Program Verification with Property Directed Reachability

Tobias Welp

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-225
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-225.html

December 18, 2013

Program Verification with Property Directed Reachability

by

Tobias Welp

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering–Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Andreas Kuehlmann, Chair
Professor Sanjit Seshia
Professor Ching-Shui Cheng

Fall 2013

Program Verification with Property Directed Reachability

Abstract

Program Verification with Property Directed Reachability

by

Tobias Welp

Doctor of Philosophy in Engineering–Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Andreas Kuehlmann, Chair

As a consequence of the increasing use of software in safety-critical systems and the considerable risk associated with their failure, effective and efficient algorithms for program verification are of high value. Despite extensive research efforts towards better software verification technology and substantial advances in the state-of-the-art, verification of larger and complex software systems must still be considered infeasible and further advances are desirable.

In 2011, Property Directed Reachablity (PDR) was proposed as a new algorithm for hardware model checking. PDR outperforms all previously known algorithms for this purpose and has additional favorable algorithmic properties, such as incrementality and parallelizability.

In this dissertation, we explore the potential of using PDR for program verification and - as product of this endeavor - present a sound and complete algorithm for intraprocedural verification of programs with static memory allocation that is based on PDR.

In the first part, we describe a generalization of the original Boolean PDR algorithm to the theory of quantifier-free formulae over bitvectors (QF_BV). We implemented the algorithm and present experimental results that show that the generalized algorithm outperforms the original algorithm applied to bit-blasted versions of the used benchmarks.

In the second part, we present a program verification frontend that uses loop invariants to construct a model of the program that overapproximates its behavior. If the

verification fails using the overapproximation, the QF_BV PDR algorithm from the first part is used to refine the model of the program. We compare the novel algorithm with other approaches to software verification on the conceptional level and quantitatively using standard competition benchmarks. Finally, we present an extension of the proposed verification framework that uses a previous dynamic approach to program verification to strengthen the discussed static algorithm.

Contents

Acknowledgments

First and foremost, I would like to thank my advisor Andreas Kuehlmann for his trust in me when he took me as student, his patience during my professional development, his excellent teaching, and his moral and financial support.

I would also like to thank the other members of my dissertation committee, Sanjit Seshia and Cheng-Shui Cheng, for reviewing my dissertation and their constructive feedback.

I am also grateful to Robert Brayton and Satish Rao for serving on my qualification examination committee and would like to acknowledge Hans Eveking, whose teaching has awakened my interests in Electronic Design Automation and who helped me with connecting with the research community in Berkeley.

I would also like to thank Donald Chai and Nathan Kitchen, who have guided me in my transition from student to researcher. Nathan has been a model for integrity and moral standards to me and his mentoring has provided orientation and understanding in my early years at Berkeley. Donald's candid feedback has often helped me to see flaws in my thinking and to correct them where necessary.

Additionally, I would like to thank Niklas Eén and Alan Mishchenko. Their enthusiasm for Property Directed Reachability has been essential to stimulate my interest for the algorithm and for finding the subject of my dissertation.

Finally, I would like to thank Philip Chong for his close friendship, altruistic help, and lasting encouragement. Without Philip, my time at Berkeley would neither have been that productive nor that fun.

Chapter 1

Introduction

We start this chapter with an introduction to program verification and some introductory notes about the algorithm Property Directed Reachability (PDR) which is the basis of the work reported on in this dissertation. Next, in Section 1.4, we introduce our framework for program verification from a high-level view. We conclude the chapter with a discussion of the overall structure of this thesis.

1.1 The Case for Quality Assurance of Software Systems

Microprocessors are ubiquitous in today's life. Practically every device sold in market sectors such as consumer electronics, domestic and medical appliances, automobiles, and avionics contains several microprocessors. Their injection have come with enormous savings in development and production costs and allowed for many more features previously infeasible to provide. A modern car, for instance, contains a large number of microprocessors for applications as diverse as the electronic stability control (ESC) system, navigation, and on-board entertainment.

The implemented microprocessors are programmed with software which is mostly written by human developers. Experience shows that writing software free of errors is practically impossible for all but the most trivial programs. One often measures the quality of software in the residual error density that is the number of errors per 1000 lines of code (kLOC). Several publications report statistics for the residual error density. The numbers reported in [McC04] suggest that in the industrial average, one can expect a residual error density of 15-50. For safety-critical applications, one can anticipate that more effort

is spent for quality insurance than on the industrial average and that the residual error density is closer to 1 as e.g. described in [CM90]. Most errors in software have only minor impact on the users' experience.

However, in particular for safety-critical applications, errors in software programs can have grave consequences. Incidents have shattered the historic overconfidence in software of developers and users alike. A particular sad example are the accidents with the Therac-25, a radiation therapy device that has been operated in cancer treatment facilities in the United States of America and in Canada. The device used software interlocks to prevent that a high energy beam is targeted at a patient without the necessary filters in place [LT93]. Unfortunately, the control software contained at least two bugs which caused the machine to administer massive overdoses to six patients, resulting in serious injuries and death. Other infamous examples include the explosion of the Ariane 5 on its maiden flight due to an unsafe number conversion in the control software costing approximately $370 million [Dow97] and the failure of the AT&T long distance network in 1990 due to a logical error in the control software of the network switches costing AT&T roughly $60 million in fees for not connected calls [Bur95]. Overall, a frequently cited NIST study from 2002 [Tas02] suggests that there is approximately a $60 billion economic loss each year due to software bugs only in the United States of America.

1.2 Technology and Methodology for Quality Software

Many methods have been devised to increase the quality of software (Figure 1.1). The first

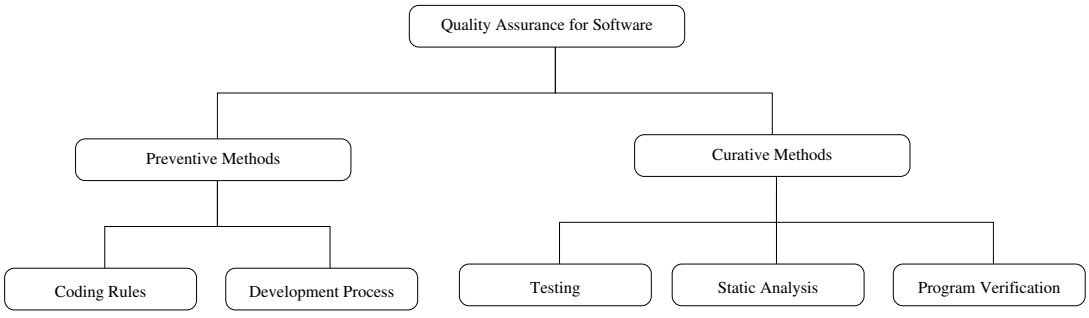


Figure 1.1: Means to Assure Quality of Software

category of methods aims at avoiding that errors are coded into the software in the first place (preventive measures). Among them are coding rules (well readable code, proper

indentation, avoidance of error-prone features of a programming language, ...) and a well-established development process. The second category aims at finding and eliminating defects that are already in the program (curative measures). The most significant technique in this category is testing: The program is exercised with user input and the system response is compared with an expected response. Testing can be applied at different levels of granularities such as at the unit (function, class, file) level or at the system level. Static alternatives to testing are static analysis and program verification. In both cases, software is analyzed to find issues in the program that are reported to the user. In the case of static analysis, the process is usually neither sound nor complete, i.e. the applied algorithms are not guaranteed to find all errors and a reported defect is not guaranteed to be a real bug. The aim of program verification is to either proof a safety property (something bad will never happen) or a lifeness property (something good will eventually happen) of a program. If the verification succeeds, the program is proved to have the property of interest. Otherwise, a counterexample is emitted.

1.2.1 Program Verification

When used in a development environment, the use of program verification adds a substantial amount of additional cost. For instance, the desired properties of the program must be defined formally, verification software must be obtained, and a verification engineer must expect a high computational burden to prove a certain property. To be useful, the benefits of using program verification must outweigh the cost.

Gerald J. Holzmann discussed this question in [Hol01]. He classifies defects into categories according to the difficulty of their stimulation and according to the severity of the potential impact. Using these categories, he formulates the hypothesis that in a typical program, these categories are not independent but that difficulty of stimulation of defects correlates positively with their severity, i.e. a bug that is difficult to stimulate is more likely to have catastrophic consequences. The author gives some arguments in support of this hypothesis and one can also argue that the hypothesis is true for the grave incidents reported above. Holzmann continues by pointing out that testing excels in finding defects that are easy to stimulate. On the other side, the strength of program verification rests in finding bugs independent of the difficulty of their stimulation. This allows to conclude that program verification is more likely to be successful in finding bugs that are hard to

stimulate than testing. If the formulated hypothesis is correct, then this also implies that program verification is more likely to find bugs that lead to catastrophic accidents.

The quest for finding good verification algorithms for software is challenged by the fact that program verification is an undecidable problem. To see why, note that Turing's halting problem [Tur36] can trivially be reduced to program verification. Strategies to cope with this challenge include to restrict the problem such as for instance disallowing dynamic memory allocation, to devise semi-algorithms that are not guaranteed to terminate, or a mixture thereof.

In practice, in addition to the challenge of undecidability, the implementations of software verification algorithms suffer from poor scalability. To alleviate this problem, verification algorithms use several abstraction techniques to reduce the computational complexity. For instance, modeling bit-vector arithmetic as linear arithmetic is a popular means to achieve this. The disadvantage of applying these methods is that the verification algorithms no longer solve the original problem, hence may produce false positives, false negatives, or both.

The extensive research efforts towards better software verification technology has advanced the state-of-the-art substantially. However, due to the high complexity of the problem, verification of larger software systems must still be considered infeasible. As such, and in the light of software systems permeating more and more safety-critical applications, advances to the currently available technology are certainly desirable. This dissertation describes such an effort.

1.3 Property Directed Reachability

PDR has originally been proposed by Bradley [BM08, Bra11] as an algorithm for hardware model checking. It shows excellent runtime performance on competition and industrial benchmarks and in particular has shown to outperform interpolation-based verification [McM03], until then the best algorithm for hardware model checking.

PDR and interpolation-based verification employ a similar high-level strategy in attempting to solve a model checking instance: Both algorithms repeatedly calculate overapproximate forward images starting from the initial set until one reaches a fixpoint. However, the mechanisms of how the overapproximate forward images are calculated differ substantially. The strategy to this end employed by interpolation-based verification is

to use interpolants [Cra57] derived from the proof of unsatisfiability of a bounded model checking instance [BCC+03]. In contrast, PDR composes the forward image piece by piece guided by suspected counterexamples to the property under verification. The latter strategy has important advantages to the former. First, it refrains from unrolling the transition relation and renders the overall algorithm to have modest memory requirements only. Second, it is incremental, and as such enables comparatively easy parallization and initialization with known facts about the model checking instance. Third, the combination of forward image calculation with guidance from counterexamples makes the algorithm bidirectional, allowing it to perform well for finding counterexamples and proving that none exist alike.

The discussed properties of PDR are not only appreciable for hardware model checking but for program verification alike. As such the question whether PDR can be used for program verification is immediate. This is the topic of this dissertation.

1.4 Using Property Directed Reachability for Program Verification

Figure 1.2 illustrates our framework that uses PDR for program verification. The frameworks implements the counterexample guided abstraction refinement paradigm [CGJ+03]: Initially, one constructs a model of the program under verification that overapproximates its behavior. If the property can be proved using this model, we can infer that the program is safe. Otherwise, one checks whether the counterexample is a real bug of the program. In case it is, we can infer that the program is not safe. If the counterexample is spurious, we use the PDR algorithm to refine the model of the program.

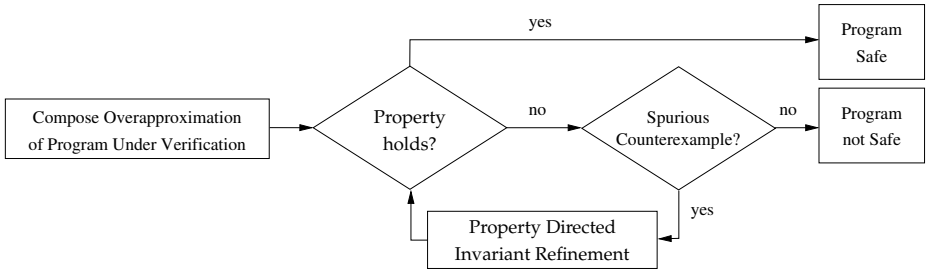


Figure 1.2: High-Level Overview of the Proposed Framework

In addition to using PDR for the refinement of the current model of the program under verification in the backend, the framework leverages several other design principles of PDR:

1. At all times, the program is modeled as a directed acyclic graph with size proportional to the size of the program under verification. Similar to PDR, which refrains from unrolling the transition relation, our algorithm refrains from unrolling loops.

2. The loops themselves are modeled using loop invariants. In the PDR algorithm, counterexamples to induction are used to refine the overapproximate forward images. Similarly, our framework uses the PDR algorithm to refine the loop invariants. In both cases, the refinement moves are guided by counterexamples, in other words, both algorithms are property directed.

3. The framework attempts to solve the overall decision problem by dividing it in many small proof obligations. As in PDR, these are processed piece by piece, constructing the proof incrementally.

1.5 Challenges to Solve

While hardware model checking problems are traditionally formulated in binary logic, programs are usually written in languages were integers are modeled as bit-vectors. As a consequence, the PDR algorithm used in our framework must solve model checking problems formulated over the theory of quantifier-free formulae over bit-vectors (QF_BV). It is well known that QF_BV formulae can be transformed in equivalent Boolean formulae, a method often referred to as *bit-blasting*. However, this process loses all meta-information encoded in the bit-vectors. For QF_BV SMT-solver, using this meta-information allows for substantially faster algorithms [BKO$^+$07]. To conserve the meta-information, a QF_BV generalization of the PDR algorithm is required. Such an algorithm has not yet been known and is one of the main contributions of this dissertation. Two specific challenges are associated with this endeavor:

- A suitable analog of Boolean cubes as atomic reasoning unit of the original PDR algorithm for QF_BV has to be found.

- A suitable generalization of the ternary simulation for bit-vector formulae needs to be devised.

At the end of a successful attempt to solving a model checking instance, the internal state of PDR encodes an invariant that reduces the model checking problem to a combinational verification problem. For our application, we desire that these invariants can be used as loop invariants. This requires a framework that formulates the corresponding model checking instances using the program under verification and uses the acquired knowledge to solve the verification problem.

1.6 Contributions of this Dissertation

This dissertation contains the following contributions:

- A generalization of the PDR algorithm to the theory QF_BV. Instead of simple bit-blasting, the method partially reasons using the integer interpretation of the bit-vectors. Though designed and tuned for the use in our software verification framework, it could be applied to other contexts where the solution of QF_BV model checking instances is required.
- A verification framework that uses the QF_BV PDR algorithm for inferring loop invariants driving a sound and complete algorithm for solving intraprocedural software verification problems in programs with static memory allocation. The overall design of the algorithm is inspired by the PDR algorithm and shares many common properties.
- A method to enhance the presented approach to software verification using loop invariants from other tools.

All presented algorithms have been implemented and their performance have been measured and compared with that of related algorithms.

1.7 Organization of this Dissertation

We organized this dissertation into two parts.

- Part I is concerned with the design of a generalization of the PDR algorithm for the theory QF_BV. Chapter 2 discusses the original PDR algorithm proposed for solving hardware model checking problems and we present our generalization to QF_BV in Chapter 3.
- Part II applies the generalized PDR algorithm introduced in Part I to the software verification problem. Chapter 4 presents our frontend verification algorithm and Chapter 5 discusses how this algorithm relates to previous approaches to program verification. Chapter 6 explains how the presented approach can be strengthened using loop invariants from other sources.

Finally, the conclusion in Chapter 7 summarizes the work presented in this dissertation and reflects on the strengths and weaknesses of the presented approach to software verification as well as potential future directions.

Chapter 2

Property Directed Reachability

For almost a decade, interpolation-based verification [McM03] has been considered the best algorithm to solve hardware model checking problems. In this approach, one repetitively solves bounded model checking instances [BCC$^+$03] with bound k. In case a counterexample is found, one has solved the model checking instance. Otherwise, one derives interpolants [Cra57] from the refutation proof, which act as overapproximations of the forward image of the initial set. Next, the procedure is repeated with the approximate forward image substituted for the initial set. After several iterations, the overapproximations of the forward image often stabilizes, i.e. one finds an inductive invariant that proves the model checking problem. In case one finds a counterexample in one of the repetitions, no conclusion can be made and one increases k which increases the precision of the approximative forward image operator but increases the burden on the backend theory solver.

Recently, a novel approach to hardware model checking has been proposed which attempts to decide a model checking problem stepwise by solving a large number of small lemmata. In contrast to interpolation-based verification, this avoids unrolling the transition relation k times which often yields large SAT instances that cannot be solved efficiently. This algorithm, later named Property Directed Reachability (PDR), has been originally proposed in [Bra11] and its implementation IC3 demonstrated remarkably good performance in the hardware model checking competition (HWMCC) 2010 (third place). The authors of [EMB11] have shown that a more efficient implementation of the algorithm would have won the HWMCC 2010. In particular, the new algorithm outperforms interpolation-based verification on relevant benchmark sets.

In addition to its excellent runtime performance on practical problems, PDR has

additional favorable properties. For instance, it excels in finding counterexamples or in proving that none exist alike; it has modest memory requirements; it is parallizable, a property which has become particularly relevant in recent years; and it is incremental and as such can be initialized with known invariants of the model checking instances, if available.

As a logical consequence, PDR has become subject of active research in the last years. Research efforts have been directed in three principal directions: First, researchers have analyzed the algorithm to better understand the roots of its excellent performance and have evaluated the potential for further algorithmic improvements. Efforts of this kind are e.g. documented in [Bra12] and [SB11]. Second, researchers have attempted to increase the performance of PDR. For instance, the authors of [BR13] propose to use a different set of state variables to speed up verification. Third, researchers have generalized PDR in order to use it in other problem domains. Three such generalizations have been published recently. The authors of [HB12] present generalizations to push-down systems and to the theory QF_LA, the authors of [KJN12] propose an extension of PDR for the verification of timed systems, and the authors of [KMNP13] devised an algorithm based on PDR for infinite state well-structured transition systems such as Petri nets.

The research discussed in this dissertation explores the potential of using PDR for program verification. In the remainder of this chapter, we will describe aspects of the PDR algorithm relevant to our work. We will start with a proper definition of the hardware model checking problem in Section 2.1 and continue with a description of the algorithm in Section 2.2. Finally, we conclude the chapter with a discussion of important characteristics of PDR in Section 2.3.

2.1 Hardware Model Checking Problem

We are given a state space spanned by the domain of n Boolean variables $\mathbf{x} = x_1, x_2, \cdots, x_n$ and define two sets of states: initial states $I(\mathbf{x})$ and bad states $B(\mathbf{x})$ using Boolean formulae over the Boolean variables $\mathbf{x}$. We model our hardware design operating in the state space $\mathbf{x}$ using a transition relation $T(\mathbf{x}, \mathbf{x}')$ that is a Boolean formula over $\mathbf{x}$ and $\mathbf{x}'$, where $\mathbf{x}'$ is a copy of $\mathbf{x}$ which corresponds to the same variables but hold the values for one time step later. The transition relation is true iff the combination $\mathbf{x}$ and $\mathbf{x}'$ represent a possible transition of the hardware design. The problem to be solved is to decide whether a state

in $B(\mathbf{x})$ can be reached from a state in $I(\mathbf{x})$ using only transitions in $T(\mathbf{x}, \mathbf{x}')$. If no bad state is reachable, we will say that the model checking instance holds. Note that for ease of notation, we will omit the dependence of I, T, and B on the variables $\mathbf{x}$ and $\mathbf{x}'$ in the remainder of this dissertation.

2.2 Solving Model Checking Problems with PDR

This section describes PDR to the degree of detail as relevant for this dissertation. Some aspects, a few of them essential for the performance of the algorithm, are omitted. We refer the reader to [EMB11] for a comprehensive discussion.

The conceptual strategy of PDR to solve a given model checking problem is to iteratively find small truth statements, or lemmata, and combine all lemmata to obtain the desired proof for the given problem. The motivation for this strategy is that solving a large number of small problems may be more tractable than attempting to solve one big problem at once.

More concretely, PDR constructs a trace t consisting of frames $f_0, f_1, \cdots$. Each frame contains a set of Boolean cubes $\{c_i\}$, where each cube $c_i = \bigwedge_j l_j$ is a conjunction of Boolean literals l_j where a Boolean literal l_j can either be a Boolean variable x or its negation $\neg x$. If there is a cube c in frame f_i, the semantic meaning of this is that all states contained in c cannot be reached within i steps from the initial states. In this case, we say that the states in c are *covered* in frame i and we will call all cubes in frame f_i the *cover*. The inverse of the cover in f_i is an overapproximation of the states that are reachable in i steps. In frame f_0, a state is covered if it is not reachable in 0 steps, in other words, if it is not in the initial set I.

Algorithm 2.1 shows the overall PDR algorithm. In each iteration, the algorithm searches for a cube in the last frame of the trace that is in B and not yet covered using FINDBADCUBE(). If such a cube c exists, the algorithm tries to recursively cover c (RECCOVERCUBE(), see Algorithm 2.2). To this end, the routine checks if c is reachable from the previous frame. Assume for now that this is the case and denote with $\tilde{c}$ a cube in the previous frame that is not covered and from which c can be reached in one step. Then RECCOVERCUBE() calls itself recursively on $\tilde{c}$. If such a sequence of recursive calls reaches back to frame f_0, the corresponding call stack effectively proves that c can be reached from I, i.e. that the property fails. Otherwise, if a cube c is proved to be unreachable from the

Algorithm 2.1 PDR(I, T, B)

```
1:  while true do
2:      Cube c = findBadCube()
3:      int l = lengthsTrace()
4:      if c then
5:          if !recCoverCube(c, l) then return "property fails"
6:      else
7:          pushNewFrame()
8:          if propagateCubes() then return "property holds"
9:      end if
10: end while
```

Algorithm 2.2 recCoverCube(Cube c, int l)

```
1: if l = 0 then return false
2: while c reachable from c̃ in one transition do
3:     if !recCoverCube(c̃, l − 1) then return false
4: end while
5: expand(c, l)
6: propagate(c, l)
7: add c to F_l
8: return true
```

previous frame, it can be added to the cover of the current frame. For efficiency, however, the algorithm attempts to expand and to propagate the cube to later frames before doing so. Continuing the discussion of Algorithm 2.1, if findBadCube() returns without successfully finding an uncovered cube in B, we know that B is covered in the last frame f_l. As we preserve the invariant that the cover in frame f_i is an underapproximation of the space not reachable within i steps, we can conclude that B is not reachable within l steps and we push a new frame to the end of the trace. Afterwards, we attempt to propagate cubes from frame l to frame $l + 1$. If this is successful for all cubes, we have shown that from an overapproximation of the reachable states in frame f_l we cannot reach any state outside this overapproximation in frame f_{l+1}. In other words, we have found an inductive invariant. Moreover, the set of bad states B is disjoint of this inductive invariant. This

proves that the property holds.

PDR is a sound and complete algorithm for the solving instances of the hardware model checking problem. We already argued why the algorithm gives the correct response upon termination. It remains to show that the algorithm terminates. The complete proof for this result is given in [EMB11]. Herein, we restrict ourselves in pointing out the main intuition of the proof which is based on the following idea: note that every state which is reachable within i steps is also reachable within $i+1$ steps. Hence, the cover of frame f_{i+1} implies the cover of f_i. If two succeeding frames have the same cover, the algorithm terminates. Otherwise, the cover of f_{i+1} must be strictly smaller than that of f_i. As the state space is finite, this implies that the algorithm will eventually terminate. Note, however, that the asymptotic worst case runtime is linear to the size of the state space which is exponential to the size of the problem, i.e. the algorithm has runtime $O(2^n)$ with n being the size of the problem.

PDR makes extensive use of a SAT-solver and a ternary simulator within the individual subroutines:

- To find a bad cube that is not yet covered (subprocedure FINDBADCUBE() on line 2 in Algorithm 2.1, one starts with solving the following SAT instance

$$B \wedge \neg \bigvee_{c_i \in F_l} c_i \tag{2.1}$$

  If the instance is unsatisfiable, there are no more uncovered bad points in the last frame and FINDBADCUBE() returns *null*. Otherwise, the SAT solver returns a satisfying assignment, a cube containing a single state. In the following, one attempts to expand the cube under the condition that all points in the resulting cube remain satisfying (2.1). The expansion sequence is as follows: assume that $c = \bigwedge_j l_j$ is the cube representing the satisfying assignment. The algorithm expands c by iteratively attempting to delete literals from c. An attempt of removing a literal l_k from $c = \bigwedge_j l_j$ is successful if every point in $\bigwedge_{j \neq k} l_j$ remains satisfying (2.1). The condition can be efficiently checked via ternary simulation [EMB11].

- To check whether a cube c in frame l is reachable from an uncovered cube in the previous frame (line 2 in Algorithm 2.2), PDR uses the following SAT instance

$$\neg \bigvee_{c_i \in F_{l-1}} c_i \wedge T \wedge c' \tag{2.2}$$

If the SAT instance is unsatisfiable, c can be added to the trace. With the intention to collect larger cubes, PDR attempts to expand and propagate the cube before doing so. For expansion, it applies the same sequence as for finding bad cubes and checks for each attempt that (2.2) remains unsatisfiable. For propagation, the algorithm increases l in (2.2) and again checks that the SAT instance remains unsatisfiable after doing so.

If the SAT instance is satisfiable, PDR extracts cube $\tilde{c}$ representing the satisfying assignment from which c can be reached in one step. In order to obtain larger proof obligations, PDR attempts to expand $\tilde{c}$. If the transition relation is in functional form, i.e. the next value of each state variable is expression as a function of the current state, this can be done via ternary simulation where the described expansion sequence is used and for each possible expansion, it is checked whether each state reachable from it in one transition is contained in c. If so, the expansion is valid.

We conclude the discussion of the algorithm by illustrating how PDR proves a safety property. Consider the example in Figure 2.1. At the first snapshot, the algorithm already covered part of the bad states B with cube c_1. Now, a call of FINDBADCUBE() returns the remainder of B as a new proof obligation. The subsequent call of RECCOVERCUBE() is successful in covering the proof obligation and adds cube c_2 in f_1 as indicated in the second snapshot. With the new cover, no additional bad and uncovered cube can be found in frame f_1. Thus, a new frame is appended to the trace. The forward propagation succeeds with propagating cube c_1 from frame f_1 to frame f_2 (third snapshot). PDR continues with searching for a bad cube in frame f_2 which is not yet covered and finds the same conflicting cube as previously when covering frame f_1. This time, however, the cube cannot be covered directly as a subset of the proof obligation can be reached from the previous frame f_1 (see fourth snapshot). RECCOVERCUBE() succeeds in resolving the new proof obligation in frame f_1 by adding cube c_3 and also succeeds in propagating the new cube to frame f_2 (fifth snapshot). With this additional cube, the proof obligation in frame f_2 can be resolved by adding cube c_4 to the cover in frame f_2 (sixth snapshot). Now, all bad states in frame f_2 are covered, a new frame is appended to the trace, and PROPAGATECUBES() succeeds with propagating all cubes from f_2 to f_3. An inductive invariant strong enough to prove the safety property is found and the algorithm terminates.

Figure 2.1: Proving a Safety Property with PDR

2.3 Characteristics of PDR

The experimental results reported in [EMB11] show that PDR has excellent performance both on the benchmark set of the HWMCC and a set of industrial benchmarks. For instance, the plot in Figure 2.2 reproduced from data of [EMB11] shows that PDR outperforms interpolation-based verification on the HWMCC benchmarks. This is particularly impressive considering that interpolation-based verification has been subject of optimization for almost a decade whereas PDR is a recent invention. The good performance can be attributed to certain characteristics of PDR.

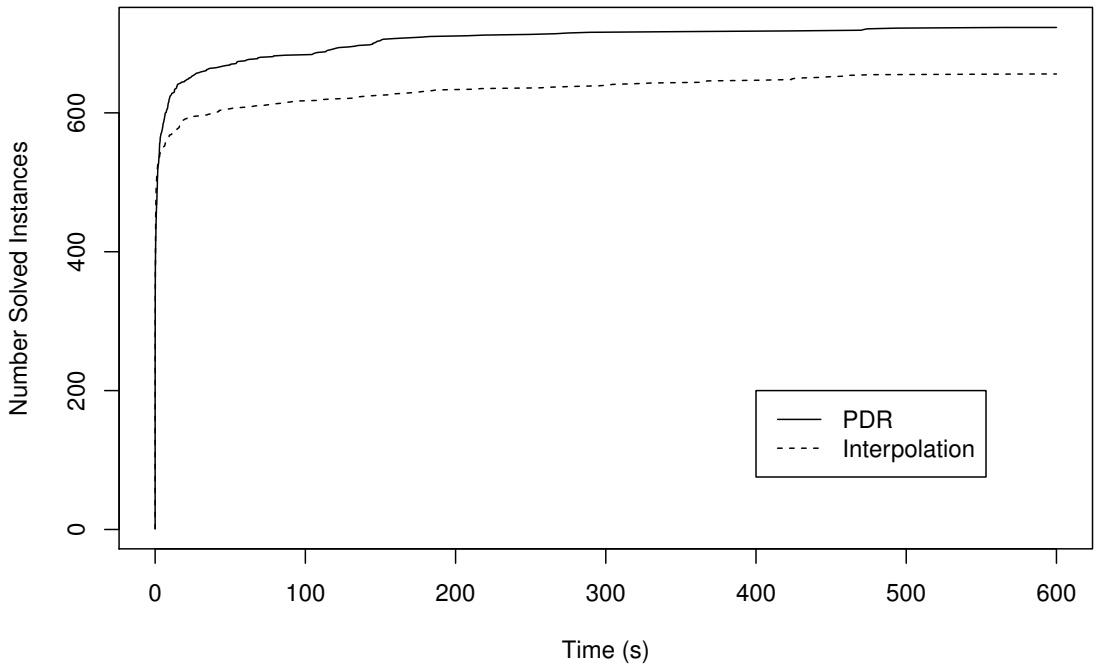


Figure 2.2: PDR vs Interpolation-Based Verification on HWMCC Benchmarks

Firstly, PDR implements a directed *bidirectional search* for an inductive invariant if the model checking instance holds or for a counterexample otherwise. The backward search is targeted at finding a counterexample proving that a bad state is reachable. As such, it resolves proof obligations, i.e. it checks if cubes, either composed of bad states or of states from which bad states are known to be reachable, cannot be reached within a certain number of transitions from the initial set. If this fails, the algorithm found a coun-

terexample for the overall model checking problem. Otherwise, the proof obligation is resolved by covering the proof obligation within the trace. In the propagation phase, the cubes in the cover are attempted to be propagated forward within the trace. This forward movement is designed for finding an inductive invariant and by doing so sieves out too specific cubes that are not inductive. One can summarize that the overall search is steered by the bad states, the backward search directly, the forward search indirectly by the inferred cubes representing the cover of the frames. In other words, the algorithm is *property directed* and hence its apt name given in [EMB11]. As such, the algorithm refrains from wasting runtime by searching for strong invariants in case a weak invariant is sufficient to prove that no bad state is reachable.

Secondly, PDR is an *incremental model checker*. By being committed to cubes as atomic reasoning unit, PDR effectively succeeds in dividing the overall model checking problem into smaller subproblems that are easier to solve. In fact, as pointed out in [Bra12], it allows to construct the proof for solving the model checking instance incrementally, a scheme that is considered favorably in [MP95]. Practically, the incremental nature of PDR provides opportunities for parallelization, effective use of incremental SAT-solvers, and to start the directed search using known facts of the transition systems by initializing the trace.

Thirdly, in contrast to many other model checking algorithms, PDR *abstains from unrolling* the transition relation. This avoids large SAT-solver instances which often require unacceptably long solving times and as consequence yield model checking algorithms that do not scale well.

Fourthly, cubes are an efficient and effective means to represent sets of states. It is efficient because cubes can be stored densely and highly optimized procedures are available for their processing. It is also an effective means because the number of cubes required to represent an inductive invariant is low for most practically relevant hardware model checking instances.

Chapter 3

Generalization of PDR to Theory QF_BV

In Chapter 2, we have discussed the convincing algorithmic properties of PDR, in terms of its superior runtime performance, its low memory consumption, and its potential for parallelization.

Similarly as the excellent algorithmic properties of the DPLL algorithm [DLL62] have fueled interest in generalizing the DPLL algorithm to richer theories [MKS09], the positive characteristics of PDR motivate research for generalizations to richer theories.

In this chapter, we describe a generalization of the Boolean PDR algorithm to the theory of QF_BV. We start in the following section with outlining the overall generalization strategy. In the subsequent two sections, we focus on the two biggest challenges of the generalization, the choice of the atomic reasoning unit, the equivalent of the cube in the Boolean case, and the expansion of proof obligations. Lastly, we conclude the chapter by presenting experimental results demonstrating the overall performance of our generalization of PDR in comparison to the Boolean version and illustrating the impact of several design choices on this performance in Section 3.4.

3.1 Overall Generalization Strategy

Table 3.1 summaries the main aspects of our generalization of the Boolean version of PDR to a more general logic. We assume that the input format of the generalized version are

QF_BV formulae, which motivated the use of a suitable SMT solver instead of a SAT-solver. Using a QF_BV SMT solver, the queries to the SAT-solver in the Boolean version of the algorithm can be transcribed more or less verbatim to constraints for the SMT solver.

	Binary Approach	Generalization
Model Checking Instance I, B, T	Boolean formulae	QF_BV formulae
Backend Solver	SAT-Solver	QF_BV SMT Solver
Atomic Reasoning Unit	Boolean Cubes	Mixed Boolean Cubes and Polytopes
Expansion of Proof Obligations	Ternary Simulation	Mixed Ternary and Interval Simulation

Table 3.1: Summary Generalization PDR

The most challenging problem for the generalization, however, is to find a suitable representation of the atomic reasoning unit (ARU). In Section 3.2, we discuss different choices for the ARU, their advantages and disadvantages, and propose the use of a hybrid solution of Boolean cubes and polytopes.

A second interesting aspect in the quest for a QF_BV PDR algorithm is to find a suitable generalization for the simulation-based expansion of proof obligations. The original version of PDR uses ternary simulation to this end, which appears not to be suitable for a QF_BV version of the algorithm if bit-vectors model integer variables. As indicated in Table 3.1, we propose the use of a mixed ternary and interval simulation. We discuss details of this issue in Section 3.3.

3.2 Choice of Atomic Reasoning Unit

The choice of the ARU is of fundamental importance for the efficiency of PDR. A good representation allows for memory efficient storage and fast processing of the basic operations of the algorithm and at the same time is able to represent typical invariants by using a small number of ARUs. The Boolean version of the PDR algorithm utilizes Boolean cubes as ARUs. As we have discussed in the second chapter of this dissertation, committing to Boolean cubes provides an effective mechanism to divide an overall model checking prob-

lem into smaller subproblems and by doing so yields an incremental scheme to compose the overall inductive invariant. Intuitively, the choice of Boolean cubes appears not to be an excellent choice for a QF_BV generalization of PDR as many trivial bit-vector constraints do not have a simple correspondence as Boolean cubes. For illustration, assume that x is a 4-bit signed bit-vector and that we use the two-complement representation of integers as bit-vectors. In this case, the simple constraint $x \leq 6$ requires at least four Boolean cubes to be represented, e.g.:

$$(1\text{- - -}) \vee (00\text{ - -}) \vee (010\text{ -}) \vee (0110)$$

This suggests a representation of the ARU that is more targeted at representing typical bit-vector constraints. At the same time, however, the choice for the ARU must not be too expressive to preserve the incremental invariant inference scheme and to keep the expansion moves tractable. As indicated in Table 3.1, we propose a mix of Boolean cubes and polytopes as the format of the ARU in our generalization. Originally, we experimented with a simpler representation, integer cubes. For ease of exposition, in the following subsection, we explain the algorithm with integer cubes and illustrate the limitations of this representation. Subsequently, in Subsection 3.2.2, we describe polytopes and how they can be used effectively in PDR to overcome these limitations. Sets of polytopes are an efficient means of representing any piecewise linear inductive invariant. Many relevant problems in the domain of program verification, however, also make use of bit-level operations that yield inductive invariants that are not piecewise linear but can be efficiently represented as sets of Boolean cubes. This suggests a hybrid approach based on Boolean cubes and polytopes; the details of which we discuss in Subsection 3.2.3.

3.2.1 Formulation with Integer Cubes

We now denote with $\mathbf{x} = x_1, x_2, \cdots, x_n$ bit-vector variables. We define an integer cube as a set of static intervals on the domain of these variables. The static intervals are to be interpreted in the conjunctive sense, i.e. a point is in the integer cube iff all variables are in their respective static intervals. As an example, consider the integer cube c defined by $c = (3 \leq x_1 \leq 5) \wedge (-4 \leq x_2 \leq 20)$. The point $x_1 = 4$, $x_2 = 0$ is in c whereas the point $x_1 = 4$, $x_2 = -10$ is not. Geometrically, an integer cube corresponds to an orthotope (a.k.a. hyperrectangle) in the n-dimensional space.

The definition of integer cubes as ARUs is a straightforward generalization of

the concept of Boolean cubes and allows for an efficient implementation of many frequent subroutines of the algorithm, such as checking for implication. The expansion operation is also straightforward: instead of skimming Boolean literals as in the formulation with Boolean cubes, we attempt to increase the intervals of the variables by decreasing lower bounds and increasing upper bounds using binary search. In the theoretical sense, the fact that all inductive invariants can be represented by a union of integer cubes appears to be promising.

We illustrate the operation and limitations of the algorithm using integer cubes as ARUs using two examples.

Example: Simple

Consider PDR was called with the following model checking problem in which B is unreachable.

$$\begin{aligned} I &:= (n \equiv 1) \wedge (x \equiv 0) \\ T &:= (n > 0) \wedge (x' \equiv x + 1) \wedge (n' \equiv n - 1) \\ B &:= (x \geq 3) \end{aligned}$$

Figure 3.1 illustrates how a simplified version of PDR with integer cubes as ARUs would prove this fact.

Initially, the trace has only one frame, f_0. As $B \wedge I = \text{false}$, B is not reachable in zero steps and f_1 is pushed at the end of the trace. In f_1, FINDBADCUBE() returns integer cube $x \geq 3$ that is in B and uncovered (indicated by the red rectangle in the first snapshot of Figure 3.1). Next, PDR checks whether there are points in $x \geq 3$ that are reachable from the initial set in f_0 in one transition. This is not possible, hence $x \geq 3$ can be covered. Before being added to the cover of f_1, the cube is expanded, yielding cube c_1 covering $x \geq 2$ as in the second snapshot of Figure 3.1. After adding c_1 to the cover, there are no longer uncovered points in B and f_2 is pushed at the back of the trace. Now, PDR attempts to propagate integer cube c_1 to f_2. This is not possible, however, because from the overapproximation of the reachable set in f_1 (the inverse of the cover) one can reach points in $x \geq 2$ in one transition. After the propagation phase, PDR continues with finding uncovered cubes in B in f_2, yielding $x \geq 3$ for another time (see trace in the second snapshot of Figure 3.1). No point in $x \geq 3$ can be reached from the reachable set in f_1 and cube c_2 is added to the

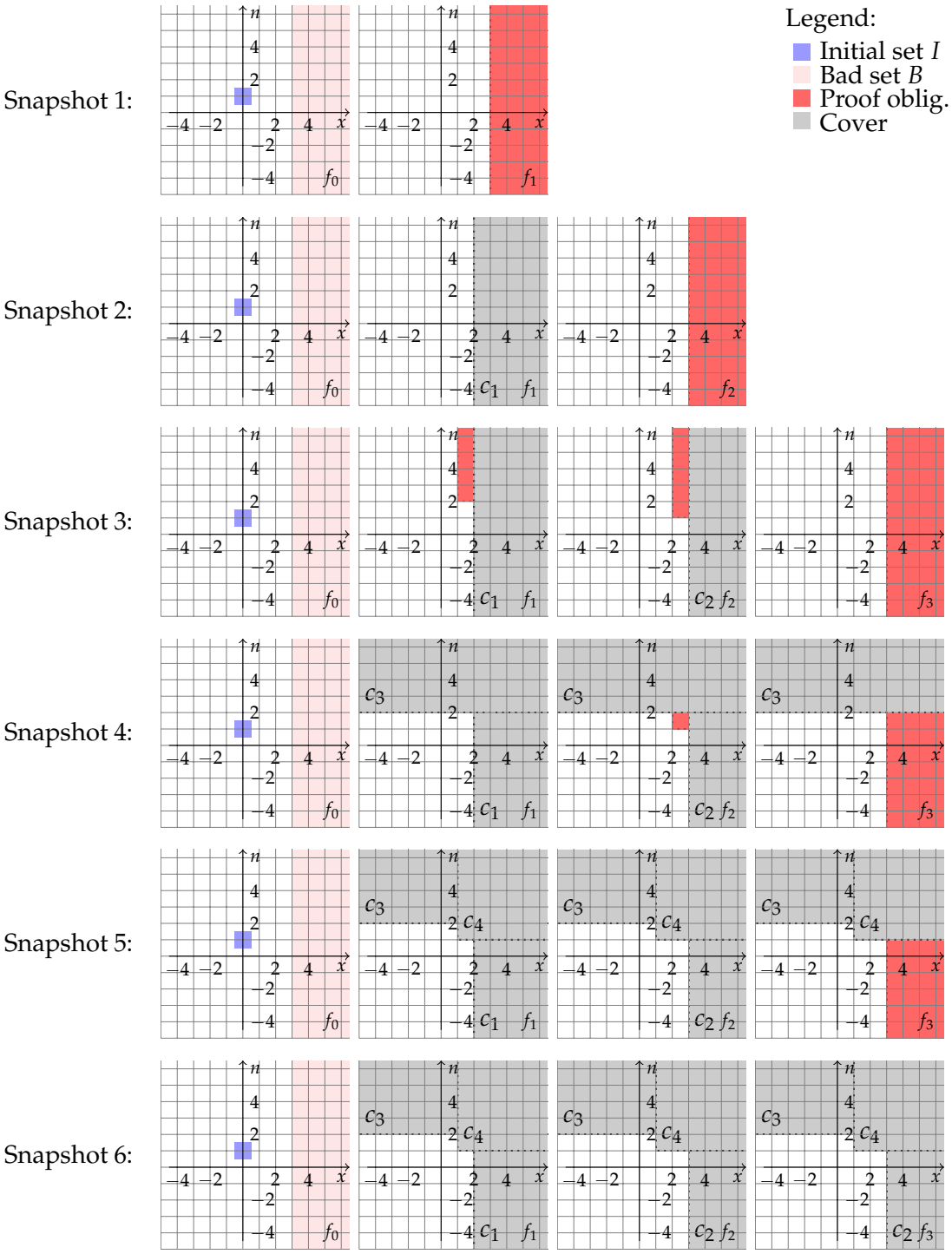


Figure 3.1: Iterative Construction of Proof for Example Simple

cover in f_2. Now, B is covered completely and PDR pushes f_3 in the end of the trace. In the subsequent propagation phase, the attempt to propagate c_2 fails. The following call of FINDBADCUBE() returns again $x \geq 3$ (see third snapshot in Figure 3.1 in f_3). This time, however, points in f_3 can be reached from the uncovered region of f_2. For instance, one can reach $x = 3, n = 0$ from $x = 2, n = 1$ in f_2 in one transition. Using simulation-based expansion, one can expand this new proof obligation to $2 \leq x < 3 \wedge 1 \leq n$. Points in this region can also be reached from the previous frame, yielding an additional proof obligation $1 \leq x < 2 \wedge 2 \leq n$ in f_1 (see third snapshot in Figure 3.1). No point in this cube can be reached from the initial set in f_0. Hence, a new cube (c_3) covering the area is generated, expanded to $n \geq 2$, and added to f_1. In the sequel, PDR also attempts to propagate c_3 to the next frames and realizes that this is in fact possible. Therefore, cube c_3 is also added in frames f_2 and f_3 (see fourth snapshot in Figure 3.1). As a consequence of adding cube c_3 to f_1, all points in $2 \leq x < 3 \wedge 1 \leq n$ in f_2 cease to be reachable from f_1. After expansion, this yields cube c_4 in the cover of f_2. As with cube c_3, this cube cannot be reached in any later frame, hence it is propagated as well. Also note that if a cube cannot be reached within two steps, it can neither be reached within one step. Hence, c_4 can also be considered covered in f_1 (see fifth snapshot in Figure 3.1). As a consequence of adding c_4 to the cover of f_2, $x \geq 3$ in f_3 ceases to be reachable and allows to cover B completely. Note that transitions such as $x = 2, n = -2$ to $x = 3, n = -3$ are invalid by the constraint $n \geq 0$ in the transition relation. We obtain the cover in the sixth snapshot in Figure 3.1. In this snapshot, the covers in f_2 and f_3 are identical. This means that no point in the overapproximation of the reachable set in f_2 can reach any state outside this overapproximation, i.e. we have found an inductive invariant proving that B is unreachable.

Example: Linear Invariant

Consider now the following model checking problem

$$\begin{aligned} I &:= (x + 2y \leq 5) \\ T &:= (x' \equiv x + 1) \wedge (y' \equiv y - 1) \wedge (x' > x) \wedge (y' < y) \\ B &:= (x + 2y > 5) \end{aligned}$$

Note that the initial condition is preserved by the transition relation and serves itself as an inductive invariant to prove that B is unreachable. Also note that the last two conjuncts in

the transition relation serve to prevent potential overflows.

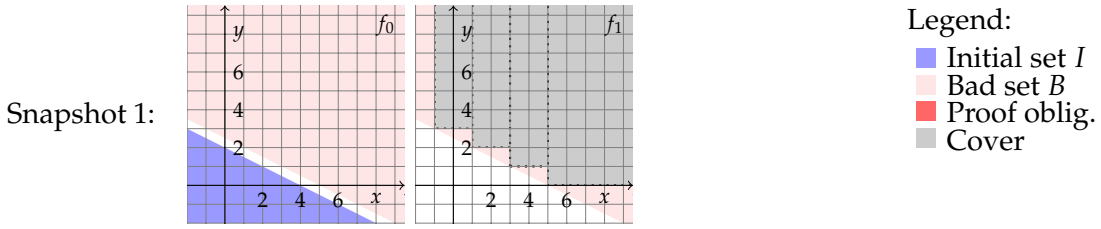


Figure 3.2: Attempt to Solve Example Linear Invariant

For this example, the formulation of PDR using integer cubes is not able to construct an inductive invariant efficiently. The trace in Figure 3.2 which was recorded after a couple of iterations of the main loop illustrates this fact. For each integer on the line $x + 2y = 5$, one needs an integer cube to cover the bad area entirely. Assuming that the variables are 32-bit integers, this means that PDR needed to add 2^{31} cubes.

In general, if the inductive invariant required to decide a model checking problem contains a relation between two or more variables, it is not possible to represent the inductive invariant efficiently using integer cubes. Inductive invariants that relate variables are common in many practical applications of our model checker, which strongly suggests that integer cubes are insufficiently expressive as ARUs.

3.2.2 Formulation with Polytopes

The limitations of the algorithm with integer cubes discussed in the previous section suggest the need of a more expressive ARU.

Instead of integer cubes, we consider polytopes as ARUs now. Mathematically, a polytope can be represented as a system of linear inequalities $\mathbf{Ax} \leq \mathbf{b}$. With this representation, the algorithm naturally generalizes to cope with polytopes. For the expansion move, we iterate over each individual boundary and attempt to relax an inequality $\sum_j a_{ij} x_j \leq b_i$ by increasing the right-hand-side b_i. As in the case of integer cubes, we find the largest b_i using binary search. Conceptually, using polytopes as the ARU permits us to represent any piecewise linear invariant efficiently. For instance, the inductive invariant in the second example can be represented using the single polytope $x + 2y > 5$. In comparison to integer cubes, the gain in expressiveness associated with polytopes comes

with a slight decrease of the efficiency of frequently used atomic operations in PDR, such as checking for implication.

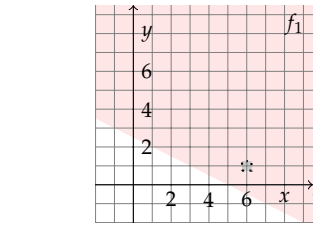
In the current formulation of the algorithm, however, only polytopes composed of unary inequalities can be found. To see why, consider that FINDBADCUBE() initially obtains a single point $x_1 = c_1, x_2 = c_2, \ldots$ in the state space as a result of a call to the backend SMT solver. This point is expressed as a polytope using sets of inequalities of the form

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & -1 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 1 & 0 & 0 & \ldots \\ 0 & 0 & 0 & -1 & 0 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_1 \\ x_2 \\ x_2 \\ \vdots \end{pmatrix} \leq \begin{pmatrix} c_1 \\ -c_1 \\ c_1 \\ -c_1 \\ \vdots \end{pmatrix} \tag{3.1}$$

The defined expand operation allows to increase the volume of the polytope by increasing the right-hand-sides of the inequalities, i.e. the values of c_i or $-c_i$ in equation (3.1). Geometrically, this moves the limiting hyperplanes parallelly outward but does not allow for changing the principal shape of the polytope that remains being an integer cube. This limitation is illustrated in Figure 3.3, where FINDBADCUBE() returns the single point $x = 6, y = 1$ as illustrated in Figure 3.3a. The first expansion move attempts to increase the right-hand-side of the inequality $x \leq 6$ and is able to increase it arbitrarily, i.e. removes the inequality altogether (see Figure 3.3b). Similarly, the third inequality $y \leq 1$ is removed in the second expansion move (see Figure 3.3c). In the third expansion move, the right-hand-side of the second inequality is increased to -3. Thereafter, the fourth inequality cannot be relaxed. We arrive at the expanded polytope $x \geq 3, y \geq 1$ as illustrated in Figure 3.3d.

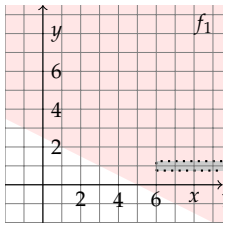
We can conclude that in the current formulation of the algorithm, polytopes are only used to encode integer cubes. Hence, we require an additional mechanism in our algorithm to use the added expressiveness of polytopes. We define a new operation, RESHAPE(), which is geared towards resolving this problem. In the recursive covering procedure discussed in Chapter 2, RESHAPE() is called after propagation and is followed by an additional expansion (see Algorithm 3.1).

Mathematically, the purpose of this reshape-operation is to increase the number of terms within the boundaries of the polytope. Initially, we only have unary boundaries, such as $a_i x_i \leq b_i$. One solution towards finding boundaries with higher arity would be to add an additional variable to obtain a boundary of the form $a_i x_i + a_j x_j \leq b_i$ and run a



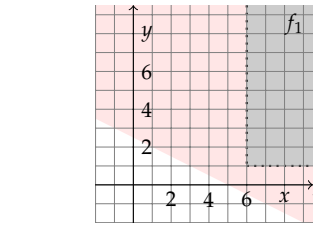
$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 6 \\ -6 \\ 1 \\ -1 \end{pmatrix}$$

(a) Before Expansion



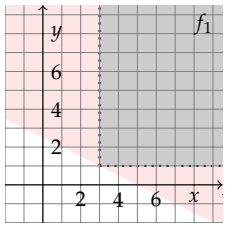
$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -6 \\ 1 \\ -1 \end{pmatrix}$$

(b) After First Expansion Step



$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -6 \\ -1 \end{pmatrix}$$

(c) After Second Expansion Step



$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -3 \\ -1 \end{pmatrix}$$

(d) After Expansion

Figure 3.3: The defined expansion operation cannot change the shape of the polytopes.

search algorithm to find values for a_i and a_j that are maximum with respect to a suitable optimization criterion such as the volume of the polytope. However, considering the large search space of this operation, this approach is unlikely efficient.

Instead, we propose a more targeted approach. The principal idea of our reshape-operation is to use information from several polytopes to make guesses for possible new boundaries. Then, one attempts to substitute these new boundaries for existing ones of lower arity.

Algorithm 3.1 RECCOVERPOLYTOPE(Polytope p, int l)

1: **if** $l = 0$ **then return false**
2: **while** p reachable from $\tilde{p}$ in one transition **do**
3: **if** !RECCOVERPOLYTOPE($\tilde{p}, l-1$) **then return false**
4: **end while**
5: EXPAND(p, l)
6: PROPAGATE(p, l)
7: RESHAPE(p, l)
8: EXPAND(p, l)
9: add p to F_l
10: **return true**

Example: Linear Invariant (revisited)

Consider our second example for another time. In Figure 3.4, we illustrate how the reshape-operation would proceed after the second polytope has been found. Snapshot 1 displays the situation right before the call of the reshape-operation. Assume that RESHAPE() is called on the striped polytope (pivot, a_1). The other polytope a_2 acts as guide. The situation suggests that the line defined by the lower-left corners of the two polytopes might be a good candidate as a new boundary. Hence, RESHAPE() calculates this line and substitutes it for one of the neighbor boundaries of the corner of a_1. In the second snapshot, the new boundary was substituted for $x \geq 1$. Next, RESHAPE() checks if polytope a_1 is still unreachable from the previous frame after the substitution. This is the case for the given example, so the substitution is kept. After the reshape-operation, EXPAND() is called once more which eliminates the other neighbor boundary of a_1 (see third snapshot). Note that guide a_2 is now subsumed by the reshaped pivot, hence will be discarded. More importantly, the reshaped polytope covers B completely and is inductive invariant, hence the model checking instance is solved.

General Reshape Algorithm

Details of the general algorithm are given in Algorithm 3.2. If called on a polytope, the routine iterates through all corners and finds for each promising set of guides G a corresponding hyperplane. The efficiency and effectiveness of RESHAPE() depends critically on the

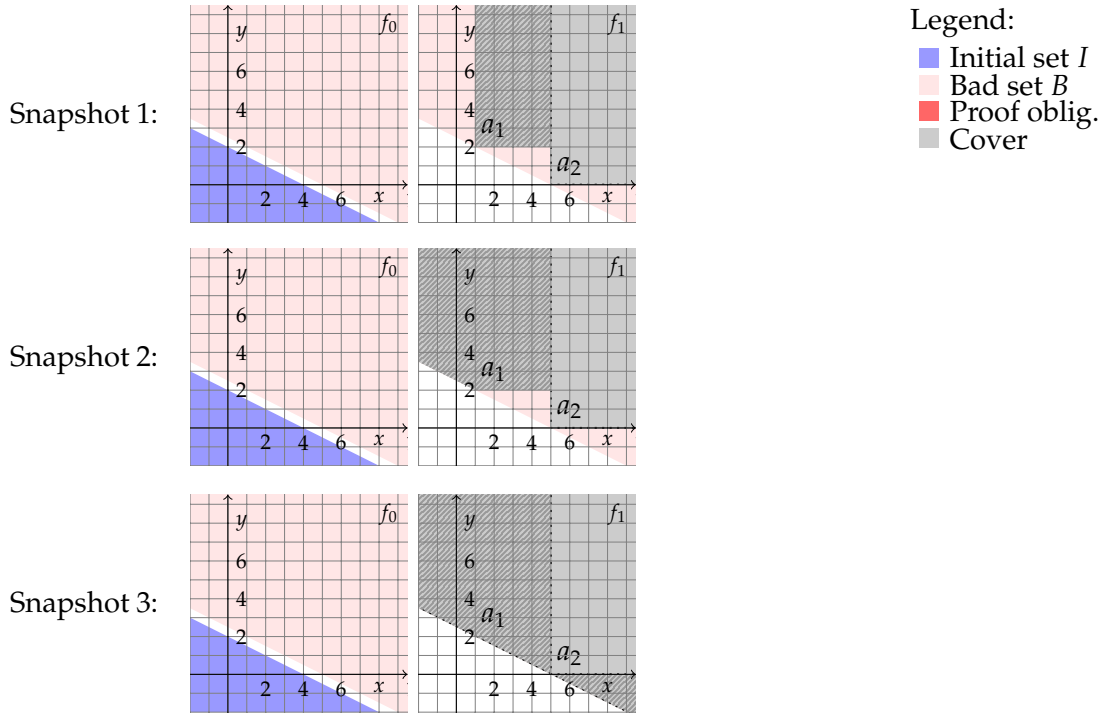


Figure 3.4: Attempt to Solve Example Linear Invariant with Polytopes

choice of guides. With respect to effectiveness, iterating through all possible sets is clearly the optimal choice. However, the number of sets grows exponentially with the number of other polytopes that may act as guide. Consequently, a smaller choice guided by heuristics is necessary. Our experimentation suggested that sets of guides that are close to p are most likely to yield a successful substitution. After a set of guides G is fixed, we calculate the hyperplane h defined by the corners of the guides and iteratively attempt to substitute it for a neighbor boundary b of the corner in p. If such a substitution yields a polytope that is reachable from the previous frame, we reverse the substitution and continue with our attempt to reshape p by substituting h for another neighbor boundary. Otherwise, we bail out, keeping the substitution and continue with trying to reshape another corner of p.

The reshape-operation as described in Algorithm 3.2 is able to substitute a $(k+1)$-ary boundary for a unary one. If n is the number of dimensions of the state space, we can choose $k = n - 1$ and the algorithm can principally find hyperplanes relating all variables with each other. This comes at the expense of an asymptotic running time that grows exponentially with parameter k. In problem domains which only require linear invariants

Algorithm 3.2 RESHAPE(Polytope p, int l)

```
1:  for each corner c of p do
2:      for each promising set G of k guides do
3:          h = FINDHYPERPLANE(c, p, G)
4:          for each neighbor boundary b of p w.r.t. c do
5:              SUBSTITUTE(p, h, b)
6:              if p reachable from f_{l-1} then SUBSTITUTE(p, b, h)
7:              else break 2
8:          end for
9:      end for
10: end for
```

that relate less than n variables with each other, a smaller value for k should be chosen for a better runtime performance. For instance, in the linear invariant example, we only require a binary linear invariant and we could chose $k = 1$. In this case, the number of promising sets $|G|$ grows linearly with the number of polytopes that may act as guides.

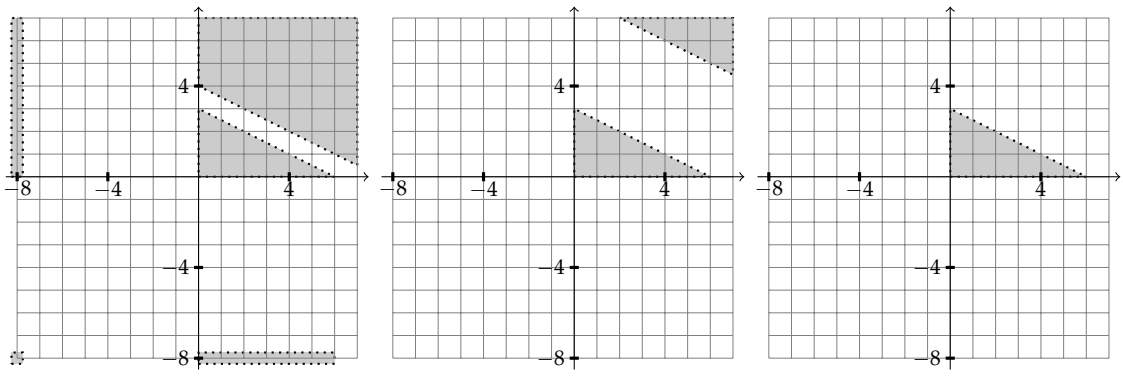
Bitwidth Extension of Polytope Expressions

Overflows are a practical problem associated with the use of QF_BV SMT solver in conjunction with the inequality constraints of which polytopes are composed of. To illustrate the issue, consider the following polytope composed out of three constraints over the 4-bit signed bit-vector variables x and y

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 6 \end{pmatrix} \tag{3.2}$$

If interpreted in the theory QF_BV with the two's complement representation, all points in the shaded regions of Figure 3.5a are contained in this polytope. This contradicts the standard arithmetic interpretation of the constraints which corresponds to the geometry displayed in Figure 3.5c. The origin of the discrepancy are overflows occurring if the polytope boundary inequalities are evaluated in bit-vector arithmetic. Consider the point $x = 4, y = 2$. As expected, the first two inequalities in (3.2) hold. However, evaluating the third inequality $x + 2y \leq 6$ with these values yields true due to an overflow occur-

ring when the summation $4 + 4$ on the left-hand-side of the inequality is evaluated. For $x = -8, y = 4$, the first inequality in (3.2) holds unexpectedly because the unary minus operation applied to the most negative number (here -8) yields the most negative number in the two's complement representation of bit-vectors which is certainly determined to be negative.



(a) Without Bit Extension (b) 1-bit Extension (c) 2-bit Extension

Figure 3.5: Interpretation of Polytopes using Bit-Vector Arithmetic

Though the interpretation of polytope constraints in bit-vector arithmetic does not impact the correctness of the defined PDR algorithm with polytopes (any consistent interpretation of the ARUs will yield a sound algorithm), it can have serious impact on its efficiency as it is not in accordance with the geometric interpretation of polytopes which originally motivated their use.

The problem can be alleviated by increasing all bitwidths of intermediate expressions of the inequality constraints. The more bits are appended, the less likely overflows impact the interpretation of the polytope. For instance, if we extend the bitwidths of the variables in the given example by one, we obtain the interpretation in Figure 3.5b which no longer contains the overflows pertaining the unary minus operation in first two inequalities of the polytope. Extending the bitwidths of the variables by two, we obtain the desired geometric interpretation of the constraints as illustrated in Figure 3.5c.

In our experimentation, we interpreted the polytope constraints using three additional bits and have no longer observed any adverse impact of overflows in the evaluation of polytope inequalities on the efficiency of the algorithm.

3.2.3 Hybrid Approach

The QF_BV PDR algorithm with polytopes as ARUs is well motivated when the inductive invariant is representable efficiently as a set of piecewise linear inequalities. As such, however, it does not outperform the original PDR algorithm with Boolean cubes on all benchmarks as shown in the next example.

Example: Hybrid Invariant

For illustration, consider the following model checking problem

$$\begin{aligned} I &:= (2 \times y \equiv x) \wedge (x + y \leq 3) \\ T &:= (y' \equiv y + 1) \wedge (x' \equiv x - 2) \wedge (y' > y) \wedge (x' < x) \\ B &:= (x + y \geq 4) \vee (x \bmod 2 \equiv 1) \end{aligned}$$

The snapshot in Figure 3.6 illustrates the issue the QF_BV PDR algorithm with polytopes as ARUs encounters when attempting to solve this model checking instance.

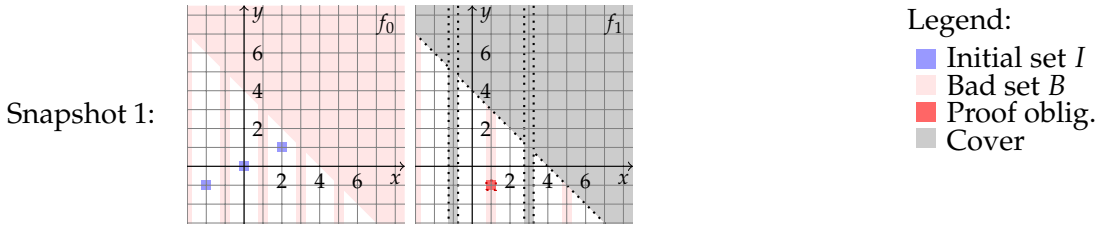


Figure 3.6: Attempt to Solve Example Hybrid Invariant

The inductive invariant of this system required to solve the instance is $(x + y \leq 3)$ $\wedge$ (x even). The corresponding cover in the PDR trace includes two parts, '$x + y \geq 4$' and 'x odd'. The first part can be covered using a polytope. However, the second part can not be efficiently represented using polytopes. Instead, the algorithm would add a polytope for each odd number representable by x. Assuming that x is a 32-bit bit-vector, this would require 2^{31} polytopes, causing the overall algorithm to be inefficient.

In contrast, the original PDR algorithm would efficiently find this second part of the inductive invariant as it can be represented by a single Boolean cube but fail in finding the first part efficiently.

	Original Formulation	QF_BV Polytopes	QF_BV Hybrid
Atomic Reasoning Unit	Boolean Cubes	Polytopes	Boolean Cubes and Polytopes
Strengths	logic invariants	arithmetic invariants	arithmetic and logic invariants
Weaknesses	arithmetic invariants	logic invariants	-

Table 3.2: Strengths and Weaknesses of Different Choices for ARUs

The shortcomings of using either exclusively Boolean cubes or polytopes as summarized in Table 3.2 motivate a hybrid approach. To this end, we define the ARU to be either a Boolean cube or a polytope. PDR constructs a cover as a union of ARUs. As such, the coexistence of ARUs of different kinds does not represent a conceptional problem.

Each proof obligation is initially a point in the state space found by the SMT solver. A point in the state space can be interpreted either as a Boolean cube or a polytope. A specialization becomes necessary before the proof obligation is expanded as the expansion sequences for Boolean cubes and polytopes are different. The decision of whether one specializes a point to a Boolean cube or a polytope can be crucial and an effective strategy for specialization is in order.

Probabilistic Specialization of Proof Obligations

We propose to specialize points to a specific kind of ARU probabilistically. In the setup with Boolean cubes and polytopes, we specialize a point to a Boolean cube with probability c and to a polytope otherwise.

This probabilistic specialization guarantees that a favorable decision can be expected to be chosen in a constant number of attempts. To see why, consider the example in Figure 3.6 for another time. With probability c, the marked proof obligation is specialized to a Boolean cube. In this case, the desired part 'x even' of the inductive invariant would be found immediately. Otherwise, the proof obligation would be covered by the polytope $x = 1$. In this case, after another call to FINDBADCUBE(), another proof obligation with x

odd is found followed by another probabilistic decision for specialization. The probability that a favorable choice is made in the i^{th} iteration is $c(1-c)^{i-1}$. Hence, the expected number of trials until the favorable specialization is found can be calculated to be

$$E\{\text{Trials until Boolean cube specialization}\} = c\sum_{i=1}^{\infty} i(1-c)^{i-1} = \frac{1}{c}$$

Analogously, one calculates

$$E\{\text{Trials until polytope specialization}\} = c\sum_{i=1}^{\infty} i(1-c)^{i-1} = \frac{1}{1-c}$$

Consequently, as long as $c = (0,1)$, one can expect a favorable decision within a constant number of times. The optimal choice for c depends on the concrete model checking instance. We will investigate this aspect in the experimental section of this chapter (see 3.4.2).

3.3 Expansion of Proof Obligations

The excellent performance of the binary version of PDR as documented in [EMB11] can be partly attributed on its ability to expand proof obligations using ternary simulation. In this section, we describe a generalization of this approach for our QF_BV PDR solver.

Proof obligations are expanded after their specialization. At this time, a proof obligation consists of a single point in the state space only. The aim of expansion is to add additional points to the proof obligation, forming a set of points, while preserving the property that if any point in the set is reachable, we proved that the model checking instance does not hold. In this case, the found set of points can be processed simultaneously after expansion stimulating more abstract reasoning. Note that expansion of proof obligations, albeit practically critical for performance, is not required for the correctness of the algorithm. In general, expanding a proof obligation to contain the maximum number of points takes time exponential to the size of the proof obligation [EMB11]. As a consequence, it is essential to find an efficient approximate alternative. An underapproximation of the maximum possible expansion that can be calculated efficiently but at the same time yields good expansions in practice is a prudent choice. On the other hand, it is important to assure that the result does not contain points from which B is not reachable. Otherwise, the overall algorithm could eventually report spurious counterexamples, i.e. the algorithm would no longer be sound.

The backbone for expansion of proof obligations is simulation of expressions. In the following, we will denote with $\Phi_e(a)$ an overapproximation of the values expression e can take under the constraint that variables take values that are contained in ARU a only. For example, consider the case that we have $e = y_1 \vee y_2$ over the two 4-bit variables y_1 and y_2 and as ARU the following Boolean cube $a := (y_1 \in \text{- 0 0 -}) \wedge (y_2 \in \text{1 0 0 -})$. In this case, e can take any value in cube 1 0 0 -. Consequently, both 1 0 0 - and - 0 0 - are valid valuations for $\Phi_e(a)$. On the other hand, - - - 0 is not valid as it does not contain 1 0 0 1.

Expansion of proof obligations is used in two contexts. First, for expanding ARUs in B that are not yet covered (line 2 in Algorithm 2.1) and second for expanding ARUs from which another proof obligation is reachable in one step (line 3 in Algorithm 2.2). In the first context, we test whether the proposed expansion to $\tilde{a}$ is valid by checking if

$$\Phi_{B \wedge u}(\tilde{a}) = \text{true} \tag{3.3}$$

where B is an expression describing the bad set of the model checking instance and u is the expression describing the points that are not yet covered. In case the equation holds, the expansion is valid.

In the context of expanding an ARU a from which another proof obligation a' is reachable in one step, we check whether the possible valuations of the variables after one step starting from $\tilde{a}$ are included in the possible valuations of the corresponding variables under a', formally

$$\forall x_i . \Phi_{\text{next}_{x_i}}(\tilde{a}) \subseteq \Phi_{x_i}(a') \tag{3.4}$$

where we denote with next_{x_i} the expression which captures the computation of the next value of x_i. Note that this requires that the transition relation is in a format that allows to calculate the values for the next state. In many applications, this is naturally the case.

Note that, under the assumption that the simulation values $\Phi_e(a)$ represent over-approximations, the checks (3.3) and (3.4) are conservative in the sense that an expansion to $\tilde{a}$ is accepted only if a bad state is reachable from all points in $\tilde{a}$.

3.3.1 Ternary Simulation

The binary PDR algorithm uses ternary simulation to calculate simulation values for expressions. In the following, we will use logic inference rule notation of the form

$$\frac{p_1 \quad p_2 \quad \cdots \quad p_n}{c}\ [\text{rule}]$$

to represent simulation rules where conclusion c can be inferred if the premises p_1, p_2, ..., and p_n hold. Representing binary variables as intervals $[0,0]$, $[0,1]$, and $[1,1]$ which stand for false, false or true (X), and true, respectively, one can use the rules in Figure 3.7 to obtain simulation values for any expression composed of AND- and INV-operations. As

$$\frac{}{\Phi_c = [c,c]}\ [\text{const}] \qquad \frac{}{\Phi_X = [0,1]}\ [\text{x}] \qquad \frac{l \le v \le u}{\Phi_v = [l,u]}\ [\text{var}]$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2]}{\Phi_{e_1 \wedge e_2} = [\min\{l_1,l_2\}, \max\{u_1,u_2\}]}\ [\text{and}] \qquad \frac{\Phi_e = [l,u]}{\Phi_{\sim e} = [1-u, 1-l]}\ [\text{inv}]$$

Figure 3.7: Simulation Rules for Ternary Simulation

one can decompose any bit-vector expression into equivalent AND-Inverter expressions using their circuit representations [KK97], this allows to calculate the simulation values required in checks (3.3) and (3.4).

It remains to discuss how the range of variables is restricted given an ARU a. This is straight-forward if the ARU is a Boolean cube. The situation is slightly more complicated if the ARU is a polytope: It is instructive to make two observations. First, note that at the beginning of an expansion move, a represents a point. Second, any proposed expansion for a polytope is derived by relaxing individual inequalities of the polytope. Combining these two observations, we can conclude that at any step of an expansion sequence, the polytope encodes a static integer interval for each variable. For a bit-vector variable x of m bits, let $[l,u]$ be the interval x can take while staying in a. With i being the index of the bit we are interested in, we can use SELECT(l,u,i) in Algorithm 3.3 to obtain the corresponding range of values the bit can take. Note that in Algorithm 3.3, we denote with $l[i:m-1]$ the vector of the $m-i$ most significant bits of l if l was stored in a bit-vector of length m. The definition of $u[i:m-1]$ is analog.

Algorithm 3.3 SELECT(l,u,i)

```
1: if l[i : m − 1] ≡ u[i : m − 1] then
2:     return [l[i], l[i]]
3: else
4:     return [0,1]
5: end if
```

The idea of Algorithm 3.3 is as follows: let j be the lowest index of the bit-vectors where $l[j : m-1]$ and $u[j : m-1]$ coincide. For all bits $i \geq j$, we can assert that the corresponding bit $x[i]$ can only take the value common in $l[i] = u[i]$ because all more significant bits in the two's complement encoding have the same values. For all other bits $i < j$, $x[i]$ can be either true or false because there is at least one more significant bit that differs. For example, assume that a polytope constrains that 4-bit variable x must take values in $[3, 5]$. In two's complement representation, that corresponds to $l = 0\ 0\ 1\ 1$ and $u = 0\ 1\ 0\ 1$. The index of the most significant bit that differs is $j = 2$. Hence, we have e.g. that SELECT$(3, 5, 3)$=[0,0] and SELECT$(3, 5, 0)$=[0,1].

We can conclude that ternary simulation can be used for the expansion of proof obligations for the QF_BV generalization of the PDR algorithm. However, expansion of proof obligations using ternary simulation cannot be expected to perform effectively if used for a model checking instance where the transition function captures high-level constraints where bit-vectors are interpreted as integers. In particular, using ternary simulation, the only way to represent any interval that contains both positive values and negative values is by assigning X to all bits, causing gross overapproximation of simulation values in practice.

3.3.2 Interval Simulation

The limitations of ternary simulation suggests a simulation method that captures integer operations more naturally. One way to achieve this is integer simulation. Consider the representative choice of simulation rules in Figure 3.8 where we denote the minimally and maximally representable number for a given expression with $-\infty$ and ∞, respectively.

The simulation rules model all operations conservatively where conservatively means that they overapproximate the set of values an expression can take. For instance, rule [plus-r] models an addition in case there is no overflow only, enforced by the antecedents $l_1 + l_2 \geq -\infty$ and $u_1 + u_2 \leq \infty$. In case there is an overflow, the result is given the interval $[-\infty, \infty]$ as specified in rule [plus-o].

In the case of interval simulation, restricting the range of variables given an ARU a is immediate if a is a polytope. Otherwise, if a is a Boolean cube, constraining the m bits of x such as $x[i] \subseteq [l_i, u_i]$, we can use procedure CONJOIN$(l_0, u_0, \ldots, l_{m-1}, u_{m-1})$ in Algorithm 3.4 to calculate a suitable interval for bit-vector x.

$$\frac{\Phi_e = [l,u] \quad l \neq -\infty}{\Phi_{-e} = [-u,-l]}\ \text{[uminus-r]} \qquad \frac{\Phi_e = [l,u] \quad l \equiv -\infty}{\Phi_{-e} = [-\infty,\infty]}\ \text{[uminus-o]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 + l_2 \geq -\infty \quad u_1 + u_2 \leq \infty}{\Phi_{e_1+e_2} = [l_1 + l_2, u_1 + u_2]}\ \text{[plus-r]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 + l_2 < -\infty \vee u_1 + u_2 > \infty}{\Phi_{e_1+e_2} = [-\infty,\infty]}\ \text{[plus-o]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad \bigwedge_{i,j\in\{1,2\}}(-\infty \leq l_i \times u_j \leq \infty)}{\Phi_{e_1\times e_2} = [\min_{i,j\in\{1,2\}}\{l_i \times u_j\}, \max_{i,j\in\{1,2\}}\{l_i \times u_j\}]}\ \text{[times-r]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad \bigvee_{i,j\in\{1,2\}}(l_i \times u_j < -\infty \vee l_i \times u_j > \infty)}{\Phi_{e_1\times e_2} = [-\infty,\infty]}\ \text{[times-o]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad u_1 < l_2}{\Phi_{e_1<e_2} = [1,1]}\ \text{[lt-1]} \qquad \frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 > u_2}{\Phi_{e_1<e_2} = [0,0]}\ \text{[lt-0]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad u_1 \geq l_2 \quad l_1 \leq u_2}{\Phi_{e_1<e_2} = [0,1]}\ \text{[lt-x]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad l_1 \equiv 1}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = \Phi_{e_2}}\ \text{[ite-t]} \qquad \frac{\Phi_{e_1} = [l_1,u_1] \quad u_1 \equiv 0}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = \Phi_{e_3}}\ \text{[ite-e]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad \Phi_{e_3} = [l_3,u_3] \quad l_1 \neq u_1}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = [\min\{l_2,l_3\}, \max\{u_2,u_3\}]}\ \text{[ite-x]}$$

Figure 3.8: Choice of Simulation Rules for Interval Simulation

Algorithm 3.4 CONJOIN($l_0, u_0, \ldots, l_{m-1}, u_{m-1}$)

1: l, u = bit-vector with length m
2: **for** $i = 0$ to $m - 1$ **do**
3: **if** $l_i \equiv u_i$ **then**
4: $u[i] = l[i] = l_i$
5: **else**
6: $l[i] = \begin{cases} 1 & \text{if } i \equiv m-1 \\ 0 & \text{otherwise} \end{cases}$
7: $u[i] = 1 - l[i]$
8: **end if**
9: **end for**
10: **return** $[l, u]$

Algorithm 3.4 constructs the bounds for variable x by iterating through all bits of x. If the bounds for a bit are equal, the corresponding bit in the bound l, u is set to the common value. Otherwise, the corresponding bit in l is set to the value which causes l to represent a smaller value in the two's complement representation. The corresponding bit in u is set to the value which causes u to represent the bigger value. For instance, assume we have a Boolean cube $a := (x \in \text{- - 0 1})$. Here, the valuation of the bit-vector with the smallest (greatest) value in two's complement representation is 1 0 0 1 (0 1 0 1). Hence, the corresponding call to CONJOIN returns $[-7, 5]$.

Though the rules in Figure 3.8 capture the meaning of integer operations well, bit-vector rules for bit-level operations such as AND are difficult to formulate and representing their results as integer intervals can loose a considerable amount of precision. Assume, e.g. we want to calculate $\Phi_{e_1 \wedge e_2}$ given $\Phi_{e_1} = [-1, 0]$ and $\Phi_{e_2} = [-8, -7]$ where all expressions have a width of 4 bits. Using the two's complement representation of integers, this corresponds to the following calculation

$$\begin{array}{rr} & [-1,\ \ 0] \\ \wedge & [-8, -7] \\ \hline & [-8,\ \ 1] \end{array} \quad \leftrightarrow \quad \begin{array}{rr} & \text{- - - -} \\ \wedge & 1\ 0\ 0\ \text{-} \\ \hline & \text{- 0 0 -} \end{array}$$

Even the optimal result representable as integer interval $[-8, 1]$ contains more points than the exact result on the right which does e.g. not include -2.

3.3.3 Hybrid Simulation

The strengths and weaknesses of each homogeneous simulation strategy suggests a hybrid approach. To this end, we define two additional simulation rules [conjoin] and [select] as in Figure 3.9 that use Algorithms 3.3 and 3.4. The rule [conjoin] serves for combining bits to a bit-vector and the rule [select] to extract individual bits from a bit-vector. Combining these rules with the rules from ternary simulation in Figure 3.7 and the rules of interval simulation for integer operations as in Figure 3.8 allows for a simulation where bit-level operations are simulated using ternary simulation and integer operations are simulated with intervals. At the interfaces between the two kinds of operations act the transformation rules [conjoin] and [select].

$$\frac{\Phi_{e[0]} = [l_0, u_0] \quad \Phi_{e[1]} = [l_1, u_1] \quad \ldots \quad \Phi_{e[n-1]} = [l_{n-1}, u_{n-1}]}{\Phi_e = \text{CONJOIN}(l_0, u_0, l_1, u_1, \ldots, l_{n-1}, u_{n-1})} \text{ [conjoin]}$$

$$\frac{\Phi_e = [l, u]}{\Phi_{e[i]} = \text{SELECT}(l, u, i)} \text{ [select]}$$

Figure 3.9: Conjoin and Select Simulation Rules

As such, the choice between interval simulation and ternary simulation is driven by the specific expressions. If an expression is composed of bit-level operations, the expression is simulated with ternary simulation. Otherwise, for operations that capture the idea of integer interpretation of bit-vectors, interval simulation is applied. The rules [conjoin] and [select] which are susceptible to loosing precision are only applied when necessary at the interface between operations of different kind.

3.3.4 Example: Simulation

We conclude the section with an example. Assume we attempted to simulate $\Phi_e(\tilde{a})$ with

$$\begin{aligned} e &:= (e_1 < 2) \wedge (0 \leq e_1) \\ e_1 &:= e_2 \wedge e_3 \\ e_2 &:= x_1 - x_2 + 2 \\ e_3 &:= y_1 \vee y_2 \end{aligned}$$

and

$$\tilde{a} \quad := \quad (1 \leq x_1 \leq 5) \wedge (0 \leq x_2 \leq 3) \wedge (y_1 \in \text{-}\,0\,0\,\text{-}) \wedge (y_2 \in 1\,0\,0\,\text{-})$$

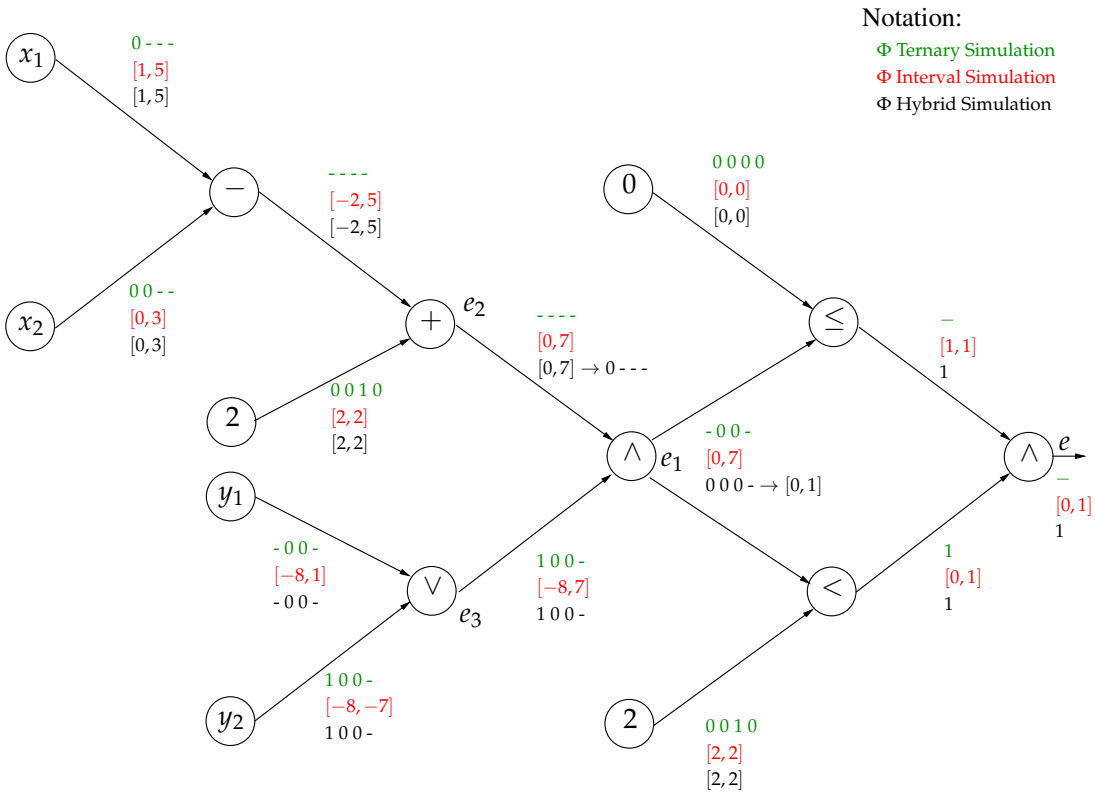


Figure 3.10: Example of a Hybrid Simulation

Figure 3.10 shows the expression DAG of e and the simulation results for all (sub)-expressions using ternary, interval, and hybrid simulation. On the one hand, ternary simulation works well for the simulation of subexpression e_3 sustaining the information that the second and third bit of the expression are low whereas interval simulation looses this information entirely. On the other hand, interval simulation performs well for the simulation of subexpression e_2 pertaining the information that the value must be positive while ternary simulation looses all information for this expression. However, only hybrid simulation is able to maintain both pieces of information and to combine them to $\Phi_{e_1}(\tilde{a}) = [0, 1]$. Hence, hybrid simulation yields the correct result that $\Phi_e(\tilde{a}) =$ true while both ternary and interval simulation fail due to their individual shortcomings. In case e corresponded to $B \wedge u$ in check (3.3), the expansion to $\tilde{a}$ would only succeed with hybrid simulation.

3.4 Implementation and Experimental Evaluation

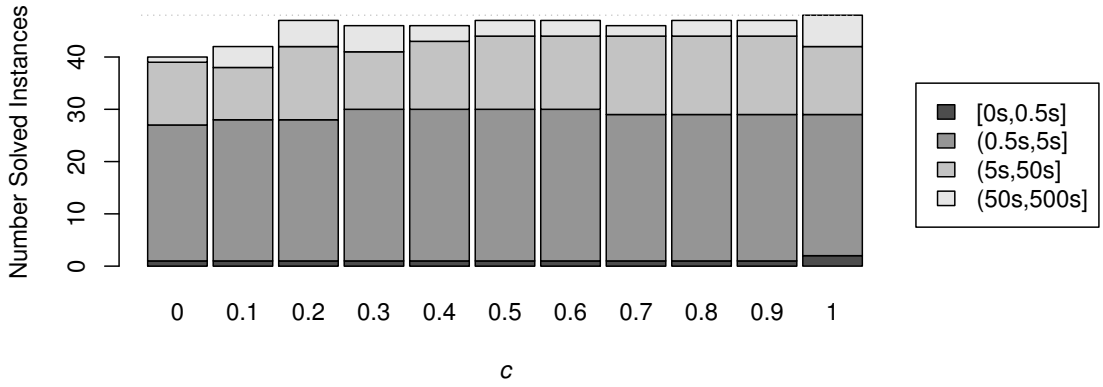
In this section, we describe our implementation of the presented QF_BV generalization of the PDR algorithm and results obtained from experimentation with the implemented software prototype. We start with a discussion of our implementation in the following subsection. Next, in Subsection 3.4.2, we present experimental results that demonstrate the impact of different choices for the ARU on the performance of the algorithm. In particular, we investigate the impact of parameter c on the probabilistic specialization of ARUs in the hybrid formulation with polytopes and Boolean cubes. Subsequently, in Subsection 3.4.3, we juxtapose the performance of simulation-based expansion of proof obligations with ternary, interval-based, and hybrid simulation. Finally, in Subsection 3.4.4, we review the overall performance of our generalized PDR algorithm by comparing it with that of the Boolean PDR algorithm provided in ABC [BM10] .

3.4.1 Implementation

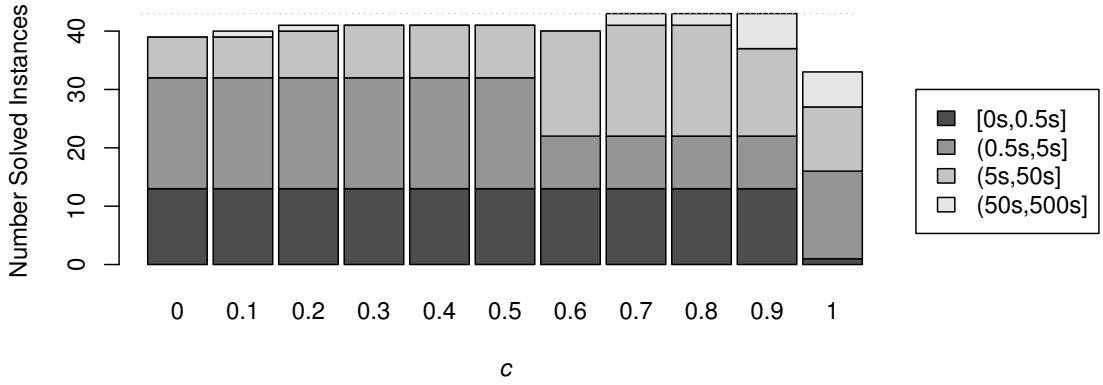
We implemented the proposed algorithm in C++. The skeleton of our implementation is similar to the one suggested in [EMB11]. As backend solver, we used the QF_BV theory part of the SMT solver Z3 [DMB08] instead of a SAT-solver. As Z3 supports assumption literals, analysis of the unsatisfiable core, and incremental solving, this choice allowed us to implement all of the suggested optimization described in [EMB11]. Most importantly, our implementation also inspects the unsatisfiable core to find opportunities for fast expansion of proof obligations and it also reuses learned clauses by the SMT solver for incremental solving using retractable assertions (hot solving). Unless stated otherwise in the following sections, the expansion of proof obligations is based on hybrid simulation as discussed in Section 3.3 and we use $c = 0.5$ as the probability that a point is specialized to a Boolean cube. To assure bit-accurate simulation results, we use the arbitrary precision integer class `APInt` provided as part of the LLVM-framework [LA04].

3.4.2 Impact of Specialization Probability Parameter c

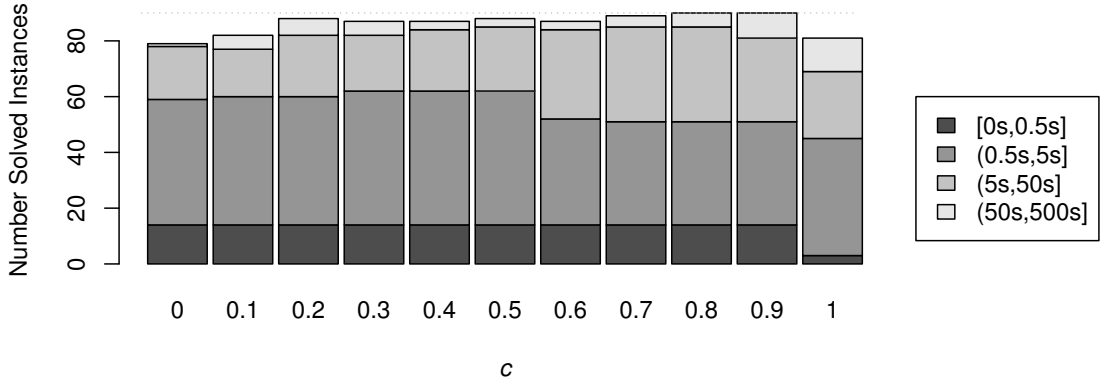
As discussed in Section 3.2, the choice of a specific kind of ARU is important for the success of the QF_BV PDR algorithm. In the proposed hybrid algorithm with polytopes and Boolean cubes, parameter c controls the probability that a point in the state space is spe-



(a) Instances extracted from the SV-COMP [Bey12] bit-vector benchmark set.



(b) Instances extracted from the INVGEN [GR09] benchmark set.



(c) All benchmark instances.

Figure 3.11: Impact of c on Number of Solved Benchmark Instances

cialized to a Boolean cube. The point is specialized to a polytope with probability $1 - c$.

Figure 3.11 documents the impact of parameter c on the efficacy of the overall algorithm on a set of model checking instances extracted from software verification benchmarks. The extreme cases $c = 1$ and $c = 0$ are the configurations of the algorithm with exclusive use of Boolean cubes and polytopes, respectively. The chart in Figure 3.11a shows the results for a subset of the entire set of instances derived from the bit-vector set of the SV-COMP [Bey12] benchmarks. On this subset, the algorithm with only Boolean cubes performs the best.

Figure 3.11b displays the same plot with instances derived from the benchmark set of INVGEN [GR09]. On this subset, the configuration with only Boolean cubes performs the worst. The varied result can be explained by the characteristics of the benchmark sets. On the one hand, the benchmarks derived from the SV-COMP benchmarks focus on bit-level characteristics. For instance, the inductive invariants required to solve these problems often specify a single bit having a specific value or two bits having the same value. On the other hand, in the INVGEN-benchmarks, bit-vectors are usually interpreted as integers and the relevant inductive invariants often represent linear relations between different variables using this interpretation. In this setting, the polytope interpretation is particularly suitable.

In Figure 3.11c, we show the impact of c shown for the entire set of model checking instances. Here, the algorithm performs similarly for all configurations but the two pure versions (only Boolean cubes or only polytopes) of the algorithm perform worse. This validates the theoretic discussion in 3.2.3 in so far as that the specific value for c does not matter much in practice unless c is chosen at the extreme values.

3.4.3 Impact of Simulation Type in Expansion of Proof Obligations

To investigate the impact of the simulation type on the performance of the algorithm, we conducted two experiments. The aim of the first experiment is to measure the impact of different simulation types on the efficacy of the expansion move itself. In the second experiment, we evaluate the impact of the simulation type on the performance of the overall model checking algorithm.

For the first experiment, we recorded the *volume* of ARUs before and after expansion moves where the volume is the number of points of the state space that are contained

in the ARU. We compared the measurements in form of the *expansion factor*, defined as the ratio of the volume of an ARU before and after the expansion move.

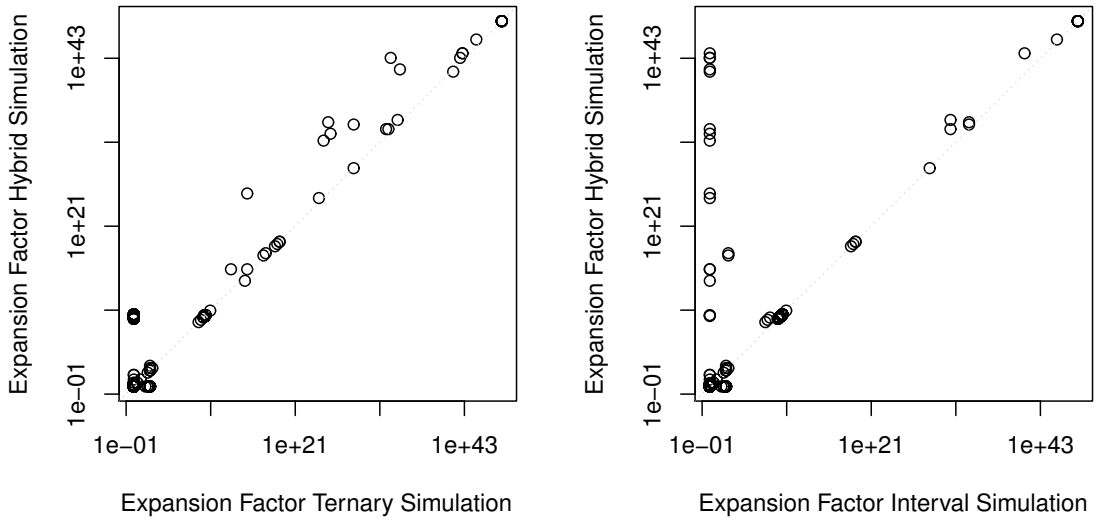
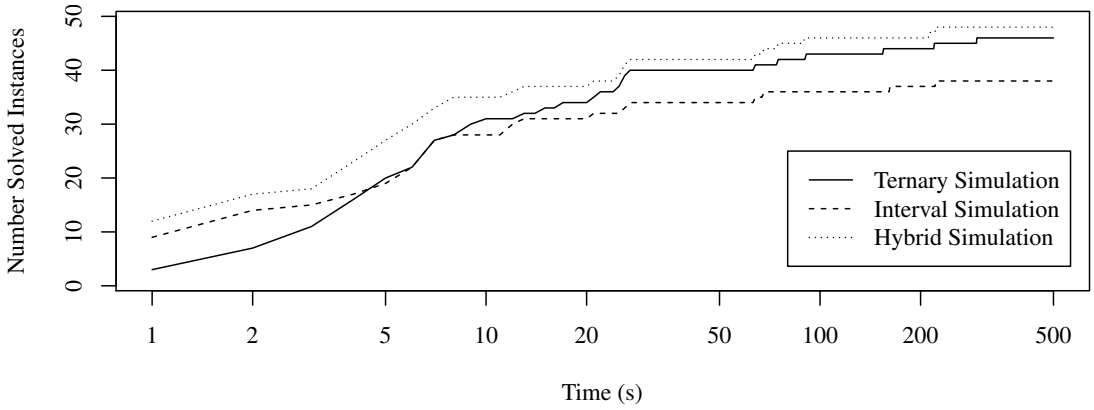


Figure 3.12: Expansion Factor of Hybrid Simulation vs Integer and Ternary Simulation.

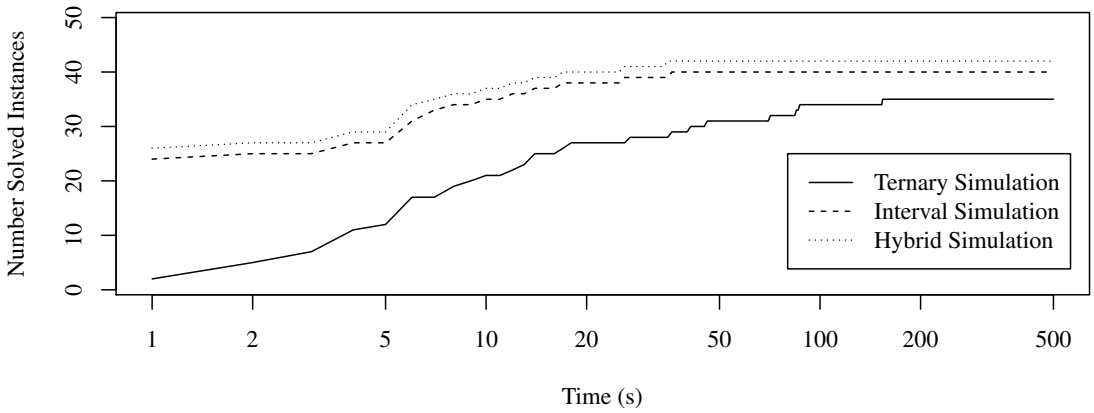
Figure 3.12 contains the results of this experiment. The comparisons between hybrid simulation on the one side and interval and ternary simulation on the other side side illustrates that hybrid simulation outperforms both pure simulation types, frequently by many orders of magnitude.

The results prove that hybrid simulation is an effective way of expanding ARUs. However, the results do not show how this impacts the overall performance of the algorithm. To investigate this question, in the second experiment, we run the set of benchmarks introduced in Subsection 3.4.2 three times each, once using ternary simulation, once using interval simulation, and once using hybrid simulation for the expansion of proof obligations. For the specialization probability, we choose $c = 0.5$ for this experiment. The results of the experiment are contained in Figure 3.13. Analog to Subsection 3.4.2, we present the results for the subset of the benchmarks derived from the SV-COMP competition repository (Figure 3.13a), for the subset of benchmarks derived from the InvGen-benchmark set (Figure 3.13b), and for the complete set of benchmarks (Figure 3.13c).

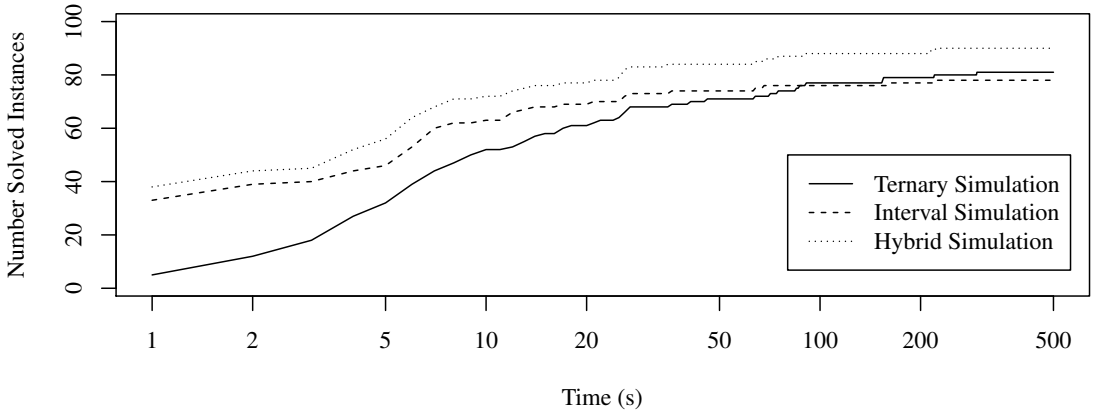
All three plots in Figure 3.13 suggest that the algorithm performs the best with hybrid simulation. Over the complete benchmark set, the algorithm with hybrid simulation is able to solve approximately 15% more instances than with either pure simulation type



(a) Instances extracted from the SV-COMP [Bey12] bit-vector benchmark set.



(b) Instances extracted from the INVGEN [GR09] benchmark set.



(c) All benchmark instances.

Figure 3.13: Impact of Simulation Type on Number of Solved Benchmark Instances

within the timeout limit of 500s. Interestingly, the plot in Figure 3.13a suggest that ternary simulation performs better than interval simulation whereas the plot in Figure 3.13b suggests the converse. As in the experimentation to investigate the impact of c on the performance of the model checker, the seemingly contradictory result can be attributed to the characteristics of the benchmarks. For model checking instances with focus on bit-level operations, ternary simulation is an appropriate means for the expansion of proof obligations but interval simulation often fails. On the other side, for the subset of benchmarks that focus on the integer interpretation of bit-vectors, interval simulation is appropriate but ternary simulation performs poorly. For the entire benchmark set, however, both pure approaches perform quite similarly.

3.4.4 Performance Comparison of QF_BV generalization vs ABC PDR

In the last experiment, we compare the performance of our generalized PDR algorithm with the original Boolean version contained in the logic synthesis and verification tool ABC [BM10]. To this end, we translated the benchmark model checking instances into BTOR [BBL08], a format for specifying word-level model checking problems, and used the tool SYNTHEBTOR, which is part of the distribution of the BOOLECTOR SMT solver [BB09], to translate the problems into AIGER [Bie06], a standard format for specifying hardware-model checking problems supported by ABC.

Figure 3.14 shows a comparison of running times to solve the given benchmark model checking instances with the presented QF_BV generalization of PDR versus the Boolean PDR model checker in ABC. For roughly 95% of the 155 benchmarks, the presented QF_BV generalization outperforms the Boolean PDR algorithm when run on the AIGER-versions of the benchmarks, often by more than an order of magnitude. This demonstrates that the QF_BV algorithm is able to use the additional information encoded in the word-level formulation of the model checking instances to speed up the solving. In the other roughly 5% of the benchmarks, the inductive invariant required to solve the model checking instance can simply be represented as a set of Boolean cubes. In this case, the generalized algorithm has no conceptional advantage over the Boolean algorithm and attempting to solve the model checking instance using polytopes is not purposeful, hence one cannot expect the generalized algorithm to outperform the original one.

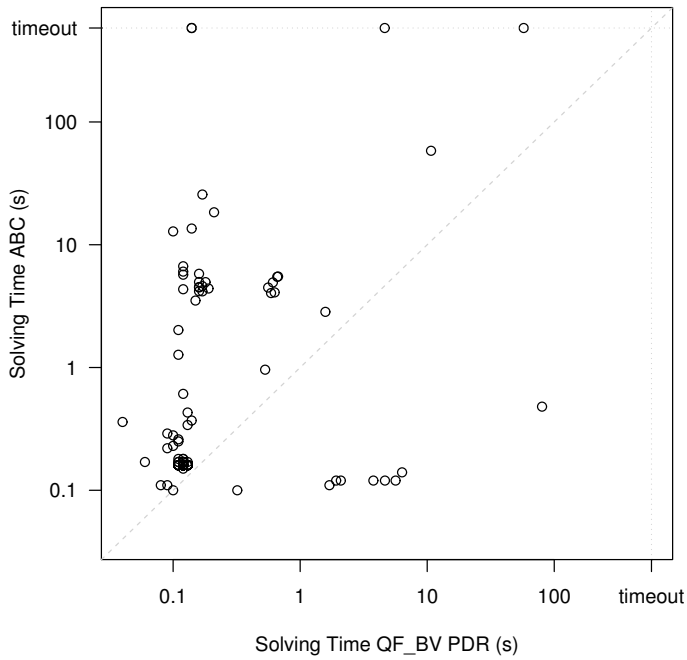


Figure 3.14: Comparison of Performances of QF_BV PDR vs ABC PDR.

Chapter 4

Program Verification with Property Directed Reachability

We start this chapter with a precise definition of the program verification problem we are considering. Subsequently, in Section 4.2, we deduce a framework to solve instances of the defined problem using PDR. Our ultimate solution, that is based on loop invariants, will be described in detail in Section 4.3. We conclude the chapter by discussing experimental results with our implementation of the presented algorithm in Section 4.4.

4.1 Definition Program Verification Problem

In the following, we assume that we are given a function-free PUV with static memory allocation that has assertions in the source code. We denote a program as safe if none of its assertions can be violated, regardless of the program input. Otherwise, we denote the program as not safe. Given an input program, the problem of interest, which we refer to as CHECK ASSERT, is to decide whether the program is safe or not. If the program is not safe, a counterexample is returned which provides input values with which the program can be interpreted such that the assertion is hit and the assertion condition does not hold.

Many practical applications can be reduced to CHECK ASSERT. Our main focus in this work is on proving safety properties of a program, which allow an immediate reduction to the given problem. However, one can also reduce the problem of test generation to CHECK ASSERT. For instance, if one was interested in covering a certain branch in the pro-

gram, one could add an assertion in front of the branching instruction asserting the branch condition. If the branch can be taken, the program would be determined as not safe and the returned counterexample encodes a suitable test. Otherwise, in case the branch cannot be taken, the program would be determined safe. In addition, problems arising in static analysis [ECCH00] and compiler optimization [ALSU06] can be reduced to CHECK ASSERT.

4.2 Towards a PDR-based Framework for Program Verification

In this section, we develop a scheme for program verification with the PDR algorithm presented in Chapter 3. We start with presenting a purely symbolic encoding of the PUV and progress towards a more effective strategy where loop iterations are mapped to transitions of model checking instances. Eventually, we arrive at a recursive scheme where loop invariants are refined using PDR.

4.2.1 Explicit Modeling the Program Counter

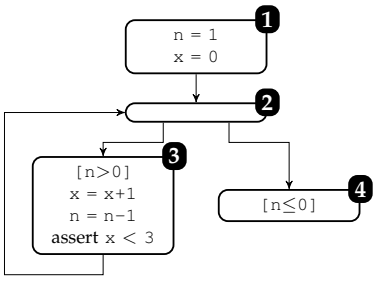
As discussed in the previous section, in this work we are concerned with recursion-free programs with static memory allocation. The most straight-forward solution to solving instances of CHECK ASSERT using the PDR algorithm is by formulating the proof obligations in the PUV as QF_BV model checking instances modeling the program pointer explicitly [ISGG05].

Consider, e.g. the program segment in Figure 4.1a and the corresponding CFG in Figure 4.1b. The numbers in the upper right corner of the basic blocks indicate their numbering that we will use in the following for reference. For instance, the entry basic block on the top of Figure 4.1b will be denoted with B_1.

If we use variable p to hold the index of the basic block the program is currently in, we can

```
n = 1;
x = 0;
while (n>0)
{
    x = x+1;
    n = n-1;
    assert(x < 3);
}
```

(a) Program Fragment



(b) Control Flow Graph

Figure 4.1: Example: Simple

transcribe the PUV into the following model checking instance

$$
\begin{aligned}
I &:= (p \equiv 1) \\
T &:= \left(x' \equiv \begin{cases} 0 & \text{if}(p \equiv 1) \\ x+1 & \text{if}(p \equiv 3) \\ x & \text{otherwise} \end{cases} \right) \wedge \\
& \left(n' \equiv \begin{cases} 1 & \text{if}(p \equiv 1) \\ n-1 & \text{if}(p \equiv 3) \\ n & \text{otherwise} \end{cases} \right) \wedge \\
& \left(p' \equiv \begin{cases} 2 & \text{if}(p \equiv 1) \vee (p \equiv 3) \\ 3 & \text{if}(p \equiv 2) \wedge (n > 0) \\ 4 & \text{otherwise} \end{cases} \right) \\
B &:= (p \equiv 3) \wedge (x \geq 2)
\end{aligned}
$$

Initially, the program is in the entry basic block, hence $p = 1$. In each iteration, the statements within basic block B_p are executed and the control flow transitions to the beginning of the next basic block. The execution of the statements is reflected in the next state functions of variables x and n. For instance, if the control flow is in the entry basic block B_1, n and x get their new respective values 1 and 0. The transitions are reflected in the next state functions for p. For instance, if the control flow is currently in basic blocks B_1 or B_3, the next basic block will be B_2.

The presented encoding is sound and complete and the derived model checking instances can be solved with the generalized PDR algorithm as presented in Chapter 3.

However, empirically, the encoding typically yields long solving times and is only practically for the smallest PUVs as we will show in Section 4.4.

4.2.2 Mapping Transitions to Loop Iterations

A more compact encoding can be achieved if loop iterations are mapped to transitions of the corresponding model checking instance. Using this paradigm, one obtains for the program in Figure 4.1 the following model checking instance that we already encountered in Chapter 3.

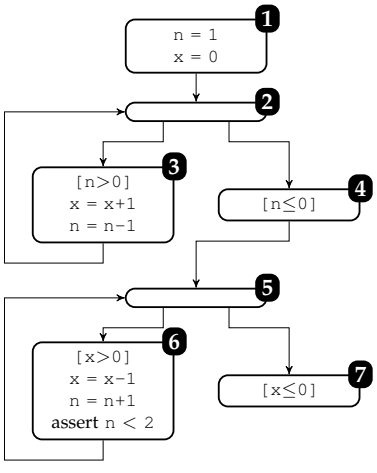
$$\begin{aligned} I &:= (n \equiv 1) \wedge (x \equiv 0) \\ T &:= (n > 0) \wedge (x' \equiv x + 1) \wedge (n' \equiv n - 1) \\ B &:= (x \geq 3) \end{aligned}$$

Here, the initial condition I reflects the portion of the program before the loop, the transition relation T the loop condition $n > 0$ and the increment and decrement of n and x, respectively, and the bad set B is the inverse of the assertion.

This encoding style is more compact than modeling the program counter explicitly. However, the paradigm does not readily generalize to programs with more than one loop. As an example, consider the program in Figure 4.2 which consists of two loops in series.

```
n = 1;
x = 0;
while (n>0)
{
    x = x+1;
    n = n-1;
}
while (x>0)
{
    x = x-1;
    n = n+1;
    assert(n < 2);
}
```

(a) Program Fragment



(b) Control Flow Graph

Figure 4.2: Example: Series

4.2.3 Multiple PDR instances for Multiple Loops

One solution to cope with a series of loops as in the program in Figure 4.2 is to use several model checking instances that are managed by a master algorithm. A proper means of communication between the individual loops are counterexamples and invariants. For the example, we initially assume that x and n can take arbitrary values at basic block B_4. We compose the model checking instance

$$\begin{aligned} I &:= \text{true} \\ T &:= (x > 0) \wedge (x' \equiv x - 1) \wedge (n' \equiv n + 1) \\ B &:= (n < 2) \end{aligned}$$

for the second loop. This model checking instance does not hold and if the PDR algorithm is run on this instance, we obtain a counterexample sequence. Assume the returned counterexample sequence is

State 0	→	State 1	→	State 2
$x = 4$		$x = 3$		$x = 2$
$n = 0$		$n = 1$		$n = 2$

The initial values $x = 4, n = 0$ can be used to compose a model checking instance for the first loop

$$\begin{aligned} I &:= (n \equiv 1) \wedge (x \equiv 0) \\ T &:= (n > 0) \wedge (x' \equiv x + 1) \wedge (n' \equiv n - 1) \\ B &:= (x \equiv 4) \wedge (n \equiv 0) \end{aligned}$$

If this model checking instance was satisfiable, we would have proved that the program is not safe. However, as we have seen in Section 3.2.1 for a more general proof obligation, the model checking instance holds. Assume that PDR returns the same invariant as calculated in Section 3.2.1:

$$(x < 3) \wedge (n < 2) \wedge (x < 1 \vee n < 1)$$

This invariant can be used as new initial condition for the second loop, yielding

the following updated model checking instance for the second loop

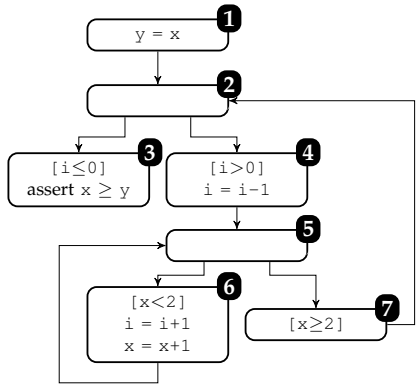
$$\begin{aligned} I &:= (x < 3) \wedge (n < 2) \wedge (x < 1 \vee n < 1) \\ T &:= (x > 0) \wedge (x' \equiv x - 1) \wedge (n' \equiv n + 1) \\ B &:= (n < 2) \end{aligned}$$

With the updated initial condition, the model checking instance holds. This proves that the PUV is safe.

The outlined algorithm has the advantage that all model checking instances only pertain single loops, i.e. their sizes are limited by the size of loops in the PUV. Unfortunately, as formulated, the algorithm cannot be applied for programs with nested loops such as the one in Figure 4.3.

```
y = x;
while (i>0)
{
     i = i-1;
     while (x<2)
     {
          x = x+1;
          i = i+1;
     }
}
assert(x >= y);
```

(a) Program Fragment



(b) Control Flow Graph

Figure 4.3: Example: Nested Loops

4.2.4 Loop Invariants

We reformulate the algorithm in order cope with nested loops. Instead of calculating invariants at points between loops, we now use the PDR algorithm to calculate loop invariants as functions of the loop variables before and after the loop. We denote the variables holding the values before a loop with a b suffix and the ones holding the values at the end of a loop with a e suffix. For the nested loop example in Figure 4.3, denote with $\mathrm{LI}_1(\cdot)$ the loop invariant of the outer loop and with $\mathrm{LI}_2(\cdot)$ the loop invariant of the inner loop. Both x and i are modified and referenced in either loop. To indicate that x can only increase

within the outer loop, we would e.g. use the loop invariant

$$\mathrm{LI}_1(x_e, i_e, x_b, i_b) := x_e \geq x_b \tag{4.1}$$

The use of loop invariants enables to model the inner loop when constructing the model checking instance of the outer loop. For the nested loops example we can compose the following model checking instance

$$\begin{aligned} I &:= (y \equiv x) \\ T &:= (y' \equiv y) \wedge (i > 0) \wedge \mathrm{LI}_2(x', i', x, i-1) \\ B &:= (i \leq 0) \wedge (x < y) \end{aligned} \tag{4.2}$$

where the inner loop is modeled by the loop invariant $\mathrm{LI}_2(\cdot)$.

It remains to discuss how the loop invariants are obtained. We propose to use PDR for this purpose. This allows for an incremental and recursive invariant refinement scheme. Assume that $\mathrm{LI}_2(\cdot)$ in the model checking instance (4.2) is *true*. In this case, the model checking instance does not hold and the PDR algorithm would return a counterexample sequence, for instance

State 0		State 1
$x = 4$	$\rightarrow$	$x = 3$
$i = 1$		$i = 0$
$y = 4$		$y = 4$

(4.3)

This counterexample sequence is spurious. The body of the nested loop does not allow the transition. We can construct the following model checking instance to refine the loop invariant for the nested loop

$$\begin{aligned} I &:= (x_b \equiv x_e) \wedge (i_b \equiv i_e) \\ T &:= (x_b' \equiv x_b) \wedge (i_b' \equiv i_b) \wedge (x_e < 2) \wedge (x_e' \equiv x_e + 1) \wedge (i_e' \equiv i_e + 1) \\ B &:= (x_b \equiv 4) \wedge (i_b \equiv 1 - 1) \wedge (x_e \equiv 3) \wedge (i_e \equiv 0 - 1) \end{aligned}$$

Initially, the values at the beginning of the loop and after the loop are equal. In each iteration of the loop, the values at the beginning of the loop stay constant but the values at the end of the loop change as defined in the body of the nested loop (see Figure 4.3). The spurious transition in (4.3) is reflected as bad set.

The model checking instance holds and PDR could infer e.g. the inductive invariant

$$\mathrm{LI}_2(x_e, i_e, x_b, i_b) := (x_b < 4) \vee (x_e \geq 3)$$

With this strengthened loop invariant, the spurious counterexample sequence would no longer be possible but the model checking instance in (4.2) continues not to hold. However, iterative application of the sketched refinement scheme would eventually infer a loop invariant strong enough to resolve CHECK ASSERT in the nested loop example.

The outlined iterative and recursive refinement scheme allows for a sound and complete algorithm to decide CHECK ASSERT on all programs we are considering. In the next section, we will present the proposed algorithm in detail and discuss its properties.

4.3 Program Verification with PDR

Figure 4.4 depicts a high-level view of the proposed algorithm to solve instances of CHECK ASSERT. In a preprocessing stage, the PUV is transformed into a model that overapproximates the behavior of the program. Next, we check whether the program is safe using this model. If so, we report that the program is safe. Otherwise, we obtain a counterexample that can either be real or spurious where we denote a counterexample as *spurious* if it is admitted by the current model of the program but not by the real program. In the former case, we report that the program is not safe. Otherwise, we use the counterexample to refine the model of the program.

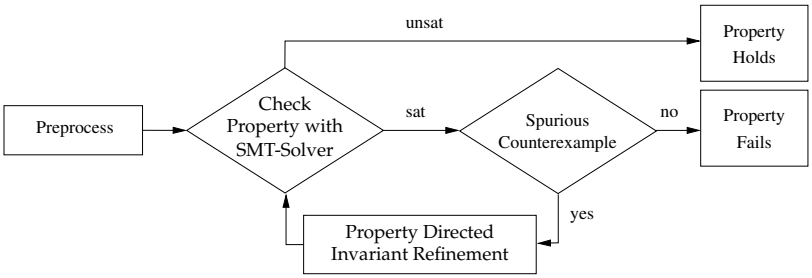


Figure 4.4: High-Level Overview of the Proposed Framework

In the following subsections, we discuss details for the individual parts of the framework. We begin with discussing the preprocessing step in the next subsection, followed by a discussion of the main loop in Subsection 4.3.2, and finally present the loop

refinement using PDR in Subsection 4.3.3. We illustrate each part using the nested loop example from Figure 4.3.

4.3.1 Preprocessing

Consider the CFG of the input program. For simplicity of exposition, we assume the program to be structured [BJ66, Dij68]. Then, every loop l in the CFG can be uniquely identified by a pair (H, T) of basic blocks, where H is the loop head and T is the loop tail. A basic block T is called a loop tail if it is dominated by the loop head H and there exists an edge (T, H) in the CFG. We call the edge (T, H) a back edge. A basic block B_1 dominates basic block B_2 if every path from the entry basic block to B_2 passes through block B_1. For more detailed information on this notation, please refer to [ALSU06].

The loop body of a loop $l = (H, T)$ (denoted as body_l) is the set composed of all basic blocks that are on any path between H and T. We associate with l two sets of variables U_l and R_l where U_l is the set of variables which are updated in loop_l and R_l is the set of variables which are referenced in loop_l. Note that $U_l \subseteq R_l$.

Algorithm 4.1 `preprocess()`

1: **for** each loop $l = (H, T)$ in PUV **do**
2: remove back edge (T, H)
3: append $\text{LI}_l(U_l, R_l)$ to H
4: set $\text{LI}_l(\cdot) = \mathit{true}$
5: **end for**
6: `passify()`

The individual steps of the preprocessing are summarized in Algorithm 4.1. We iterate through all loops and for each loop $l = (H, T)$, we cut the associated back edge (T, H). Note that after this step, the CFG becomes a directed acyclic graph (DAG) that underapproximates the behavior of the PUV. Next, we append a loop invariant $\text{LI}_l(U_l, R_l)$, a predicate over the two sets of variables U_l and R_l, to the loop header H. The first(second) argument of $\text{LI}_l(\cdot)$ corresponds to the values of the variables after(before) the loop. Semantically, $\text{LI}_l(\cdot)$ can be understood as a multivariate assignment to all variables in U_l. The new values of the variables are arbitrary under the condition that the predicate $\text{LI}_l(\cdot)$ evaluates to *true* when applied to the combination of variable assignments before and after

the loop. Initially, we set all loop invariants to *true*. Intuitively, this means that after the loop invariant, all loop variables in U_l may take arbitrary values. Notice that after this preprocessing, the obtained model overapproximates the behavior of the PUV.

In the main loop of our verification algorithm, we will check whether or not the program is safe using an SMT solver. To this end, we construct an SMT formula F that is unsatisfiable only if the program is safe. We encode the program using three sets of constraints.

The first set of constraints assures that a possible solution to the SMT formula corresponds to a feasible flow through the program. Therefore, we associate with each basic block B_i in the CFG a Boolean variable b_i and encode the control flow as follows. For each basic block B_i with the exception of the entry basic block we add a clause

$$b_i \Rightarrow \bigvee_{B_j \in \text{Predecessors}(B_i)} b_j$$

that assures that B_i can only be visited if at least one of the predecessors is visited, too.

The second set of constraints is to model the data-flow of the program. Therefore, we translate the PUV into a passive program using dynamic single-assignment [Fea91]. The principal idea of this technique is that each time a variable is updated, the new value is assigned to a new copy of the variable. In accordance to the literature, we will refer to these copies as variants. In our notation, we indicate the i^{th} variant of variable x with x_i. As an example, a statement such as $x = x + 1$ in the PUV would be represented as $x_{i+1} \equiv x_i + 1$ in the passive program. As a loop invariant $\text{LI}_l(U_l, R_l)$ corresponds to a multivariate assignment to all the variables in the first argument, we increment the variant counter for each variable in U_l. The predicate itself is represented by substituting each variable corresponding to the value after(before) the loop with the updated(previous) variant. For instance, loop invariant (4.1) would be represented as $x_{i+1} \geq x_i$ in the passive program. For each passive formula f encoding a statement, assumption, or loop invariant of the resulting passive program, we add a clause of the form

$$b_i \Rightarrow f$$

to F if B_i is the basic block in which f resides. Intuitively, the clause encodes that if the control flow visits B_i, then f must be true.

Lastly, in the third set of constraints, we constrain that the model of the program is not safe. For an assertion A in basic block $B_{\pi(A)}$ with passive formula a as condition to

be violated, we must visit block $B_{\pi(A)}$ and a must not hold. The model is not safe if at least one assertion is violated. Hence, we have formally

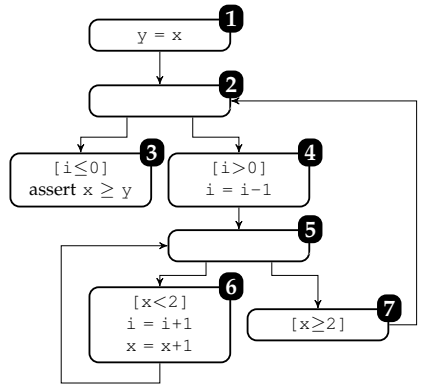
$$\bigvee_{A \in \text{Assertions}} b_{\pi(A)} \wedge \neg a$$

Example: Nested Loops

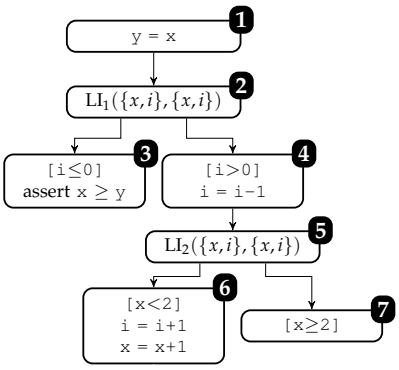
Reconsider the program fragment and corresponding control flow graph of the nested loops example from Section 4.2.3 (reprinted in Figures 4.5a-b for convenience) for another time.

```
y = x;
while (i>0)
{
     i = i-1;
     while (x<2)
     {
          x = x+1;
          i = i+1;
     }
}
assert(x >= y);
```

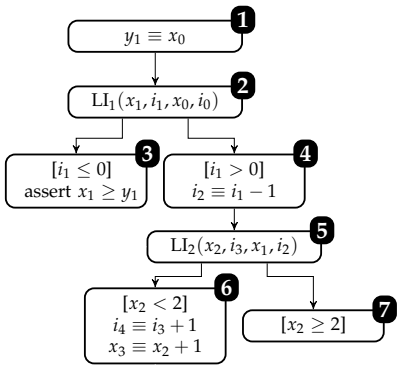
(a) Program Fragment



(b) Control Flow Graph



(c) Acyclic Program Model



(d) Passive Program

Figure 4.5: Preprocessing for Nested Loops Example

The program fragment contains two loops $l_1 = (B_2, B_7)$ and $l_2 = (B_5, B_6)$. We remove the two corresponding back edges (B_7, B_2) and (B_6, B_5) and append $\text{LI}_1(\cdot)$ and

$\mathrm{LI}_2(\cdot)$ to the loop heads B_2 and B_5, respectively. These steps yields the DAG in Figure 4.5c. Next, the DAG is passified and one obtains the graph in Figure 4.5d.

To check whether the so obtained model of the PUV is safe, we construct an SMT formula F as follows. We assure that every satisfying assignment corresponds to a valid control flow by adding the conjunction of the following control-flow constraints to F where, for instance, the first constraint assures that basic block B_2 cannot be visited unless B_1 is also visited.

$$(b_2 \Rightarrow b_1) \wedge (b_4 \Rightarrow b_2) \wedge (b_6 \Rightarrow b_5) \wedge (b_3 \Rightarrow b_2) \wedge (b_5 \Rightarrow b_4) \wedge (b_7 \Rightarrow b_5)$$

Next, we add the following constraints to the model to assure proper data-flow.

$$\begin{array}{lllllllllllll}
(b_1 & \Rightarrow & (y_1 \equiv x_0)) & \wedge & (b_2 & \Rightarrow & \mathrm{LI}_1(x_1, i_1, x_0, i_0)) & \wedge & (b_3 & \Rightarrow & (i_1 \leq 0)) \\
(b_4 & \Rightarrow & (i_1 > 0)) & \wedge & (b_4 & \Rightarrow & (i_2 \equiv i_1 - 1)) & \wedge & (b_5 & \Rightarrow & \mathrm{LI}_2(x_2, i_3, x_1, i_2)) \\
(b_6 & \Rightarrow & (x_2 < 2)) & \wedge & (b_6 & \Rightarrow & (i_4 \equiv i_3 + 1)) & \wedge & (b_6 & \Rightarrow & (x_3 \equiv x_2 + 1)) \\
(b_7 & \Rightarrow & (x_2 \geq 2)) & & & & & & & & &
\end{array}$$

As an example, the third constraint in the first line requires that if basic block B_3 is visited, then condition $i_1 \leq 0$ must hold.

Lastly, we constrain that the assertion in B_3 must be violated. We enforce this by asserting that

$$b_3 \wedge (x_1 < y_1)$$

If the so composed SMT formula F is unsatisfiable, then the program is safe.

4.3.2 Iterative Refinement

Algorithm 4.2 summarizes the main loop of our proposed algorithm for program verification. After having preprocessed the program and having constructed the SMT formula F as discussed in the previous subsection, we check if F is satisfiable. If this is not the case, then we return that the program is safe. Otherwise, we obtain a satisfying assignment to all variables in F, counterexample c, and check whether c corresponds to a run that violates an assertion. To this end, we interpret the PUV with the arguments encoded in c starting at the entry basic block B_e. If the interpretation run violates an assertion, we report the program as not safe. Otherwise, the observed discrepancy between the real program and its model encoded as SMT formula is due to the weakness of one or more

loop invariants, i.e. there exists at least one loop whose loop invariant admits c but its loop body does not. We refer to such a loop as being *weakly modeled* with respect to the counterexample. We find such a weakly modeled loop l with respect to c using the subroutine `findWeaklyModeledLoop()` and strengthen its invariant using `refine()` such that the invariant continues to overapproximate the behavior of the loop but no longer admits the spurious counterexample c. Then we repeat with checking for a counterexample in the refined model of the PUV.

Algorithm 4.2 `verifyProgram()`

```
1: preprocess()
2: while counterexample c exists do
3:     if interpretationHitsAssertion(c, B_e) then
4:         return "Program not safe"
5:     else
6:         l = findWeaklyModeledLoop(c)
7:         refine(l, c)
8:     end if
9: end while
10: return "Program safe"
```

Theorem 1. *Algorithm 4.2 is a sound and complete algorithm to solve instances of* CHECK ASSERT.

Proof. We sketch the proof of Theorem 1. We start by showing partial correctness. First note that if `verifyProgram()` returns "Program not safe", then it has found a real counterexample. To see why the program is safe if the algorithm returns "Program safe", note that at any time, the loop invariants are overapproximations of the behavior of the loop because the loop invariants are initially assigned true and any refinement step preserves the property that the pertaining loop invariant continues to overapproximate the behavior of the actual loop. Hence, at any time F models an overapproximation of the complete program. If the program is safe using this overapproximation, it is also safe on the concrete program. It remains to show that the algorithm terminates. As we assume static memory allocation, each counterexample is over a fixed number of variables and each variable can only hold a finite number of values. Hence, there is only a finite number of different

counterexamples. In each call of `refine()`, the model of the PUV is refined such that all previous and the current counterexample are no longer admitted. Hence, in each iteration of the main loop, at least one additional spurious counterexample is excluded and the algorithm terminates in a finite number of iterations. Also, each called subroutine terminates in finite time. This is obvious for routine `preprocess()` as discussed in the previous subroutine. We will argue why the other routines terminate in finite time in the following subsections. This implies the result. □

Counterexample Analysis with Interpretation

We use interpretation to validate the correspondence between a counterexample and the original sequential program in two contexts: In the first context (applied in line 3 in Algorithm 4.2), we want to validate that a counterexample corresponds to a run of the PUV hitting an assertion. To this end, we extract the values for all variables at the entry basic block B_e from the counterexample c, initialize an interpreter with these values and start the interpretation run. If the run hits an assertion, `interpretationHitsAssertion()` returns *true*. Note that this check does not guarantee that the actual run of the program corresponds to the one predicted by the counterexample c but only that an assertion will be hit with the initial values encoded in c. In the context of algorithm 4.2, however, this inaccuracy is not only irrelevant but beneficial because it allows for the accidental finding of a real counterexample using an inconsistent counterexample.

Programs with infinite loops represent another challenge for the algorithm. If the interpreter enters an infinite loop, it may continue interpreting the same code repeatedly with the same valuation of the variables. To guarantee termination of the interpretation, our implementation of the interpreter records already visited program states and bails out of the interpretation run reporting that no assertion is hit if the same program state is visited a second time.

In the second context (applied in line 6 in Algorithm 4.2), we have already determined that the current counterexample c is spurious and we are interested in finding a loop which is weakly modeled with respect to c. Algorithm 4.3 gives details of how this is done: The b_i variables with true value in c encode a path Π in the DAG of the passive program. Denote with $B_{\pi(i)}$ the i^{th} of n loop headers in Π. Given this notation, we find a weakly modeled loop l by iterating from n to 1 and test for each i whether an interpre-

tation run starting from the i^{th} loop header hits an assertion. Note that Algorithm 4.3 is

Algorithm 4.3 `findWeaklyModeledLoop(c)`

```
1: Π = path through passive program
2: n = number of loop headers in Π
3: for i = n to 1 do
4:     if not interpretationHitsAssertion(c, B_π(i)) then
5:         return π(i)
6:     end if
7: end for
8: assert(false)
```

guaranteed to find a weakly modeled loop if c is spurious, i.e. that the assertion in line 6 will never be violated. To see why, note that the for-loop in the algorithm has the invariant that in the i^{th} iteration of the for-loop, interpreting the PUV from the corresponding loop-header of the PUV with the post-values encoded in the counterexample hits an assertion. If the interpretation run started with the pre-values encoded in c does not hit an assertion, we have found a weakly modeled loop. Otherwise, we know that the post-values of the $(i-1)^{\text{th}}$ loop on the path must hit an assertion as all program constructs but loops are modeled precisely in the passive program.

Example: Nested Loops (cont'd)

With the loop invariants $\text{LI}_1(\cdot)$ and $\text{LI}_2(\cdot)$ being initially *true*, the program cannot be proved to be safe. Assume that the SMT solver returned the counterexample

$$c = \{y_1 = 0, x_0 = 0, i_0 = 0, x_1 = -1, i_1 = 0, \dots\} \tag{4.4}$$

An interpretation run from the beginning of the program with the initial values `x=0, i=0` quickly shows that the counterexample c is spurious. The path modeled by the counterexample is $B_1 \to B_2 \to B_3$. Starting an interpretation run from the latest loop header on this path (B_2) with the pre-values `x=0, i=0, y=0` does not violate any assertion. Hence we can conclude that $\text{LI}_1(\cdot)$ is too weak. Next, we call the procedure `refine()` on loop 1 which strengthens its loop invariant to

$$\text{LI}_1(\cdot) = (x_0 < 0) \vee (x_1 > -1)$$

In the next iteration of the while-loop, we can find another spurious counterexample such as

$$c = \{y_1 = 1, x_0 = 1, i_0 = 0, x_1 = 0, i_1 = 0, \dots\}$$

Again, we refine the invariant for loop 1. This time, we obtain

$$\mathrm{LI}_1(\cdot) = (x_1 \geq x_0)$$

as new, strengthened loop invariant. With this invariant for loop 1, the SMT formula F is no longer satisfiable. This proves that the program is safe.

It is instructive to note that the derived loop invariant is not the strongest loop invariant that can be inferred for loop 1 but that it is strong enough to prove the desired property. In this sense, our algorithm is also property directed and therefore appears to be well suited to be used in combination with PDR. At the same time as being property directed, however, our algorithm also assures abstraction from the counterexamples. For instance, the inferred loop invariant is independent from variable i.

4.3.3 Property Directed Invariant Refinement

Algorithm 4.4 gives an overview of how $\mathrm{LI}_l(\cdot)$ is refined given a spurious counterexample c. After generalizing c, we invoke the QF_BV PDR algorithm with the intent to infer a strengthened loop invariant for l that contradicts c. If the attempt is successful, we update the SMT formula F to reflect the new loop invariant returned by the backend model checker. Note that this can be implemented by simply adding $b_H \Rightarrow \mathrm{LI}_l(\cdot)$ to the constraint

Algorithm 4.4 `refine(`l`, `c`)`

1: `generalizeCounterexample(`l`, `c`)`
2: **while** `PDR(`I, T, c`)` does not hold **do**
3: let s be the counterexample sequence
4: let c' be a spurious transition in s
5: let l' be a loop whose $\mathrm{LI}_{l'}$ admits c' but body_l does not
6: `refine(`l', c`)`
7: **end while**
8: update F to reflect strengthened $\mathrm{LI}_l(\cdot)$

base because any new loop invariant implies all previous loop invariants which allows for effective use of incremental solving.

In the presence of nested loops with weak loop invariants, it is also possible that PDR returns a spurious counterexample sequence s. In this case, it remains to determine an individual spurious transition c' in s and to find a nested loop l' whose invariant admits c' while $\text{body}_{l'}$ does not. These tasks can be performed using interpretation similar as in Algorithm 4.3. Once a culprit loop l' is determined, `refine()` is called recursively to strengthen $\text{LI}_{l'}(\cdot)$. This recursive strengthening is repeated until all loop invariants of nested loops are strong enough such that PDR succeeds in disproving c.

In the following subsection, we discuss details of the counterexample generalization. Next, we describe the composition of the model checking instances which are fed to PDR for the actual invariant refinement. Both subsections are followed by illustrations which demonstrate how the discussed algorithms perform on our running example.

Generalization of Counterexamples

The motivation of the generalization of spurious counterexamples at the beginning of the `refine()` procedure is to promote that the QF_BV PDR model checker finds loop invariants that do not only exclude the current counterexample but also exclude a large number of related spurious counterexamples. As such, it is a pure optimization that is not required for the correctness of the algorithm. To some extend, counterexample generalization mirrors simulation-based expansion of proof obligations in the PDR algorithm originally presented in [EMB11] and generalized to a hybrid simulation approach in the QF_BV PDR solver presented in Chapter 3.

We propose to generalize counterexamples using abstract interpretation [CC77] with the abstract domain of integer intervals and vectors of ternary values. The choice for a specific abstract domain is furnished by the kind of operation that calculates an abstract value. We use integer intervals if a result is calculated by an operation that interprets bitvectors as integers (e.g. addition). For all other operations (e.g. conjunctions), we use vectors of ternary values.

We start the abstract interpretation runs at a loop header l and continue with the interpretation of the loop until a fixed point is found. The final value sets at loop header l represent overapproximations of the reachable values of the variables for the given initial

values.

Algorithm 4.5 `generalizeCounterexample`$(l,\ c = \{c_b, c_e\})$

1: r = `abstractInterpret`(l, c_b)
2: **for each** $x \in U_l$ **do**
3: **if** $r[x]$ disjoint of $c_e[x]$ **then**
4: expand $c_e[x]$ maximally preserving disjointedness from $r[x]$.
5: **for all** $y \neq x \in U_l$ **do** $c_e[y] = \top$
6: **break**
7: **end if**
8: **end for**
9: **for each** $x \in R_l$ **do**
10: Substitute $c_b[x]$ with largest value set $v \supseteq c_b[x]$ s.t.
11: $\exists y \in U_l$.`abstractInterpret`$(l, c_b/v)[y]$ is disjoint of $c_e[y]$
12: **end for**

Algorithm 4.5 gives details of how we use the results of abstract interpretation runs for the actual generalization of counterexamples. Note that we use c_b (c_e) to denote the portion of c that corresponds to the values of the variables before (at the end) of the loop and $\top$ to denote the maximum value set representable by a certain abstract domain. Lines 1-8 of Algorithm 4.5 aim at generalizing the value sets of the post-values of the counterexample c_e: To this end, the program is interpreted with the initial values c_b given in the counterexample to obtain an overapproximation r of the reachable values after the loop. If, for any variable $x \in U_l$, the set of reachable values $r[x]$ is disjoint from the corresponding post-values $c_e[x]$ in the counterexample, $c_e[x]$ is expanded maximally while preserving the disjointedness from the reachable value set. If $c_e[x]$ is represented as an integer interval, this is achieved by pushing the upper and lower bounds until $c_e[x]$ ceases to be disjoint of $r[x]$. In case $c_e[x]$ is represented as a vector of ternary values, we substitute 0 and 1 values with don't care values until $c_e[x]$ ceases to be disjoint of $r[x]$. This procedure guarantees that the resulting counterexample remains spurious regardless of any other variables. Hence, we can enlarge all other value sets corresponding to other post-variables maximally.

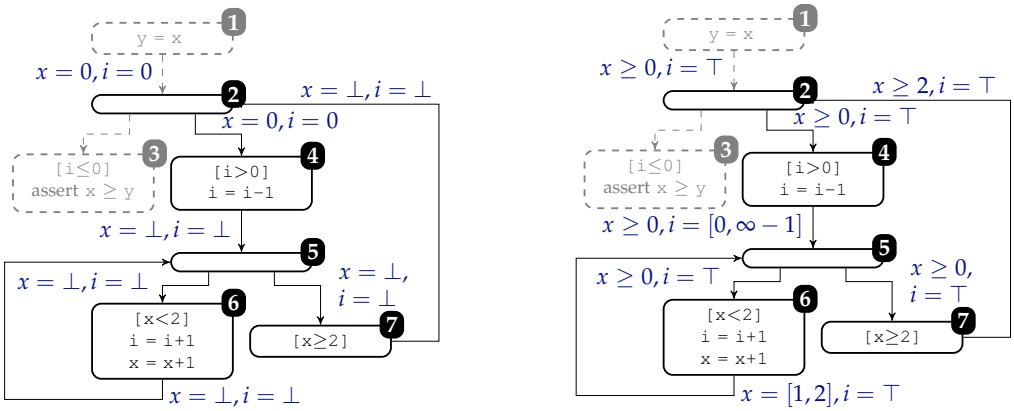
Lines 9-12 aim at generalizing the value sets for the initial values c_b in the counterexample. To this end, for each variable $x \in R_l$, we attempt to expand $c_b[x]$ maximally

such that for at least one variable y, the overapproximation of its reachable value set $r[y]$ remains disjoint from the corresponding post-values $c_e[y]$ in the counterexample. Again, the expansion procedure for finding a large value set depends on the specific abstract domain: If the value set is an integer interval, we attempt to increase it using binary search. Otherwise, in case the value set is a vector of ternary values, we attempt to replace 0 or 1 values with don't care values for each bit.

Example: Nested Loops (cont'd)

We consider the situation when `verifyProgram()` in Algorithm 4.2 calls `refine()` for the first time. Here, `generalizeCounterexample`() is called on loop 1 with the counterexample in equation (4.4). Mapping variants to variables, we have

$$c = \{c_b = \{x = 0, i = 0\}, c_e = \{x = -1, i = 0\}\}$$



(a) Fix-Point Abstract Interpretation 1

(b) Fix-Point Abstract Interpretation 2

Figure 4.6: Fix-Points of the Abstract Interpretations during Counterexample Generalization for the Nested Loops Example

An abstract interpretation run with the initial values in c_b yields to the fixed point in Figure 4.6a. With the initial values c_b, the loop is not entered and the reachable intervals at the loop header remain $r = \{x = 0, i = 0\}$. As $r[x]$ is disjoint from $c_e[x]$, we can expand c_e as defined in lines 4-5 of Algorithm 4.5 to obtain

$$c = \{c_b = \{x = 0, i = 0\}, c_e = \{x \leq -1, i = \top\}\}$$

Next, we expand the value sets in the c_b-portion of the counterexample. For $c_b[x]$, decreasing the lower bound causes that the reachable value sets r returned by the abstract interpretation ceases to be disjoint from the post-values in c_e. However, increasing the upper bound preserves this condition. For $c_b[i]$, we can increase the interval in both directions arbitrarily without causing that c_e ceases to be disjoint from the reachable intervals r (see fixed point of the corresponding abstract interpretation in Figure 4.6b. As in Chapter 3, we denote with ∞ the maximum value a bitvector can take). We obtain the generalized counterexample

$$c = \{c_b = \{x \geq 0, i = \top\}, c_e = \{x \leq -1, i = \top\}\} \tag{4.5}$$

Invariant Refinement

Given a loop $l = (H, T)$ and a spurious counterexample c, the input model checking instance (I, T, B) of the PDR-algorithm for loop l is constructed as follows. For each variable $x \in U_l$, denote with x_b and x_e, respectively, the corresponding variants at the beginning and at the end of loop l. For each variable $x \in R_l \setminus U_l$, denote with x_b the corresponding variant. To specify the set of initial states I, we require that for each variable x in U_l, we have that $x_e \equiv x_b$. Intuitively, this means that the values of a variable before the loop and after the loop are equal before any iterations. The transition relation T is composed of four parts. First, for each variable $x \in R_l$, we constrain that the initial value does not change: $x'_b \equiv x_b$. Second, for each variable $x \in U_l$, we constrain $x'_e \equiv x_T$ where x_T is the last instantiated variant of x in the loop tail T. Third, we add the SMT formula corresponding to the part of the CFG that represents the loop$_l$. Nested loops are represented by their loop invariants but their bodies are excluded. Fourth, we constrain b_T, i.e. that each transition requires that the loop tail is reached. The bad states B are the states that are contained in the generalized counterexample c.

This construction guarantees that the size of the model checking instances are bound by the size of the loop. Unrolling or "inlining" of nested loops are purposely avoided. In our experience, this usually entails that SMT formulae are solved quickly.

We preserve the trace constructed for loop l by the PDR algorithm between calls. This avoids that runtime required to prove that a certain subspace is unreachable in a certain number of steps is spent more than once. We argue why this memoization does not compromise the correctness of the results returned by PDR. Consider any two successive

invocations $\texttt{PDR}(I_1, T_1, B_1)$ and $\texttt{PDR}(I_2, T_2, B_2)$ for refinement of the same loop. The trace constructed during the first call may be reused in the second call, because

1. we always call PDR with the same set of initial states, i.e. $I_1 = I_2$,
2. and any refinement of a nested loop invariant leads to a strengthening of the transition relation. As a consequence, T_2 is stricter than T_1 (formally $T_2 \Rightarrow T_1$) and any state determined unreachable within a certain number of states using the weaker transition relation T_1 is also unreachable under the stronger transition relation T_2.

Example: Nested Loops (cont'd)

We use the QF_BV PDR-solver to strengthen the loop invariant of loop 1 with the generalized counterexample in equation (4.5). Constructing the model checking instance using the directions above, we obtain the following model checking instance as input of the PDR algorithm.

$$\begin{aligned} I &= (x_1 \equiv x_0) \wedge (i_1 \equiv i_0) \\ T &= (x_0' \equiv x_0) \wedge (i_0' \equiv i_0) \wedge (x_1' \equiv x_2) \wedge (i_1' \equiv i_3) \wedge (b_5 \Rightarrow b_4) \\ &\quad \wedge (b_7 \Rightarrow b_5) \wedge (b_4 \Rightarrow (i_1 > 0)) \wedge (b_4 \Rightarrow (i_2 \equiv i_1 - 1)) \\ &\quad \wedge (b_5 \Rightarrow \mathrm{LI}_2(\cdot)) \wedge (b_7 \Rightarrow (x_2 \geq 2)) \wedge b_7 \\ B &= (x_0 \geq 0) \wedge (x_1 \leq -1) \end{aligned}$$

Note that the current version of $\mathrm{LI}_2(\cdot)$ is used within the transition relation. As this loop invariant is initially *true*, this practically allows any update in one iteration. Assume that PDR returns the following counterexample sequence

State 0		State 1
$i_0 = 0, x_0 = 0$	$\rightarrow$	$i_0 = 0, x_0 = 0$
$i_1 = 0, x_1 = 0$		$i_1 = 0, x_1 = -1$

As the sequence has only one transition, it is clear that this one must be spurious. Similarly, as there is only one nested loop within loop 1, it is clear that the current invariant $\mathrm{LI}_2(\cdot)$ causes the discrepancy between model and program. We extract the following counterexample for loop 2 by projecting the variables appropriately:

$$c' = \{x_1 = 0, i_2 = 0, x_2 = -1, i_3 = 0\}$$

and call `refine()` recursively on loop 2 and c'. The procedure returns after having refined the invariant of loop 2 to

$$\mathrm{LI}_2(\cdot) = (x_1 < 0) \vee (x_2 > -1)$$

In the next iteration of the while-loop, we call the PDR-solver with the same model checking instance as above but with the updated $\mathrm{LI}_2(\cdot)$ within the transition relation. With this strengthening, the counterexample can be disproved and we can infer the loop invariant

$$\mathrm{LI}_1(\cdot) = (x_0 < 0) \vee (x_1 > -1)$$

from the trace of the PDR invocation.

When `refine()` is called the second time in `verifyProgram()`, the process of strengthening $\mathrm{LI}_1(\cdot)$ is similar with one significant difference: We start PDR with the trace constructed in the previous invocation. Therefore, the subspace $(x_0 \geq 0) \wedge (x_1 \leq -1)$ previously determined as unreachable is available. If, for instance, the next call of the QF_BV PDR solver was with the spurious counterexample

$$c = (x_0 \geq 2) \wedge (x_1 \leq 1)$$

then the cover in last frame of the PDR trace in the backend solver will eventually be as displayed in Figure 4.7a. This situation triggers a reshape-action as described in Section 3.2.2, yielding the cover in Figure 4.7b which corresponds to the invariant

$$\mathrm{LI}_1(\cdot) = (x_1 \geq x_0)$$

which is strong enough to prove the safety of the program.

(a) Before Reshape

(b) After Reshape

Figure 4.7: Cover in the last frame before and after the reshape-operation.

4.4 Implementation and Experimental Evaluation

In this section, we describe our practical work. We start with a discussion of our implementation in Section 4.4.1 and continue with presenting results obtained from experimentation with our implementation on benchmark programs contained in the distribution of INVGEN [GR09] and from the benchmark set of the SV-COMP competition on Software Verification [Bey12] in Section 4.4.2.

4.4.1 Implementation with LLVM

We implemented the presented algorithm in C++ and use functionality of the LLVM-framework [LA04] for parsing, representation, and analysis of the PUV. Working with the LLVM intermediate representation instead of a custom frontend for a specific language has the important advantage that we can process programs of any language for which an LLVM-frontend exists. For example, in addition to the C-language frontend we used in our experimentation, there exist frontends for the languages Fortran, Ada, Python, and Haskell.

The disadvantage of using the LLVM intermediate representation is that the correctness of the verification results depends on the correctness of the frontend compiler. In addition, even if correct, the compiler frontend is free to define the behavior of constructs that are undefined, indeterminate, or implementation defined by the C-standard [ISO99].

An example where this yields incorrect verification results in our setup with the C-fronted Clang [Cla] used in our experimentation is given in Figure 4.8. The program in Figure 4.8a is not safe because by the C standard, the value of `j` is indeterminate every time the loop body is entered. As such, `l` and `k` can get different values assigned in the loop and the assertion in the last line does not necessarily hold. However, a compiler is free to assign any value to `j`. Clang resolves the situation by allocating `j` in the basic block before entering the loop. This interpretation is equivalent with the safe program in Figure 4.8b. Though the initial value of `j` is indeterminate in the safe program as well, the value cannot change between iterations and the variables `l` and `k` are guaranteed to have identical values at the end of the loop and the assertion holds.

As the problem of detecting any kind of constructs that are undefined, implementation defined, etc. is undecidable, this can be considered a conceptional flaw of our verification strategy. Practically, however, most problematic constructs can be detected

```
int l;
int k;
for (int i=0; i < 2; ++i)
{
    int j;
    if (i == 0) l = j;
    if (i == 1) k = j;
}
assert (l == k);
```

(a) Unsafe Program

```
int l;
int k;
int j;
for (int i=0; i < 2; ++i)
{
    if (i == 0) l = j;
    if (i == 1) k = j;
}
assert (l == k);
```

(b) Safe Program

Figure 4.8: Safe and unsafe program compiled to the same LLVM intermediate representation.

statically and resolved for sound verification. Otherwise, the strategy employed by Clang to resolve problematic constructs is similar to the strategy employed by most other compilers. For instance, for the program in Figure 4.8a, it can be expected that most compilers will allocate `j` on the same position on the stack in all iterations, compiling the unsafe program in a safe manner. In this case, the wrong verification result does not matter because the program error will never manifest in practice.

4.4.2 Experimental Results

Table 4.1 summarizes runtime characteristics of the presented algorithm (referred to as Property Directed Program Verification or PDPV for short) for the set of benchmarks. Column 1 gives the name of the benchmark, column 2 the origin of the benchmark, and column 3 the lines of code as a crude measure of benchmark size. The next nine columns contain runtime statistics of our algorithm once both with counterexample generalization and memoization of reachability information in the PDR trace (default, columns 4-6), once without counterexample generalization (columns 7-9), and once without memoization (columns 10-12). For each configuration, we report runtime, peak memory requirement, and the percentage of the runtime spent in the QF_BV PDR model checker. As point of reference, columns 13 and 14 report running times and memory consumption when using the fully symbolic approach modeling the program counter explicitly as discussed in Section 4.2 on these benchmarks. We report timeout in case a verification attempt does not terminate within 600 seconds.

Benchmark	Repository	LOC	Property Directed Program Verification									Symbolic Encoding	
			default			no generalization			no memoization				
		(#)	Runtime (s)	Mem (MB)	PDR (%)	Runtime (s)	Mem (MB)	PDR (%)	Runtime (s)	Mem (MB)	PDR (%)	Runtime (s)	Mem (MB)
bind_expands_vars2	InvGen	59	1.38	62	3	3.28	71	88	1.61	63	2	timeout	timeout
bound	InvGen	47	0.71	57	4	2.3	60	99	3.94	62	11	287.27	448
disj_simple	InvGen	25	0.52	59	54	2.05	59	99	timeout	timeout	timeout	timeout	timeout
gulwani_cegar2	InvGen	42	8.55	103	79	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
heapsort1	InvGen	58	38.47	225	62	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
id_build	InvGen	34	0.18	57	22	2.64	64	97	1.65	63	44	168.81	240
id_trans	InvGen	43	0.48	71	10	1.75	73	82	0.5	72	10	timeout	timeout
mergesort	InvGen	72	0.02	51	0	0.01	51	0	0.01	51	0	timeout	timeout
nested1	InvGen	40	0.82	64	10	0.22	58	86	2.56	70	75	timeout	timeout
nested2	InvGen	39	0.79	64	10	0.23	58	87	2.33	70	74	timeout	timeout
nest-if1	InvGen	28	0.39	64	26	0.20	57	90	4.51	74	93	timeout	timeout
nest-len	InvGen	30	0.68	58	4	1.16	56	70	17.81	77	75	timeout	timeout
NetBSD_loop	InvGen	43	0.32	58	9	0.52	57	96	0.34	58	6	310.48	278
NetBSD_loop_int	InvGen	47	2.49	66	4	0.84	58	96	3.39	70	78	timeout	timeout
sendmail_close_angle	InvGen	104	5.54	73	3	26.10	139	98	11.12	89	1	timeout	timeout
sendmail_mime7to8	InvGen	89	6.73	79	8	2.94	62	88	timeout	timeout	timeout	timeout	timeout
simple	InvGen	38	0.54	59	56	1.04	57	97	527.05	109	100	timeout	timeout
simple_if	InvGen	26	0.55	58	42	0.44	57	93	0.66	59	35	14.95	85
simple_nest	InvGen	27	18.91	98	99	23.80	65	99	18.95	98	99	timeout	timeout
up_nested	InvGen	28	0.01	51	0	0.01	51	0	0.01	51	0	1.06	58
drevil_2	SV-COMP	21	1.51	72	80	2.85	70	77	2.15	70	77	timeout	timeout
jain_1	SV-COMP	17	1.80	59	88	1.53	58	97	1.56	58	89	timeout	timeout
jain_2	SV-COMP	19	3.50	70	91	3.40	66	97	7.72	63	99	timeout	timeout
for_bounded_loop	SV-COMP	28	0.01	51	0	0.01	51	0	0.01	51	0	8.44	92
trex01_safe	SV-COMP	50	0.51	57	45	1.02	54	67	0.98	58	67	timeout	timeout
trex01_unsafe	SV-COMP	50	0.01	50	0	0.01	50	0	0.01	50	0	timeout	timeout
trex04	SV-COMP	44	0.14	51	0	0.14	51	0	0.13	51	0	timeout	timeout

Table 4.1: Performance of Property Directed Program Verification

As can be seen, our verifier solves all verification tasks within reasonable runtime limits. Memory requirements are generally low, which is explained by the avoidance of unrolling both in the frontend and in the backend model checker of the program verifier. Generalization of counterexamples reduces the running time often substantially. In these cases, the runtime savings can be explained by the reduction of the burden on the backend model checker. Without the use of memoization, the algorithm is less stable, documented by several timeouts. With the fully symbolic encoding, only a small fraction of the benchmarks can be solved within the limit on running time and in these cases, the performance is substantially worse than the proposed verifier.

Chapter 5

Related Approaches To Program Verification

In this chapter, we discuss related approaches to program verification. Two techniques are particularly relevant with respect to the research discussed in this dissertation: In Section 5.1, we discuss a verification framework that, similar to PDPV, uses loop invariants to reduce a proof obligation for a safety property in a sequential program to a combinational verification problem that can be fed to a theory solver. In Section 5.2, we introduce a verification technique that is based on interpolation and has been used in conjunction with PDR as a hybrid verification technique. We will survey both techniques with a focus on comparing them to our technique.

5.1 Program Verification with Invariants

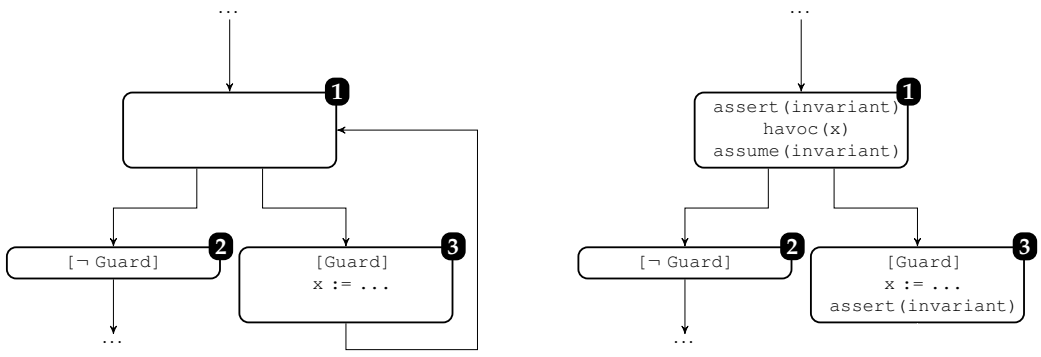
In this section, we start with outlining the loop-invariant-based framework in the following section. The success of the approach is contingent on whether or not sufficiently strong loop invariants are available. If loop invariants are not available, one can attempt to infer them automatically. In Section 5.1.2, we discuss techniques that have been devised for this purpose.

5.1.1 Overall Framework

The popular paradigm for program verification discussed in [BL05] proves or disproves a safety property in the PUV by converting the proof obligation in the CFG of the PUV into a logic formula which is unsatisfiable only if the program has the property of interest. The conversion from the original problem to the logic formula is similar to the technique we described in Chapter 4: The CFG is transformed into a directed acyclic graph (DAG), the derived DAG is passified, and the passive, combinational program is translated into a logic formula using weakest preconditions [Dij76] which can be fed to a theory solver.

In the first of the three transformation steps, backedges of loops in the PUV are cut and one attempts to capture the meaning of the loops using loop invariants. The format of the loop invariants differs from the one applied in our framework. In the framework discussed in [BL05], any program variable can be in the support of the loop invariant while in PDPV, only variables referenced in the loop can be in the support of the loop invariant. Also, in the framework in [BL05], loop invariants constrain the values in the variables at any time the control visits the loop header. In PDPV, the loop invariants relate the possible values of the variables after the loop with those at the beginning. These two differences are significant as they enable the local reasoning applied to refine loops discussed in Section 4.3.3.

Figure 5.1 illustrates the ideas in [BL05] graphically: For each loop, the corre-



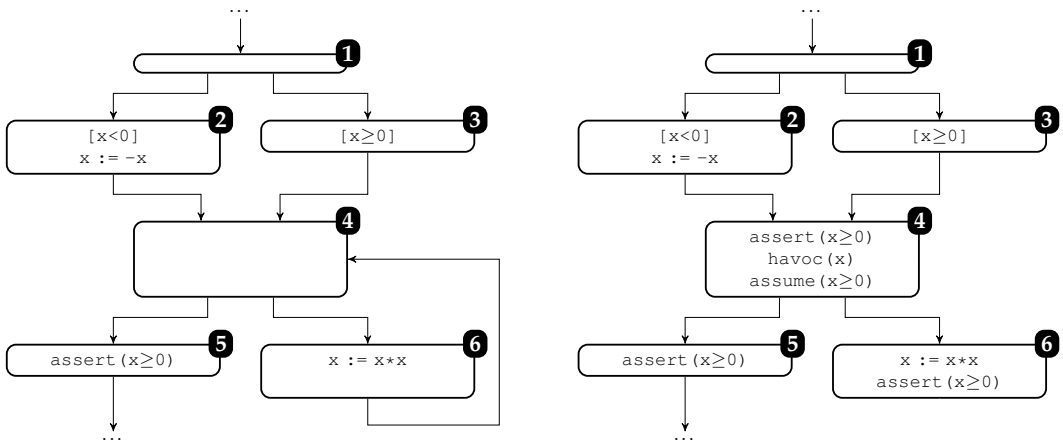
(a) Original Sequential Program (b) Combinational Model

Figure 5.1: Derivation of a combinational model from a sequential program.

sponding backedge is cut, variables updated in the loop are allowed to take arbitrary values (indicated by the `havoc`-statement in the figure), and the values are restricted there-

after by assuming the loop invariant. If the supplied loop invariants are too strong, the framework can produce false positives (an unsafe program is declared safe). To exclude this possibility, the initiation and consecution conditions are verified by adding assertions in the loop header and body. Note that in PDPV, adding these conditions would be redundant as loop invariants are correct by construction.

To illustrate the above, consider the concrete example in Figure 5.2a where one attempts to prove that the assertion in basic block 5 holds regardless of the value of x at the beginning of the code fragment. Cutting the backedge, adding the havoc-statement, and applying the loop invariant $x \geq 0$ yields the combinational model in Figure 5.2b. It is easy to see that all assertions hold regardless of the initial value of x in this model. This proves both the original proof obligation and that the applied loop invariant are valid.



(a) Original Sequential Program

(b) Combinational Model

Figure 5.2: Verifying a program with a loop invariant

The success of the discussed framework to program verification depends on the existence of loop invariants strong enough to prove the desired properties of the PUV. Unfortunately, loop invariants are often not available and despite arguments from the academic community in favor of program annotation (see e.g. [DF88]), developers are usually not willing or able to provide loop invariants to the verification system. This instigated interest for automatic inference of loop invariants. We will survey relevant approaches to loop inference in the following section.

5.1.2 Approaches to Invariant Inference

There is a rich body of literature describing approaches to automatically inferring loop invariants in programs. Herein, we describe the three most relevant approaches. We start with the most established approach, abstract interpretation, which is not only used for program verification but in a larger number of applications such as static analysis and compiler optimization. Subsequently, we describe a more recent dynamic approach which is particularly targeted to applications where the desired invariants are relatively simple. Finally, we discuss a similar but static approach to invariant inference which represents the conceptual basis of INVGEN [GR09], a popular tool for inferring linear loop invariants.

Abstract Interpretation

Traditionally, a vehicle for invariant inference is abstract interpretation as originally introduced in [CC77]. In abstract interpretation, one selects an abstract domain $\mathcal{A}$, abstraction and concretization functions, as well as an abstract transition relation. Given the abstract transition system, one calculates the least fixed point to obtain invariants in the abstract domain. These can be transformed to invariants of the PUV using the concretization function.

To illustrate the technique to invariant inference, reconsider the example in Figure 5.2. Assume we chose $\mathcal{A} = \{\bot, (-), (+), (\pm)\}$ which relates to the concrete domain of x as given in Table 5.1. We start the calculation of the least fixed point assuming the

Abstract Domain	Concrete Domain
$\bot$	$x \in \emptyset$
$(-)$	$x < 0$
$(+)$	$x \geq 0$
$(\pm)$	$x \in [-\infty, \infty]$

Table 5.1: Relation between Abstract and Concrete Domains of x

bottom element for x at each location in the program (see Figure 5.3a). At the beginning of the program, x an take any value. In the abstract domain, this corresponds to $(\pm)$ (see Figure 5.3b). In the next iteration of the least fixed point calculation, the $(\pm)$-value at the input of basic block 1 is propagated to the fanout of this basic block, as basic block 1 cor-

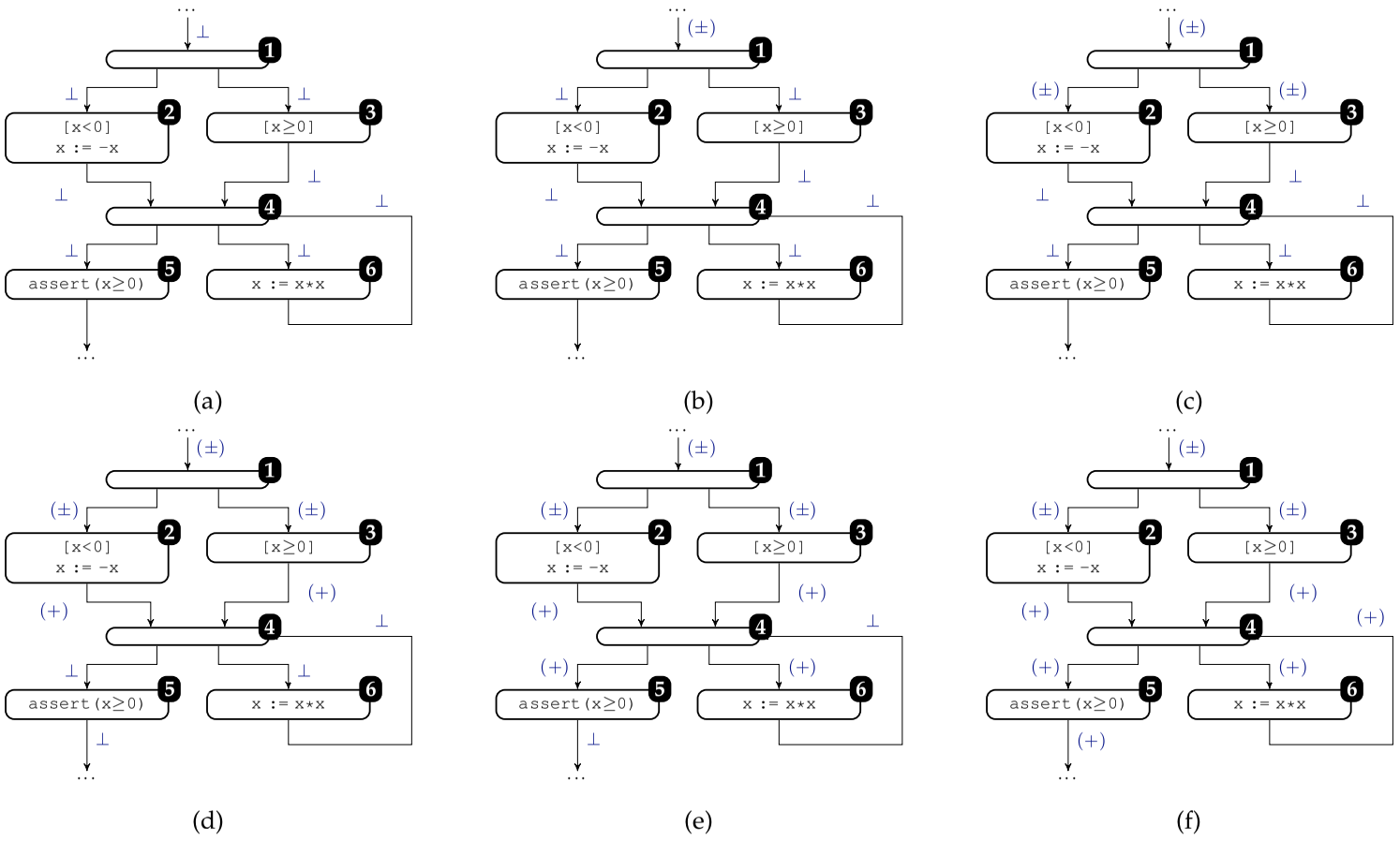


Figure 5.3: Inference of a Loop Invariant with Abstract Interpretation

responds to an identity function in the abstract transition relation (see Figure 5.3c). In the next iteration, the abstract values at the fanout of basic blocks 2 and 3 are updated. In both cases, x must be greater or equal than zero, reflected as $(+)$ in the abstract domain (see Figure 5.3d). In the next iteration, these values are propagated through basic block 4 (see Figure 5.3e). As multiplying two positive values yields a positive value, the abstract value at the fanout of basic block 6 is calculated to $(+)$ in the next iteration (see Figure 5.3f). At this point, applying the abstract transition relation does no longer update any abstract value, i.e. we have found the least fixed point. The abstract value at the loop header is $(+)$, hence we can infer that $x \geq 0$ is an invariant of the loop in the program fragment.

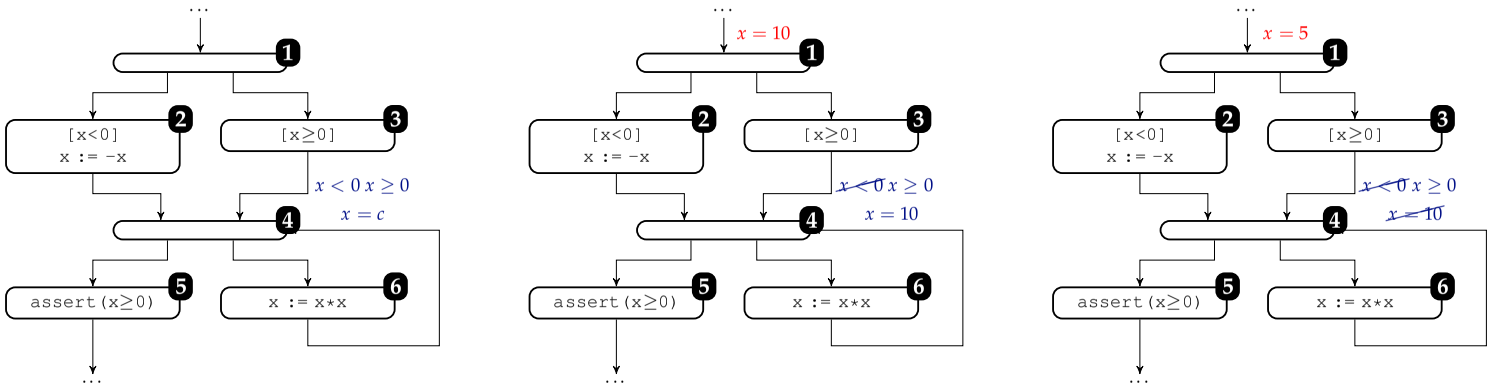
The main practical challenge associated with using abstract interpretation to invariant inference is to find a suitable abstract domain for the specific PUV and properties of interest. To increase the probability that abstract interpretation infers useful aspects about loops, many different abstract domains can be used. In any case, however, there is no guarantee that abstract interpretation will yield strong loop invariants.

Dynamic Invariant Inference

In the DAIKON-approach [ECGN01], one assumes that a test set for the PUV is available. The algorithm consists of three phases. In the first phase, DAIKON compiles a set of likely invariants. The set is based on experience with typical programs and contains for instance hypotheses about a single variable x such as $x = c$, $x \neq 0$, $x \in [a, b]$, etc. or hypotheses about pairs of variables x, y such as $x \leq y$, $ax + by = c$, etc. In the second phase, the test set is used to filter out candidate invariants that contradict the observed behavior. In the third phase, the remaining candidates are verified using a theory solver.

Figure 5.4 illustrates how DAIKON could be used to infer loop invariants for the example. Assume that in the first phase of the algorithm, the candidates $x < 0$, $x \geq 0$, and $x = c$ are generated. In the second phase, DAIKON interprets the program fragment with the test set containing two tests with x being initially 10 and -5. Interpreting the first test allows DAIKON to filter out the candidate $x < 0$ and to specialize the candidate $x = c$ to $x = 10$. Applying the second test additionally cancels candidate $x = 10$. In the third phase of the algorithm, DAIKON verifies with a theory solver that $x \geq 0$ is indeed a valid invariant of the loop in the program fragment.

In practice, DAIKON has two limitations: First, it can only find loop invariants



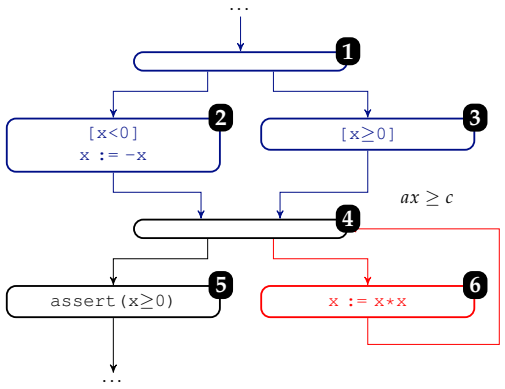
(a) (b) (c)

Figure 5.4: Inference of a Loop Invariant with DAIKON

that are in the initial set of candidates. Second, DAIKON requires a strong test set such that the set of candidates can be reduced to a manageable number. Otherwise, the verification phase will require unacceptably long.

Template Approaches

A static alternative to the DAIKON-approach are template solutions. In these approaches, one assumes a certain parametrized form of the loop invariants, constructs a suitable constraint system that potential parameters must obey to, and uses a theory solver to solve for the parameters. One implementation of this approach is discussed in [CSS03] where the authors assume that the loop invariants can be formulated as linear inequalities.



(a)

Initiation:

$$x < 0 \Rightarrow -ax \geq c$$

$$x \geq 0 \Rightarrow ax \geq c$$

Consecution:

$$ax \geq c \Rightarrow ax^2 \geq c$$

(b)

Figure 5.5: Inference of a Loop Invariant with Template Approach

We illustrate the technique for our example in Figure 5.5 where we assume that the invariant is of the form $ax > c$. To be valid, the invariant needs to adhere to two sets of constraints, pertaining initiation and consecution. There are two ways of reaching the loop initially, through basic blocks 2 and 3. The path through basic block 2 is only taken if $x < 0$. Also in basic block 2, the sign of x is flipped. When reaching basic block 4, the invariant must hold, hence we get the constraint $x < 0 \Rightarrow -ax \geq c$. The constraint for the path through basic block 3 is constructed in analog fashion. To assure consecution, we need to add a constraint for the loop through basic block 6. Here we can assume that the invariant $ax \geq c$ holds when leaving the loop header. x is squared in the loop. Hence, to assure that invariant holds when reentering the loop, we must have $ax^2 \geq c$. To find

possible parameters a and c, the three constraints are solved using a theory solver. It is easy to verify that with $a = 1$ and $c = 0$, the constraints hold for any value of x, yielding the same invariant $x \geq 0$ as with the other discussed approaches for invariant inference.

A well-known tool that uses the template approach as main reasoning engine is INVGEN [GR09]. After applying a couple of dynamic strategies, INVGEN sets up a system of equations and solves for the template parameters. This hybrid approach yields a relatively robust algorithm. However, INVGEN has three main limitations: First, INVGEN commits to conjunctions of linear inequalities as loop invariants. If the invariant required to prove a program safe is of any other form, the algorithm fails. Second, INVGEN models programs using the quantifier free theory of linear arithmetic (QF_LA). Consequently, INVGEN does not support programs with bit-level operations and the verification results are not guaranteed to be correct, e.g. in the presence of possible overflows. Third, INVGEN is not able to generate counterexamples. If the verification fails, it is not clear if the program is unsafe or if INVGEN failed.

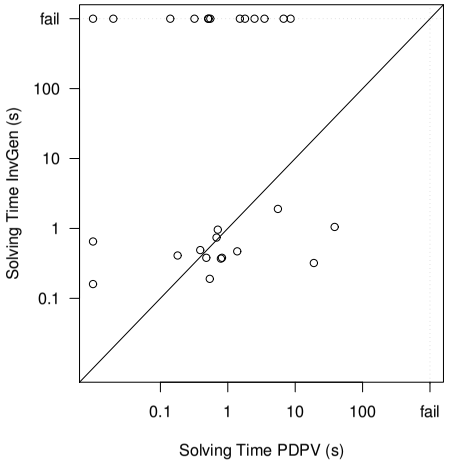
Figure 5.6 contains an empirical comparison between PDPV and INVGEN. As in Chapter 4, we report timeout in case the program did not terminate within 600 seconds. Additionally, we report error if the program terminated without providing a result (either by giving up or due to a crash) and false positive if an unsafe program is reported as safe. INVGEN terminates faster than PDPV in the majority of the cases but fails in roughly 40% of the benchmarks. In one case, it returns an incorrect verification result.

Detailed Study of Selected Benchmarks INVGEN uses linear arithmetic to model bitvector arithmetic. The consequences of this inaccurate modeling can be studied e.g. in the example `NetBSD_loop.c` of which a simplified excerpt is given in Figure 5.7a. Here, INVGEN disregards the possible overflow in the addition of the second assertion and returns that the problem is save. On the other side, PDPV returns a counterexample stimulating this overflow.

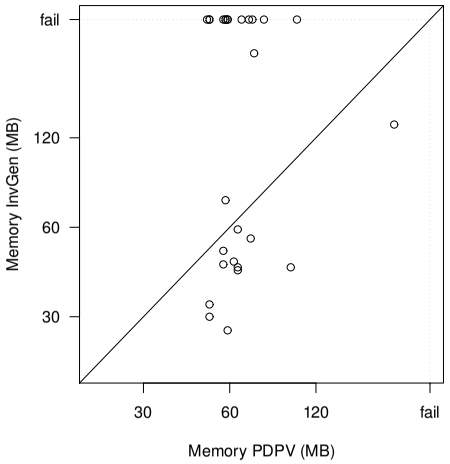
The example `disj_simple.c` for which a simplified excerpt is given in Figure 5.7b is safe in the theory QF_LA. However, INVGEN fails in constructing the proof because it can only find invariants that are conjunctions of linear inequalities by default. Otherwise, the tool requires template annotation and performs less efficient as most optimization gets disabled. In particular, invariants that are disjunctions represent a challenge for INVGEN and many other program verification tools alike but are often needed in prac-

Benchmark	PDPV		INVGEN [GR09]	
	Runtime (s)	Mem (MB)	Runtime (s)	Mem (MB)
bind_expands_vars2	1.38	62	0.47	46
bound	0.71	57	0.96	45
disj_simple	0.52	59	error	error
gulwani_cegar2	8.55	103	error	error
heapsort1	38.47	225	1.05	133
id_build	0.18	57	0.41	50
id_trans	0.48	71	0.38	55
mergesort	0.02	51	timeout	timeout
nested1	0.82	64	0.38	44
nested2	0.79	64	0.37	43
nest-if1	0.39	64	0.49	59
nest-len	0.68	58	0.74	74
NetBSD_loop	0.32	58	fal. pos.	fal. pos.
NetBSD_loop_int	2.49	66	fal. pos.	fal. pos.
sendmail_close_angle	5.54	73	1.90	231
sendmail_mime7to8	6.73	79	error	error
simple	0.54	59	0.19	27
simple_if	0.55	58	error	error
simple_nest	18.91	98	0.32	44
up_nested	0.01	51	0.65	30
drevil_2	1.51	72	error	error
jain_1	1.80	59	error	error
jain_2	3.50	70	error	error
for_bounded_loop	0.01	51	0.16	33
trex01_safe	0.51	57	error	error
trex01_unsafe	0.01	50	error	error
trex04	0.14	51	error	error

(a) Detailed Comparison



(b) Comparison Runtime



(c) Comparison Memory

Figure 5.6: Comparison Property Directed Program Verification vs. INVGEN

```
if(lenPath>0)
{
  off=len;

  for(i=0;
      i<=off;
      i++)
  {
    assert(0<=i);
    assert(i<len+1);
  }
}
```

(a) NetBSD_loop.c

```
int x=0;
int n;

while(x<n)
{
  x++;
}

if(n>0)
{
  assert(x<=n);
}
```

(b) disj_simple.c

Figure 5.7: Program fragments for which INVGEN fails but PDPV succeeds.

tice [SDDA11]. To prove the specific example in Figure 5.7b safe, the disjunctive invariant $(x = n) \vee (n < 0)$ must be inferred. Even with a suitable template annotation, INVGEN fails in calculating the invariant. In contrast, disjunctive invariants do not represent a structural challenge for PDPV and hence it is able to verify the program quickly.

At the cores of all three surveyed approaches to invariant inference rest certain assumptions about the form of the invariants; in abstract interpretation by defining the abstract domain, in DAIKON by the compiled set of likely invariants, and in the template approaches by the actual choice of the template. These assumptions make the inference process tractable but at the same time limit their generality as no invariants can be found that do not match these assumptions. Another common trait of all three approaches is that they are not directed towards finding invariants useful in proving certain properties in a program. This is in stark contrast to our verification framework, which does not suffer from these limitations. In principle, PDPV is able to find any invariant and it is directed towards resolving the actual proof obligations.

5.2 Program Verification with Interpolation

The software verification algorithm IMPACT [McM06] constructs an unwinding of the CFG that is subsequently annotated with reachability information calculated using Craig Inter-

polation [Cra57].

Figure 5.8 illustrates how IMPACT proves that our running example from Figure 5.2a is safe. Via depth-first search, IMPACT traverses the CFG until an error location is reached. Along the way, a tree T is constructed. Each vertex of T corresponds to a program location after a basic block is executed. In our illustration, we denote the i^{th} vertex that corresponds to the program location after basic block B_j with j_i. Each vertex in T is associated with a predicate that holds in the corresponding program location. Initially, all vertices are annotated with the predicate $\top$. For our example, assume that the path through basic blocks B_1, B_2, and B_4 to the error location is explored first. (see Figure 5.8a).

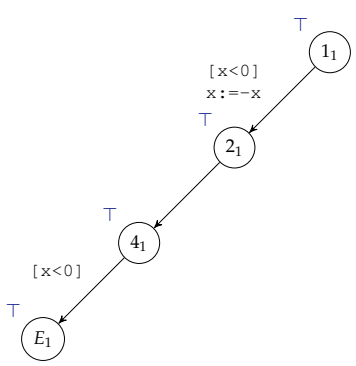
For each error path, IMPACT constructs an SMT formula that is satisfiable iff the path is viable and the SMT formula is fed to an SMT solver. If the SMT formula is satisfiable, the returned model encodes a counterexample proving that the program is not safe. Otherwise, IMPACT applies an interpolation algorithm to derive reachability information from the proof of unsatisfiability derived by the SMT solver. An interpolant I for an inconsistent SMT formula composed of conjuncts A and B is a formula that is implied by A ($A \Rightarrow I$), implies $\neg B$ ($I \Rightarrow \neg B$), and is over variables that are in the supports of both A and B. For the error path in Figure 5.8a, we could derive the SMT formula

$$\underbrace{(x_0 < 0) \wedge (x_1 = -x_0)}_{A} \wedge \underbrace{(x_1 < 0)}_{B} \tag{5.1}$$

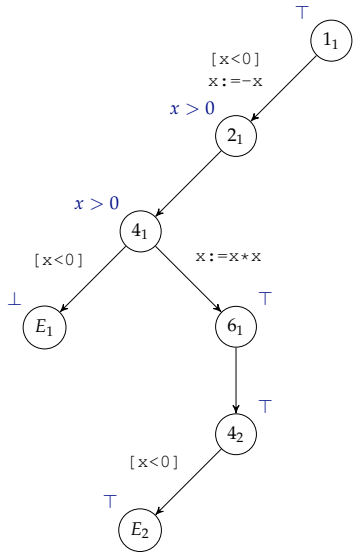
that is clearly inconsistent. To derive e.g. reachability information for vertex 2_1, we partition the SMT formula as indicated in equation (5.1) and apply an interpolation algorithm to obtain the interpolant $x_1 > 0$.

The annotation for all vertices on the first error path is contained in Figure 5.8b along with the second error path that is resolved using the same steps yielding the annotation as in Figure 5.8c. Note that vertices 4_1 and 4_2 correspond to the same program location and that the annotation of vertex 4_2 implies the annotation of vertex 4_1. Intuitively, this means that everything that can be reached from vertex 4_2 can also be reached from vertex 4_1 and we can abstain from further expanding 4_2. In this situation, we say that 4_2 is covered by 4_1 and we indicate the relationship with a dotted directed edge in our illustrations. IMPACT terminates if all vertices are either fully expanded or covered. For our example, this yields the unwinding in Figure 5.8c.

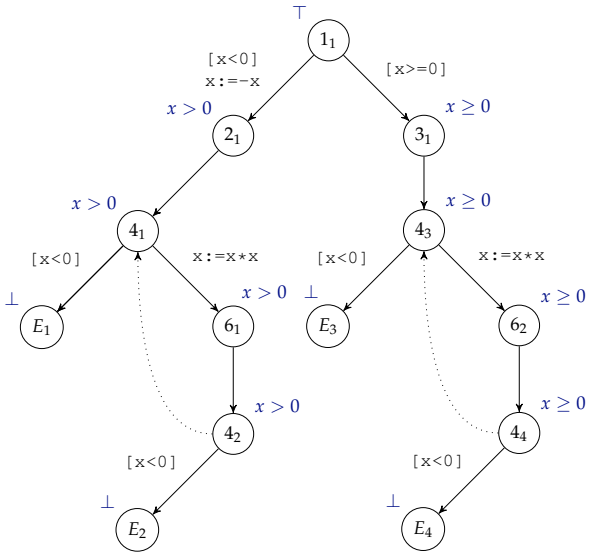
In [CG12], this verification algorithm is extended by proposing a procedure simi-



(a) First Error

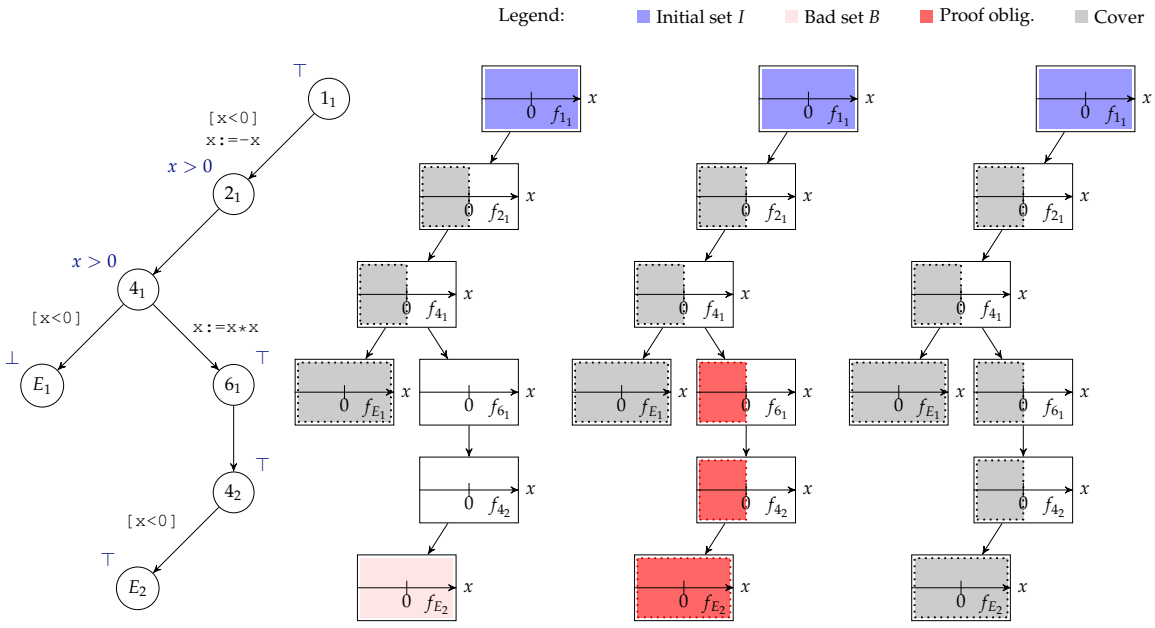


(b) Second Error



(c) Termination

Figure 5.8: Stages of Unwinding of IMPACT



(a) Second Error (b) Initial State (c) Intermediate State (d) Termination

Figure 5.9: Application of TREEIC3 for Annotation

lar to PDR (referred to as TREEIC3) as a secondary method for annotating the unwinding. To this end, one constructs a trace-tree where each frame corresponds to a vertex in the unwinding T. The cover of a frame f_{j_i} indicates which states are unreachable in the corresponding program location j_i. Given an error path, the objective is to cover a frame in the trace-tree corresponding to a program location on the error path entirely, effectively proving that the path is not viable. As in the PDR algorithm, TREEIC3 resorts to local reasoning involving at most two frames at a time. The role of the transition relation is taken by the statements executed between two program locations. The principle is illustrated in Figure 5.9 where we assume that the first error path was already processed and one attempts to strengthen the annotation to exclude the second error path to E_2. Figure 5.9b contains the trace-tree constructed for this purpose. Note that the frames f_{2_1}, f_{4_1}, and f_{E_1} reflect the annotations in Figure 5.9a. We attempt to prove that E_2 is not reachable. By means of a call of FINDBADCUBE(), we obtain the proof obligation $\top$ in f_{E_2} and using a procedure similar to RECCOVERCUBE(), we infer the additional proof obligations in f_{6_1} and f_{4_2}. (see Figure 5.9b). No state in the proof obligation in frame f_{6_1} can be reached from frame f_{4_1}, hence the proof obligations can be covered. Similar reasoning allows to cover

the remaining proof obligations in frames f_{4_2} and f_{E_2} and to arrive at the terminal state in Figure 5.9d.

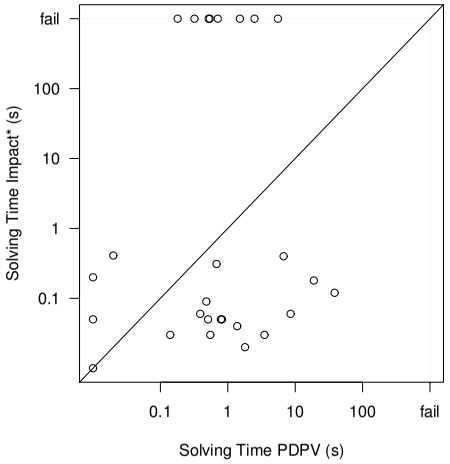
The authors of [CG12] propose to use the PDR-inspired annotation scheme as a secondary annotation method if the original interpolation-based annotation method performs poorly. Using this hybrid scheme, the verification algorithm outperforms the original version of the algorithm with exclusive use of interpolation on a large fraction of benchmarks, often substantially. In the following, we refer to the formulation of the algorithm with both annotation schemes as IMPACT* while we continue to refer to the original formulation with exclusive use of interpolation for annotation as IMPACT.

There are two main differences between our algorithm and IMPACT*: First, PDPV calculates invariants for real locations in the PUV while IMPACT* calculates invariants for nodes in an unrolling of the CFG. In addition to the potential efficiency problems due to this unrolling described in [McM10], restricting to real program locations as in PDPV has the advantages that the invariants can be initialized with potentially known invariants, that the proofs generated by the verifier are easier to comprehend, and that the inferred invariants can be used for further analyses or optimization. Second, as IMPACT* calculates reachability information partially using interpolation, its success rests on the availability of an efficient interpolation algorithm. Such an interpolation procedure is not known for QF_BV. To avoid bit-blasting and the application of a Boolean interpolation algorithm such as presented in [McM03], IMPACT* resorts to an interpolation algorithm for the quantifier free theory of linear rational arithmetic (QF_LRA) [McM04]. This choice gives a very efficient algorithm, but does not allow for accurate modeling of any bitvector arithmetic. A quantitative comparison to PDPV is contained in Figure 5.10. IMPACT* outperforms our algorithm in the majority of the benchmarks. However, the algorithm yields wrong verification results (false positives or false negatives) in roughly 30% of the cases.

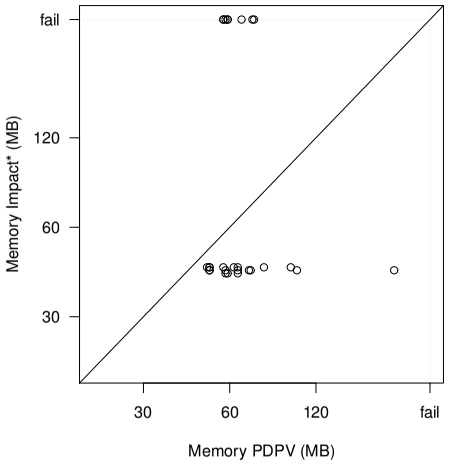
The problems leading to the large number of bogus verification results can be illustrated using the same examples as in Section 5.1.2. Similarly to INVGEN, IMPACT* is ignorant to the potential overflow in the program `NetBSD_loop.c` and erroneously returns that the program is safe. For the example `disj_simple.c`, IMPACT* does not suffer from the limitations regarding disjunctive invariants but returns a spurious counterexample as an artifact of modeling the program using rational arithmetic: In this interpretation, the program is indeed not safe. For instance, `n` can initially be assigned 0.5. This allows for a run where x gets incremented once and the assertion fails.

Benchmark	PDPV		IMPACT* [CG12]	
	Runtime (s)	Mem (MB)	Runtime (s)	Mem (MB)
bind_expands_vars2	1.38	62	0.04	44
bound	0.71	57	fal. neg.	fal. neg.
disj_simple	0.52	59	fal. neg.	fal. neg.
gulwani_cegar2	8.55	103	0.06	43
heapsort1	38.47	225	0.12	43
id_build	0.18	57	fal. neg.	fal. neg.
id_trans	0.48	71	0.09	43
mergesort	0.02	51	0.41	44
nested1	0.82	64	0.05	43
nested2	0.79	64	0.05	44
nest-if1	0.39	64	0.06	42
nest-len	0.68	58	0.31	43
NetBSD_loop	0.32	58	fal. pos.	fal. pos.
NetBSD_loop_int	2.49	66	fal. pos.	fal. pos.
sendmail_close_angle	5.54	73	fal. neg.	fal. neg.
sendmail_mime7to8	6.73	79	0.40	44
simple	0.54	59	fal. neg.	fal. neg.
simple_if	0.55	58	0.03	42
simple_nest	18.91	98	0.18	44
up_nested	0.01	51	0.05	43
drevil_2	1.51	72	fal. neg.	fal. neg.
jain_1	1.80	59	0.02	42
jain_2	3.50	70	0.03	43
for_bounded_loop	0.01	51	0.20	43
trex01_safe	0.51	57	0.05	44
trex01_unsafe	0.01	50	0.01	44
trex04	0.14	51	0.03	44

(a) Detailed Comparison



(b) Comparison Runtime



(c) Comparison Memory

Figure 5.10: Comparison Property Directed Program Verification vs. IMPACT*

Chapter 6

Initialization of Loop Invariants

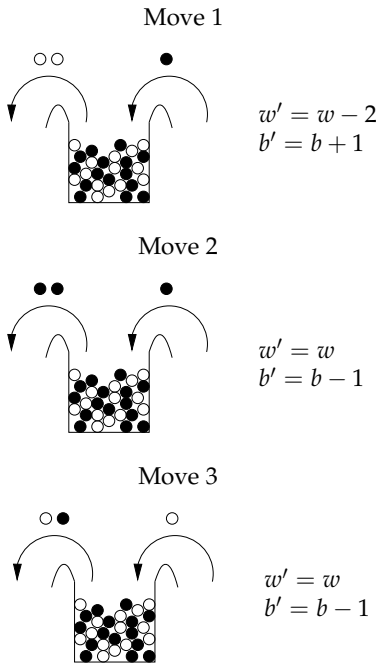
At the core of the verification framework described in Chapter 4 rests the counterexample guided strengthening of loop invariants using Property Directed Reachability. As described there, the loop invariants are initially assumed to be true, i.e. the initial loop model allows arbitrary valuations of its loop variables before and after the loop.

A promising extension to this approach is to initialize the loop invariants. This may speed up the solving of the model checking instances in the PDR backend or avoid calls to the PDR backend altogether.

In this chapter, we start with discussing how to integrate initial loop invariants into our framework for program verification in the following section. Next, we discuss several options to obtain loop invariants in Section 6.2. Finally, in Section 6.3, we describe a case study where we use one specific tool, DAIKON [ECGN01], to obtain loop invariants and assist the solving of the benchmark instances we introduced in Chapter 4.

6.0.1 Dijkstra's Urn Example

We will illustrate the discussion in this chapter using Dijkstra's urn example as presented in [Dij90] (see Figure 6.1). Assume you are given an urn filled with white and black balls. Every iteration, two arbitrarily selected balls are taken out of the urn. If the two balls are white, one of the two balls is painted black and returned to the urn (Move 1). If the two balls are black, one of the black balls is returned to the urn (Move 2). Otherwise, if a black and a white ball were selected, only the white ball is returned to the urn (Move 3). Note that the outcomes of the second and the third moves are equal with respect to



(a) Illustration of possible moves

```
int
urn(int w, int b)
{
    assume (w >= 0);
    assume (b >= 0);
    assume (w + b >= 2);
    int z = w;
    while (w + b >= 2)
    {
        if (* && w >= 2)
        {
            // Move 1
            w -= 2;
            b += 1;
        }
        else if (b >= 1)
        {
            // Move 2 or 3
            b -= 1;
        }
    }
    if (z & 1)
    {
        assert(b == 0 && w == 1)
    }
    else
    {
        assert(b == 1 && w == 0)
    }
}
```

(b) `urn.c`

Figure 6.1: Dijstra's urn example

the difference in numbers of black and white balls in the urn before and after the move. The program in Figure 6.1b models the urn example where we used the $\star$-symbol in the if-condition to indicate the random selection. The property of interest pertains the state of the urn after termination of the while-loop: If the count of white balls is initially odd, only a single white ball remains in the urn. Otherwise, only a single black ball remains in the urn. The property is encoded in the assertions at the end of `urn.c`.

The while-loop in function `urn()` has many invariants. With respect to our proof obligations, the most important ones are that the number of white or black balls can never become negative ($w \geq 0$, $b \geq 0$) and that the parity of the number of white balls does not change (z odd $\Leftrightarrow$ w odd). These three invariants and the fact that the loop is only left if $w + b < 2$ immediately imply the assertions, i.e. prove that the program is safe.

6.1 Integration of External Loop Invariants

As discussed in Chapter 5, loop invariants are traditionally formulated in terms of program variables. In contrast, PDPV encodes loop invariants as predicates over two copies of loop variables, holding their values before and after loop, only. As in previous chapters, we refer to the variant holding the value of a variable v before (at the end of) the loop as v_b (v_e).

In Dijkstra's urn example, we have the program variables w, b, and z. Only w and b are loop variables, hence PDPV expects loop invariants over w_b, w_e, b_b, and b_e.

If a given loop invariant in the traditional formulation holds for a specific loop, it holds before and after the loop, i.e. the loop invariant can be transcribed into the format expected by PDPV both in terms of the variants holding the value before and after the loop. For instance, the loop invariant $w \geq 0$ can be reflected by adding $w_b \geq 0$ and $w_e \geq 0$.

Internally, loop invariants for a specific loop l are associated with the PDR-trace corresponding to l. Denoting the set of invariants with I, they are reflected in subroutines as follows:

- For finding a bad ARU that is not yet covered, one uses the following generalization of equation (2.1)

$$B \wedge \neg \bigvee_{c_i \in F_l} c_i \wedge \bigwedge_{i \in I} i \tag{6.1}$$

  where, in addition to specifying that the ARU must be in bad and is not yet covered, one also specifies that all invariants must hold.

- Similarly, one adds the invariants to the check if a proof obligation in frame l is reachable from the previous frame $l-1$ to arrive at the following generalized form of equation (2.2):

$$\neg \bigvee_{c_i \in F_{l-1}} c_i \wedge \bigwedge_{i \in I} i \wedge T \wedge c' \tag{6.2}$$

  Note that adding a term such as $\bigwedge_{i \in I} i'$ to (6.2) would be redundant as c' is guaranteed to adhere to all invariants in I as it was previously found using (6.1) or (6.2).

6.2 Origin of Initial Loop Invariants

The most immediate option to obtain loop invariants is by program annotation. Even though program annotation is generally unpopular in practice, adding a few loop invariants in safety-critical code that otherwise fails verification may be considered acceptable. In some contexts, loop invariants may also be readily available in particular if automatic code generation is used.

Otherwise, loop invariants can be inferred by any of the approaches discussed in Section 5.1.2. The fact that checking loop invariants is computationally easier than inferring them is making this alternative viable as it allows to check invariants which may or may not be correct. To name an example where this matters, consider INVGEN [GR09]. We have seen in Section 5.1.2, that the fact that INVGEN uses the theory QF_LA rather than QF_BV can yield wrong verification results. However, even though a wrong theory is used, this does not imply that the inferred invariants are actually wrong. If the initialization and consecution tests pass using the theory QF_BV, the invariants can be used regardless of how the invariant was calculated.

Extending our verification framework by using invariants inferred with DAIKON is a particular promising approach. As described in Section 5.1.2, DAIKON executes a dynamic analysis to obtain candidates for loop invariants. On the other hand, our framework executes a purely static analysis. It is generally accepted in the community that static and dynamic tools have complementary advantages and disadvantages. Using the loop invariants inferred by DAIKON in our static tool may combine the strengths of both paradigms.

6.3 Initializing Loop Invariants with DAIKON

In this section, we describe a case study where we use DAIKON to infer loop invariants for the benchmark set introduced in Chapter 4. We start with discussing some technical details of our experimental setup in the next subsection and report experimental results in Subsection 6.3.2.

6.3.1 Implementation

The generation of loop invariants with the DAIKON invariant generator is associated with two technical challenges. First, DAIKON assumes the availability of test sets but we did

not have any tests for our benchmarks. Second, DAIKON does not support the generation of loop invariants but that of function invariants only.

We resolved the first challenge by automatically generating appropriate test sets for all benchmark problems. To this end, we applied procedure GENERATETESTS() as coded in Algorithm 6.1. As in Chapter 4, we denote with $-\infty$ and ∞ the minimally and maximally representable value within the type of a specific argument. Further, we use the function RAND(a, b), that returns integers selected uniformly at random in the interval $[a, b]$.

Algorithm 6.1 GENERATETESTS(t, a)

```
1:  for each i in 1 to t do
2:      test = []
3:      for each j in 1 to a do
4:          switch (r = RAND(1,9))
5:              case r = 1: test.APPENDARGUMENT(0)
6:              case r = 2: test.APPENDARGUMENT(-1)
7:              case r = 3: test.APPENDARGUMENT(1)
8:              case r = 4: test.APPENDARGUMENT(−∞)
9:              case r = 5: test.APPENDARGUMENT(∞)
10:             case r = 6: test.APPENDARGUMENT(RAND(-10,-2))
11:             case r = 7: test.APPENDARGUMENT(RAND(2,10))
12:             case r = 8: test.APPENDARGUMENT(RAND(−∞+1,-11))
13:             case r = 9: test.APPENDARGUMENT(RAND(11,∞-1)))
14:         end case
15:     end for
16:     OUTPUT(test)
17: end for
```

When called, GENERATETESTS(t, a) composes t tests with a arguments at random, but oversamples corner values such as -1, 0 and ∞ for the arguments. In our experiments, we have observed that this yields a better test coverage for the benchmark problems than a purely random argument generation within the allowed intervals of the arguments. A higher test coverage in turn yields better pruning of potential invariants in the second phase of the DAIKON approach as described in Section 5.1.2.

```
...
int z = w;
while (w + b >= 2)
{
    while_loop_head(z, w, b);
    if (* && w >= 2)
    {
        // Move 1
        w -= 2;
        ...
```

(a) `urn.c` with Call to Dummy Function

```
void
while_loop_head(int z,
                int w,
                int b)
{
    ;
}
```

(b) Dummy Function

Figure 6.2: Example Insertion of Dummy Function Calls

To cope with the second challenge with DAIKON not generating loop invariants by default but only function invariants, we add dummy function calls at the loop headers. As arguments, the dummy functions receive all variables relevant to a specific loop. Once inserted, DAIKON generates function invariants for these dummy functions which are semantically equivalent to the desired loop invariants.

For the program modeling Dijkstra's urn example, we add the function call to `while_loop_head()` at the beginning of the while-loop as illustrated in Figure 6.2. The function itself does not do anything but once inserted, DAIKON generates a function invariant for it. These function invariants generated for `while_loop_head()` correspond to the desired loop invariants.

With this setup, DAIKON finds the invariants $w \geq 0$ and $b \geq 0$ for the while-loop. However, it fails in finding the third required invariant (z odd $\Leftrightarrow$ w odd) as DAIKON does not provide an appropriate template constraint for this type of invariant.

6.3.2 Experimental Evaluation

To measure the impact of the DAIKON invariants on the solving times of the INVGEN and SV-COMP benchmarks introduced in Chapter 4, we generated DAIKON invariants for the problems and compared the solving times with and without their use. The scatter plot in Figure 6.3 summarizes the results.

In our experimentation, we made the following observations: For the majority of roughly 80% of the benchmarks, either DAIKON does not generate any loop invariants or the generated loop invariants are useless for solving the proof obligation. In these cases,

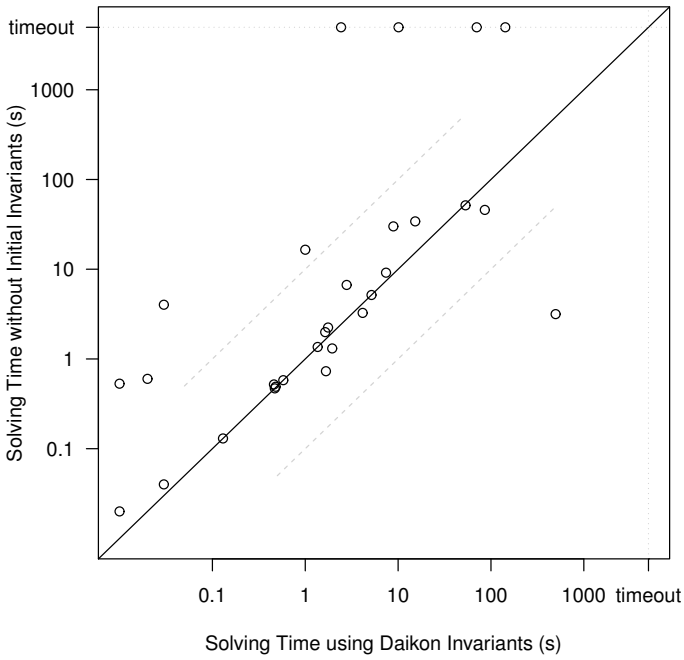


Figure 6.3: Impact of Initialization with DAIKON Invariants on Solving Time.

initializing the loop invariants has usually only marginal impact on the running time. If there is an impact on the running time, it is typically due to the solving times of the back-end SMT-solver which may increase or decrease by the additional constraints added. One notable exception is `jain_1.c`, where the running time increased by almost two orders of magnitude due to the additional constraints to the SMT solver. For the remaining benchmarks, the invariants generated by DAIKON simplify the proof obligations substantially and the solving time decreases, often by more than an order of magnitude. This is e.g. the case in Dijkstra's urn example where the two invariants $w > 0$ and $b > 0$ cut off three quarters of the search space and effectively direct the PDR algorithm towards finding the last invariant quickly. In the extreme case, the invariants inferred by DAIKON were strong enough to solve the benchmarks without the need of any invariant refinement by PDR. For the set of used benchmarks, this has been the case three times.

Chapter 7

Conclusions

In this dissertation, we investigated the potential of using Property Directed Reachability (PDR) for program verification. More concretely, we devised a sound and complete algorithm for intraprocedural verification of programs with static memory allocation that uses PDR to refine loop invariants.

The main parts of this effort is a generalization of the original Boolean PDR algorithm for the theory of quantifier-free formulae over bitvectors and a program verification frontend that uses this generalized PDR algorithm to infer loop invariants strong enough to settle the proof obligations of interest.

We summarize these two contributions in the next two sections, point out their strengths and weaknesses, and discuss directions for future work.

7.1 QF_BV Model Checking with PDR

In Chapter 3, we presented a QF_BV generalization of the PDR algorithm from [EMB11]. Though formulated for Boolean logic, the majority of the original algorithm can be used for QF_BV by substituting SAT formulae with SMT formulae and the SAT solver with an SMT solver for QF_BV. A major challenge has been to find a proper atomic reasoning unit (ARU) that allows to represent commonly encountered QF_BV invariants effectively but at the same time enables abstraction and fast reasoning as do Boolean cubes in the Boolean case. To this end, we discovered that a mix of polytopes and Boolean cubes performs satisfactory. While polytopes allow to represent piecewise linear invariants effectively, Boolean cubes are particularly useful for bitwise relations. For expansion of proof obligations, the

generalized PDR algorithm resorts to hybrid simulation where arithmetic operations are simulated using interval simulation and all other operations using ternary simulation. We have implemented the proposed algorithm and our experiments indicate that the model checker uses the meta information encoded in the QF_BV model checking instances and solves instances typically faster than the PDR algorithm included in ABC [BM10].

The generalized PDR algorithm appears to inherit most of the strengths and weaknesses of the original PDR algorithm. In addition to the convincing performance, the algorithm has modest memory requirements, is incremental, and can be parallelized.

In our experimentation, we have encountered two limitations of the algorithm. First, the algorithm occasionally suffers from a negative impact of the eager, simulation-based expansion of ARUs in the cover. To illustrate, consider the following slightly modified version of the simple example from Chapter 3.

$$\begin{aligned} I &:= (n \equiv 1) \wedge (x \equiv 0) \\ T &:= (n > 0) \wedge (x' \equiv x + 1) \wedge (n' \equiv n - 1) \\ B &:= (x \geq 1000) \end{aligned}$$

Figure 7.1 contains the state of the trace after several iterations of the main loop of PDR. In each frame f_i with $i < 999$, the following sequence happens: First, a proof obligation $x \geq 1000$ is generated. Next, the proof obligation is determined to be unreachable from the previous frame and after expansion, an integer cube $x > i + 1$ is added to the cover. Thereafter, no additional uncovered points in bad exist and a new frame is appended at the end of the trace.

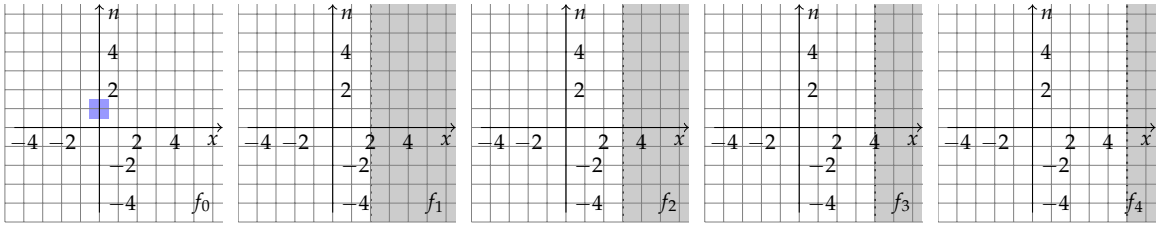


Figure 7.1: Illustration of Negative Impact of Eager Expansion

This implies that 1000 frames need to be added to the frame before a similar invariant as for the original example (see 3.2.1) is found. In our experiments, we have determined that the illustrated problem can be alleviated by providing more general proof obligations (e.g. the

counterexample generalization discussed in Chapter 4 has this effect) and by randomizing the expansion sequence. The root cause, however, is an inherited property of the PDR algorithm and critical to the performance of the algorithm for many benchmark problems. The illustrated problem appears to be more prevalent in QF_BV model checking problems than in Boolean ones, though.

Another performance problem we have encountered occurs in some benchmarks which require complex invariants over many variables to be solved. Consider e.g. a benchmark where the invariant $x_e - x_b \geq y_e - y_b$ is required. Such a loop invariant would indicate that x increases at least as much as y. To find this invariant, at least four applications of the reshape-operation are necessary. In practice, one could expect that many more futile attempts are made until the desired loop invariant is found. One strategy to alleviate the problem could be to add proxy variables standing for empirically frequent combinations of raw variables. For the example invariant above, the proxy variables $x_p = x_e - x_b$ and $y_p = y_e - y_b$ would be particularly helpful. A proper investigation of the potential of using proxy variables will be an interesting direction for future work.

7.2 Property Directed Program Verification

In the second part of this dissertation, we have devised a sound and complete algorithm for intraprocedural verification of programs with static memory allocation. The program verification algorithm is based on Property Directed Reachability in two ways: First, it uses the QF_BV PDR algorithm from the first part of this dissertation as backend model checker to infer loop invariants. Second, its design is based on PDR and incorporates several of its features. Among others, similar to PDR, PDPV divides the overall verification instances into smaller and more manageable subproblems, applies interpretation to generalize proof obligations, and refines the current model of the program only if necessary (*lazy*) and in a targeted manner (*property-directed*).

In addition to these properties, PDPV has the advantageous property that it uses a form of loop invariants to infer a combinational model of the program strong enough to prove the properties of interest. Loop invariants are a well defined and understood interface. As such, the algorithm can be combined with other approaches to program verification as we have demonstrated in Chapter 6. In contrast to other approaches to invariant inference (see Section 5.1.2), the PDR-based approach is principally able to derive

any invariant necessary to prove a program safe.

The main weakness of the presented approach lies in the fact that it is only suitable for intraprocedural verification. If the PUV is non-recursive, a straight-forward solution that allows to cope with this problem is *inlining*, i.e. by copying the body of functions into their call sites. However, this process can increase the size of the PUV substantially and renders the process of subsequent verification infeasible for all but the smallest programs.

A more practical solution to this end may be the use of *function summaries*, logic formulae that capture the meaning of the function body. We conjecture that the recursive refinement scheme as applied to loop invariants may also be adequate for inferring function summaries.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BB09] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools & Algorithms for the Construction & Analysis of Systems (TACAS'09)*, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[BBL08] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proceedings of the Joint Workshop of the 6th International Workshop on SMT & 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 33–38, New York, NY, USA, 2008. ACM.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.

[Bey12] Dirk Beyer. Competition on Software Verification. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, pages 504–524, Berlin, Heidelberg, 2012. Springer-Verlag.

[Bie06] Armin Biere. The AIGER And-Inverter Graph format. http://fmv.jku.at/aiger, 2006.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines and

Languages with Only two Formation Rules. *Comm. ACM*, 9(5):366–371, May 1966.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pages 358–372, Berlin, Heidelberg, 2007. Springer-Verlag.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 82–87, New York, NY, USA, 2005. ACM.

[BM08] Aaron R. Bradley and Zohar Manna. Property-Directed Incremental Invariant Generation. *Form. Asp. Comput.*, 20(4-5):379–405, June 2008.

[BM10] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40, 2010.

[BR13] John Backes and Marc Riedel. Using Cubes of Non-State Variables with Property Directed Reachability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, Washington, DC, USA, 2013. IEEE Computer Society.

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[Bra12] Aaron R. Bradley. Understanding IC3. In *Proceedings of the 15th International Conference Theory & Applications of Satisfiability Testing (SAT'12)*, pages 1–14, June 2012.

[Bur95] Dennis Burke. All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse, 1995.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[CG12] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *Proceedings of Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293, 2012.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[Cla] Clang. A C Language Frontend for LLVM. Technical Report http://clang.llvm.org.

[CM90] Richard H. Cobb and Harlan D. Mills. Engineering Software under Statistical Quality Control. *IEEE Software*, 7(6):44–54, November 1990.

[Cra57] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *The Journal of Symbolic Logic*, 22(3):pp. 250–268, 1957.

[CSS03] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Proceedings of Computer Aided Verification (CAV'03)*, pages 420–432. Springer Verlag, 2003.

[DF88] Edsger Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison Wesley, 1988.

[Dij68] Edsger W. Dijkstra. Go To Statement Considered Harmful. *Comm. ACM*, 11(3):147–148, 1968. letter to the Editor.

[Dij76] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dij90] Edsger W. Dijkstra. *Reasoning about Programs (videotape)*. University Video Communications, Stanford, CA, USA, 1990.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem Proving. *Comms. ACM*, 5:394–397, July 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Dow97] Mark Dowson. The Ariane 5 Software Failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, March 1997.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient Implementation of Property Directed Reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011.

[Fea91] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20:23–53, 1991. 10.1007/BF01407931.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proceedings of Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer-Verlag, July 2009.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference Theory & Appl, Satisfiability Testing (SAT)*, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[Hol01] Gerard J. Holzmann. Economics of Software Verification. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 80–89. ACM, 2001.

[ISGG05] Franco Ivančić, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model Checking C Programs Using F-SOFT. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[KJN12] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. SMT-based Induction Methods for Timed Systems. In *Proceedings of the 10th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'12, pages 171–187, Berlin, Heidelberg, 2012. Springer-Verlag.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 263–268, Anaheim, CA, June 1997.

[KMNP13] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, Inductive Coverability. In *Proceedings of Computer Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 158–173. Springer-Verlag, 2013.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LT93] N.G. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[McC04] Steve McConnell. *Code Complete (2nd Edition)*. Microsoft Press, USA, 2004.

[McM03] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Proceedings of Computer Aided Verification (CAV'03)*, volume 2742 of *Lecture Notes in Computer Science*, pages 1–13, 2003.

[McM04] K.L. McMillan. An Interpolating Theorem Prover. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS'10), volume 2988 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2004.

[McM06] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag, July 2006.

[McM10] Kenneth L. McMillan. Lazy Annotation for Program Testing and Verification. In *Proceedings of Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, July 2010.

[MKS09] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to Richer Logics. In *Proceedings of Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 462–476, 2009.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[SB11] Fabio Somenzi and Aaron R. Bradley. IC3: Where Monolithic and Incremental Meet. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 3–8, Austin, TX, 2011.

[SDDA11] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying Loop Invariant Generation using Splitter Predicates. In *Proceedings of Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 703–719. Springer-Verlag, July 2011.

[Tas02] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology (NIST), 2002.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.