$$

Given that s and s' are arbitrary, this can be written algebraically as, $V \cap T \cap B \subseteq \widehat{V}$. Whence the following definition,

Definition 2. *Given a while statement of the form,* `w = while t do B`*, a unary loop invariant is defined as a vector* V *on* S *that satisfies the following condition:*

$$V \cap T \cap B \subseteq \widehat{V},$$

where T *is the vector defined by predicate* t*.*

3.2 Binary Loop Invariants

Because binary loop invariants involve two states, a past state and a current state, it is natural to represent them with binary relations.

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 a* binary loop invariant *for* w *if and only if* R *is reflexive, transitive, and satisfies the following conditions (where* T *is the vector defined by predicate* t*):*

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

The following proposition explains why we refer to the first condition of definition 3 as the *invariance condition.*

Proposition 1. *Given a while statement* `w = while t do B` *on space* S*, and given a binary loop invariant* R *of* w*. If we let* $\chi(s_0, s)$ *be the predicate defined by:* $\chi(s_0, s) \equiv (s_0, s) \in R$ *for some state* s_0 *of* S*, then* χ *satisfies the following Hoare formula:*

$$\{\chi(s_0, s) \wedge t(s)\}\ B\ \{\chi(s_0, s)\}.$$

Proof. By hypothesis, we have $T \cap [B] \subseteq R$. Left multiplying by R on both sides, we obtain $R \circ (T \cap [B]) \subseteq R \circ R$. Because R is transitive, we get: $R \circ (T \cap [B]) \subseteq R$. By set theory, we get: $(s_0, s) \in (R \circ (T \cap [B])) \Rightarrow (s_0, s) \in R$. By definition of the relational product, we get:

$$(s_0, s') \in R \wedge t(s') \wedge (s', s) \in [B] \Rightarrow (s_0, s) \in R.$$

Because $[B]$ is a function, we can write this as: $(s_0, s') \in R \wedge t(s') \Rightarrow (s_0, [B](s')) \in R$. Using the definition of predicate $\chi(s_0, s)$, we find

$$\chi(s_0, s') \wedge t(s') \Rightarrow \chi(s_0, [B](s')).$$

Interpreting this in the Hoare notation, and replacing the mute variable s' by the equally mute s, we find

$$\{\chi(s_0, s) \wedge t(s)\}\ [B]\ \{\chi(s_0, s)\}.$$

qed

In other words, the invariance condition (in the sense of definition 3) of binary loop invariants yields the invariance condition (in the traditional sense of Hoare's method) of unary loop invariants. As for the convergence condition of definition 3, it can be interpreted as follows: for any state s in space S, there exists a state s' such that (s, s') is an element of R and s' satisfies $\neg t$. In other words, R links any state s into a state s' that satisfies the termination condition $(\neg t)$.

4 Relational Characterizations

Whereas in the previous section we presented definitions of unary and binary loop invariants, in this section we attempt to give general characterizations of these invariants. In order to make the contrast between unary and binary loop invariants palatable, we aim to derive the strongest (in a sense to be defined) loop invariants of each type.

4.1 Unary Loop Invariants

We consider a while loop on space S, of the form

```
w =  while t do B
```

and we are interested in a strongest unary loop invariant for this loop. As we have discussed in section 1.2, a unary loop invariant does not depend exclusively on the loop, but on the loop's context. Hence we embed this loop in a larger program structure, which we use to derive a unary loop invariant. Specifically, we consider the following program structure, which we annotate by intermediate assertions and inductive 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. [19], which is based on earlier findings by Morris and Wegbreit [24], Mills [22] and Basu and Misra [2] provide that program f computes some total function F on S if:

1. The specification N defined by $N = F\widehat{F} \cap L(\widehat{F \cap W})$ (where W is the function of the while loop) is total.
2. Segment `init` is correct with respect to specification $N = F\widehat{F} \cap L(\widehat{F \cap W})$.
3. The following predicate is a loop invariant: $\chi(s_0, s) \equiv (s_0, s) \in F\widehat{F} \cap L(\widehat{F \cap W})$.

To illustrate this result, we consider again the program of array sum that we used in section 1.1. The space of interest is defined by the following variable declarations:

```
x:  xtype; i:  1..N+1;  a:  array [1..N] of xtype;
```

As for the program structure, it is defined as follows:

```
f= begin
   x:= 0;  i:= 1;
   while (i <> N+1) do
      begin  x:= x+a[i]; i:= i+1 end;
   end
```

The function of program f, which we denote by F, is given by the following formula:

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

As for the function of the loop, it is defined by

$$W\begin{pmatrix} x \\ i \\ a \end{pmatrix} = \begin{pmatrix} x + \sum_{k=i}^{N} a[k] \\ N+1 \\ a \end{pmatrix}$$

From these definitions, we derive the specification N according to the formula provided above:

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

Details of this calculation are not given here; the interested reader may consult [21]. This specification is total; also, the initialization segment of the program of interest is correct with respect to N, since by setting x to zero and setting i to 1 it makes the following equation hold:

$$x = \sum_{k=1}^{i-1} a[k].$$

A (strongest) unary loop invariant (strong enough to capture all the functional details of F) for this while loop is then $(s_0, s) \in N$, which we interpret as

$$a = a_0 \wedge x = \sum_{k=1}^{i-1} a[k],$$

which is the same loop invariant that we had proposed in section 1.1, only there we proposed it intuitively, whereas here we compute it from our formula.

4.2 Binary Loop Invariants

Whereas unary loop invariants depended on the loop as well as its context, binary loop invariants depend solely on the while loop. The following proposition provides a strongest binary loop invariant for a while loop.

Proposition 2. *Given a while loop* `w = while t do B` *on some space S, we let W be the function of w. Then, the relation $R = W\widehat{W}$ is a binary loop invariant for w.*

The proof of this proposition is beyond the scope of this paper; the interested reader may consult [21]. Proposition 2 shows that $W\widehat{W}$ is a binary loop invariant, but does not show that it is a strongest binary loop invariant, not to mention that we did not even define what it means to be strongest. The next section will elucidate both questions.

5 Binary Loop Invariants and Loop Functions

5.1 Theorem of Binary Loop Invariants

The interest of binary loop invariants is reflected in the following theorem, due to [20].

Theorem 1. *We consider a while loop on space S of the form* `w = while t do B`. *If R is a binary loop invariant for w, then the loop function W refines the following expression:*

$$R \cap \widehat{\widehat{T}}$$

where T is the vector defined by predicate t.

In other words, if R is a binary loop invariant for w, then we can infer

$$W \sqsupseteq R \cap \widehat{\widehat{T}}.$$

Interpretation of this theorem: The function of the loop is actually given by the following expression:

$$W = (T \cap [B])^* \cap \widehat{\widehat{T}},$$

where $(T \cap [B])^*$ is the reflexive transitive closure of $(T \cap [B])$, i.e. the smallest reflexive transitive relation that is a superset of $[B]$. Of course, in practice, it is very difficult in general to derive the transitive closure of an arbitrary function. What this theorem does is to strike a deal with us:

- Rather than ask us to derive the smallest superset of $(T \cap [B])$ that is reflexive and transitive, it asks us for a binary loop invariant of w, which is an arbitrarily large superset of $(T \cap [B])$ that is reflexive and transitive.
- On the other hand, rather than provide us with the exact function of the loop, it provides us with a lower bound (in the refinement ordering) of the loop function.

This theorem enables us to derive the loop function by deriving arbitrary (arbitrarily large/ weak) binary loop invariants, then combining them (by the lattice operation of *join*) to obtain the loop function (or an approximation thereof). For example, if we have established:

$$W \sqsupseteq V_1,$$

$$W \sqsupseteq V_2,$$

then we can infer

$$W \sqsupseteq V_1 \sqcup V_2.$$

This approach is all the more attractive if we write the loop body B as a set of concurrent assignments, because then we can can derive supersets of B by looking at any subset of the concurrent assignments at a time (for example, one at a time, two at a time, three at a time).

5.2 Deriving Lower Bounds

Theorem 1 tells us how to derive a lower bound of the loop function once we have a binary loop invariant, but we still need means to derive binary loop invariants to use it. In the sequel, we briefly present a pattern recognition method that derives binary loop invariants by matching concurrent assignments of the loop body against specific patterns, and infers a lower bound whenever a match is successful.

To this effect, we use a repository of *recognizers*, where a recognizer is characterized by its state space, the pattern of statements it recognizes, and the lower bound that it provides for $[w]$. Hence the derivation of the loop function may proceed by matching parts of the loop body, written as a set of concurrent assignments, against existing statement patterns, and producing lower bounds for $[w]$ in case of a match. This algorithm has at its disposal a database of recognizers, which it scans starting with 1-Recognizers (that match one assignment statement), then 2-Recognizers (that match combinations of two statements), then 3-Recognizers (that match triplets). To keep the combinatorics tractable, we limit ourselves to recognizers whose length does not exceed 3. Due to space limitations, we limit ourselves in this paper to recognizers of length 2; the interested reader is referred to [21].

5.3 Sample 1-Recognizer

Generally, 1-Recognizers answer the question: what can we infer about the loop function if we know that this statement (in the loop body) gets executed an arbitrary number of times? We present and illustrate a sample 1-Recognizer given in Figure 1. For illustration, let us consider a while loop whose loop body is written as a set of concurrent assignments, as follows:

```
while y>0 do
   {... ... ...
   x:= x+c,
   ... ... ...}
```

where x is an integer variable and c is an integer constant greater than 0. Application of this sample recognizer provides that $[w]$ refines the following specification:

$$V = \{(s, s') | x \textbf{ mod } c = x' \textbf{ mod } c \wedge y' \leq 0\}.$$

Note that we could make this claim on the loop function using very little information on the loop, regardless of what the ellipsis in the loop body stands for.

5.4 Sample 2-Recognizers

Generally, 2-Recognizers answer the question: what can we infer about the loop function if we know that these two statements get executed the same number

State Space	Semantic Pattern	Lower Bound
x: int const c: int >0	x:=x+c	$V =$ $\{(s,s')\|x \textbf{ mod } c = x' \textbf{ mod } c \wedge \neg t(s')\}$

Fig. 1. Sample 1-Recognizer

State Space	Semantic Pattern	Lower Bound
x: listType y: listType	y:=y.head(x) x:=tail(x)	$V =$ $\{(s,s')\|x.y = x'.y' \wedge \neg t(s')\}$
i: int x: sometype	i:=i-1, x:=f(x)	$V =$ $\{(s,s')\|f^i(x) = f^{i'}(x') \wedge \neg t(s')\}$

Fig. 2. Sample 2-Recognizer

of times? We present and illustrate two sample 2-Recognizers given in Figure 2, where `head` and `tail` represent respectively the head of the list (its first element) and its tail (the remainder of the list), and f is an arbitrary function on `sometype`. For illustration, we consider a while statement that contains the following statements:

```
while not empty(x)
   {
   ... ... ...
   y:= y.head(x),
   x:= tail(x),
   i:=i-1,
   ... ... ...
   }
```

Application of the first semantic recognizer to the first and second statements produces (after simplification) the following lower bound for $[w]$:

$$V_1 = \{(s,s')|x' = \epsilon \wedge y' = y.x\}$$

where ϵ is the empty sequence. Application of the second recognizer to the second and third line produces (after simplification, using the axiomatization of lists) the following lower bound for $[w]$:

$$V_2 = \{(s,s')|x' = \epsilon \wedge i' = i - length(x)\},$$

where $length(x)$ is the length of x. Taking the join, we find

$$[w] \sqsupseteq \{(s,s')|x' = \epsilon \wedge y' = y.x \wedge i' = i - length(x)\}.$$

6 Conclusion

6.1 Summary and Assessment

In this paper, we have briefly presented the concept of *binary loop invariant* and contrasted it to the more commonly used *unary loop invariant*. Also, we have briefly and cursorily discussed how binary loop invariants can be used to derive unary loop invariants, and most importantly how they can be used to derive loop functions.

One of the most interesting results of this paper is perhaps the distinct characterizations we have given of a strongest (in the sense: strong enough to prove that the program computes its function) unary loop invariant and a strongest (in the sense: strong enough to derive the loop function) binary loop invariant. We have found a strongest unary loop invariant to have the form

$$(s_0, s) \in F\widehat{F} \cap L(\widehat{(F \cap W)}),$$

where W is the function of the unitialized while loop and F is the function of the initialized while loop. On the other hand, we have found that a strongest binary loop invariant has the form

$$(s, s') \in W\widehat{W},$$

where W is the function of the loop. Of course these formulas are not useful for deriving these invariants but they are useful for elucidating what these invariants represent.

Another important result is that binary loop invariants can be used to derive loop functions, and that they in turn can be derived using pattern matching of data structures and control structures of the loop.

6.2 Relation to Other Work

In this section we briefly mention some samples of current research on loop invariants, and characterize their approach. In [10] Ernst et al. discuss a system for dynamic detection of likely invariants; this system, called Daikon, runs candidate programs and observes their behavior at user-selected points, and reports properties that were true over the observed executions, using machine learning techniques. Because these are empirical observations, the system produces probabilistic claims of invariance. In [8], Denney and Fischer analyze generated code against safety properties, for the purpose of certifying the code. To this effect, they proceed by matching the generated code against known idioms of the code generator, which they parameterize with relevant safety properties. Safety properties are formulated by invariants (including loop invariants), which are inferred by propagation through the code. In [6], Colon et al. consider loop invariants of numeric programs as linear expressions and derive the coefficients of the linear expressions by solving a set of linear equations; they extend this work to non linear expressions in [25]. In [15, 16] Kovacs and Jebelean derive

loop invariants by solving recurrence relations; they pose the loop invariants as solutions to recurrence relations, and derive closed forms of the solution using a theorem prover (Theorema) to support the process. In [4] Rodriguez Carbonnell et al. derive loop invariants by forward propagation and fixed point computation, with robust theorem proving support; they represent loop bodies as conditional concurrent assignments, hence their insights are of interest to us as we envision to integrate conditionals into our concurrent assignments. Less recent work on loop invariants includes work by Cheatham and Townley [5], Karr [14], Cousot and Halbwachs [7], and Mili et al. [18]. Work on loop analysis and loop transformations in the context of compiler construction is also related to functional extraction, although to a lesser degree than work on loop invariants [1, 11].

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, Boston, MA, 1993.
2. S. Basu and J. Misra. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1):76–86, 1975.
3. N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.
4. E. R. Carbonnell and D. Kapur. Program verification using automatic generation of invariants. In *Proceedings, International Conference on Theoretical Aspects of Computing '2004*, volume 3407, pages 325–340. Lecture Notes in Computer Science, Springer Verlag, 2004.
5. T. E. Cheatham and J. A. Townley. Symbolic evaluation of programs: A look at loop analysis. In *Proc. of ACM Symposium on Symbolic and Algebraic Computation*, pages 90–96, 1976.
6. M. A. Colon, S. Sankaranarayana, and H. B. Sipna. Linear invariant generation using non linear constraint solving. In *Proceedings, Computer Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 84–97, 1978.
8. E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings, the Fifth International Conference on Generative programming and Component Engineering*, Portland, Oregon, 2006.
9. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
10. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.
11. T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers*. Springer Verlag, Berlin, Germany, 2003.
12. D. Gries. *The Science of programming*. Springer Verlag, 1981.
13. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.

14. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
15. L. Kovacs and T. Jebelean. Automated generation of loop invariants by recurrence solving in theorema. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, pages 451–464, Timisoara, Romania, 2004. Mirton Publisher.
16. T. J. L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. P. et. al., editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS05), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC05)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.
17. Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.
18. A. Mili, J. Desharnais, and J. R. Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.
19. A. Mili, J. Desharnais, and F. Mili. Relational heuristics for the design of deterministic programs. *Acta Informatica*, 24(3):239–276, 1987.
20. A. Mili, M. Pleszkoch, and R. C. Linger. Towards the automated derivation of loop functions. Technical report, New Jersey Institute of Technology, http://web.njit.edu/˜ mili/loopx.pdf, 2006.
21. A. Mili. Reflexive Transitive Loop Invariants: A Basis for Computing Loop Functions. Technical report, New Jersey Institute of Technology, http://web.njit.edu/˜ mili/wing.pdf, 2007.
22. H. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.
23. C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.
24. J. Morris and B. Wegbreit. Program verification by subgoal induction. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume II, chapter 8. Prentice Hall, Englewood Cliffs, NJ, 1977.
25. S. Sankaranarayana, H. B. Sipna, and Z. Manna. Non linear loop invariant generation using groebner bases. In *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*, pages 381–329, 2004.
26. A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3), September 1941.
27. A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, (5):285–309, 1955.

Author Index