Automated Incremental Software Verification

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Grigory Fedyukovich

under the supervision of
Prof. Natasha Sharygina

December 2015

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Grigory Fedyukovich
Lugano, 10 December 2015

Abstract

Software continuously evolves to meet rapidly changing human needs. Each evolved transformation of a program is expected to preserve important correctness and security properties. Aiming to assure program correctness after a change, formal verification techniques, such as Software Model Checking, have recently benefited from fully automated solutions based on symbolic reasoning and abstraction. However, the majority of the state-of-the-art model checkers are designed that each new software version has to be verified from scratch.

In this dissertation, we investigate the new Formal Incremental Verification (FIV) techniques that aim at making software analysis more efficient by reusing invested efforts between verification runs. In order to show that FIV can be built on the top of different verification techniques, we focus on three complementary approaches to automated formal verification.

First, we contribute the FIV technique for SAT-based Bounded Model Checking developed to verify programs with (possibly recursive) functions with respect to the set of pre-defined assertions. We present the function-summarization framework based on Craig interpolation that allows extracting and reusing over-approximations of the function behaviors. We introduce the algorithm to revalidate the summaries of one program locally in order to prevent re-verification of another program from scratch.

Second, we contribute the technique for simulation relation synthesis for loop-free programs that do not necessarily contain assertions. We introduce an SMT-based abstraction-refinement algorithm that proceeds by guessing a relation and checking whether it is a simulation relation. We present a novel algorithm for discovering simulations symbolically, by means of solving $\forall\exists$-formulas and extracting witnessing Skolem relations.

Third, we contribute the FIV technique for SMT-based Unbounded Model Checking developed to verify programs with (possibly nested) loops. We present an algorithm that automatically derives simulations between programs with different loop structures. The automatically synthesized simulation relation is then used to migrate the safe inductive invariants across the evolution boundaries.

Finally, we contribute the implementation and evaluation of all our algorithmic contributions, and confirm that the state-of-the-art model checking tools can successfully be extended by the FIV capabilities.

Contents

Chapter 1

Introduction

Software development is a continuous process that repeatedly iterates between the stages of implementing a program and checking its safety. To satisfy quality standards, a software product should pass through a myriad of intermediate verification stages, each of which assures safety of a particular change with respect to its baseline version. Motivated by the need to make the software analysis exhaustive and fully automated, the formal verification techniques, such as Software Model Checking [45], have recently benefited from efficient and sound solutions that were acknowledged with prestigious recognitions, such as the Turing award [1].

Without detracting from the merits of the state-of-the-art model checking analyzing individual program versions, there is a further demand for new methods to make other steps in the typical "verify-bugfix-verify" workflow automated and exhaustive. In particular, there is a clear need for new techniques that would make the software analysis more efficient by reusing invested efforts between verification runs. This aspiration gives rise to two research questions that we address in this dissertation. First, one should find a *reusable specification* of the already verified program version to be used while verifying another program version. Second, one should find a *relational specification* between program versions that describes how the versions relate to each other. These specifications are essential ingredients of the Formal Incremental Verification (FIV) approach and can be used for various tasks such as upgrade checking, incremental analysis of different properties of the same code, change impact calculation, etc.

Nowadays there exist various automated approaches to formal verification, the underlying methods of which are orthogonal to each other. The techniques may treat a given verification condition unequally and discover the incompatible verification certificates. The underlying conceptual differences decrease the chances for a possible FIV solution to support different model checkers. The intention of this dissertation is to develop innovative FIV approaches which would be generic and could be used complementary by different model checking tools.

Having its goal of enhancing various automated verification techniques by the FIV capabilities, this dissertation considers three complementary approaches to model checking with three application domains respectively. First, we develop a FIV technique for Bounded Model Checking designed to verify programs with (possibly recursive) functions. Second, we develop a technique for simulation relation synthesis for loop-free programs that do not require using pre-defined assertions. Finally, we develop a FIV technique for Unbounded Model Checking designed to verify programs with (possibly nested) loops. In the rest of this chapter, we will state the main contributions of this dissertation and prepare the reader to a more formal description in the following chapters.

1.1 Automated Formal Verification

Almost a half century has passed after the rise of program verification. The formal verification approach offered strict mathematical reasoning about programs and some desired properties about program variables in a particular program state. Apart from testing the program with respect to some particular inputs, formal verification allowed proving that properties hold in the program for any possible inputs. Thus the process of analyzing the constructed formalism has to be performed statically, i.e., without actual running the program, but dealing only with its source code. One of the first mentionings of such analysis appears in the seminal work on *Hoare logic* [78]. The idea is to construct so called Hoare triples by annotating a program with a pair of predicates, a pre- and a post-condition. To check correctness of the program means to deductively prove that if the precondition holds, and the program terminates, then the post-condition has to hold. Hoare logic inspired a large number of semi-automated formal techniques based on type inference and interactive theorem proving.

Model checking, an alternative formal verification approach to Hoare-style derivation, was proposed in the early eighties by [44; 116]. The main idea is to convert the actual program to a formal model by means of a transition system, a directed acyclic graph (DAG) with program states as nodes and transitions as edges. Each transition in the DAG is coupled with the state from which the transition starts (called a *pre-state*) and a state reachable after this transition (called a *post-state*). This formalism allows traversing the model and checking whether a bad state is reachable, and thus, can obviously be automatized. Another self-explanatory benefit of model checking is its ability to provide counter-examples witnessing the reachability of the bad state. Evidently, it is a path in the DAG starting from the initial state and leading to the bad state.

The drawback of the original approaches to model checking (nowadays called *explicit-state* model checking) is in its high computational burden (also referred to as *state-space explosion*). Moreover, program statements such as loops, nested function calls and recursion force the state space to grow rapidly, which makes explicit-state model checking inefficient (if not completely impossible). Until recently, classical model checkers were used mainly for verification of protocols, written in restrictive domain-specific languages [81; 8; 28].

To combat the state-space explosion, in the early nineties the focus in model checking turned to symbolic reasoning. To bridge the gap between a DAG representing the transition system and a Boolean function, the authors of [32] proposed to use Binary-Decision Diagrams (BDD) [30], a canonical form for Boolean expressions. With BDDs, as opposed to explicit representation, each state is encoded into a Boolean formula, and computing the post-states requires expensive quantifier elimination. Nowadays, due to the high cost, BDDs are not widely used, but the symbolic reasoning is performed by SAT, SMT or Horn solvers. The rise of various methods for symbolic reasoning made it easier for the model checkers to construct over-approximations and under-approximations of programs, and that in turn provided the research area with new revolutionary solutions.

The key insight behind over-approximations used in symbolic model checking is that they allow representing a larger set of program states in a more compact way. Whenever the unreachability of an error is happened to be proven in an over-approximated model, it immediately implies the error unreachability in the precise model. However, the use of over-approximation may cause emergence of spurious errors, requiring to refine the over-approximation and iteratively repeat the check. If this approach succeeds, it produces the *proofs* of program safety, namely safe inductive invariants. *Safe* means that the invariant defines an over-approximation of the set of reachable program states excluding those that violate the given assertion. *Inductive* means that the invariant covers all the initial program states, and any computation starting from a state described by the invariant can only reach states still represented by the invariant.

Nowadays both, creating over-approximation and refining it on demand, are done automatically. This paradigm to symbolic model checking, called *Counter-Example-Guided Abstraction Refinement* (CEGAR) [41], allows multiple ways for handling abstractions, including the Lazy Abstraction [76] and the Predicate Abstraction [68]. One of the most useful techniques for the refinement is *Craig interpolation* [48]. Given a pair of unsatisfiable formulas $A \wedge B$, a Craig interpolant I is a formula implied by A which is still unsatisfiable with B and expressed over the common alphabet of A and B. The refinement procedure is of great importance for the CEGAR-based model checkers, and the systematic failures to succeed in refinement often lead to diverging the model checker.

The most recent breakthrough in symbolic model checking is *Property Directed Reachability* (PDR/IC3) [26; 53]. Similarly to CEGAR, the goal of PDR can be formulated as searching for a safe inductive invariant. However, PDR is based on a backward search and starts with the set of bad states. It uses a SAT/SMT solver to iteratively extend the bad suffix towards the initial state until a pre-state for some transition and some suffix does not exist. Thus, if the bad suffix is extended to the initial state then a counter-example is found. Otherwise PDR requires a *generalization* procedure to learn a set of unreachable states that blocks the current suffix. The intersection of *all* such learned sets is in fact a safe inductive invariant. The generalization is analogous to the refinement in CEGAR, and some SMT-based extensions of PDR perform the generalization using Craig interpolation [80; 23].

Like classical BDD-based model checking, PDR requires symbolic computation of the sets of states that need to be processed in the next iteration. However the former does the good post-states, and the latter does the bad pre-states of some transitions. PDR earns some benefits from this fundamental difference, since in contrast to BDD-based model checking, it does not necessarily require a computed set of states to be complete. While blocking a strict subset of bad pre-states, PDR does not break soundness of the entire approach, but just considers a coarser over-approximation of the program. On the other hand, while exploring a strict subset of good post-states, a BDD-based model checker might miss some counter-examples, thus making the analysis incomplete. This observation is exploited by the state-of-the-art PDR-based model checkers that under-approximate pre-state computation, in particular, using a *Model-Based Projection* [86; 52].

Despite leading to incomplete results, symbolic model checking that under-approximates the set of good post-states is still practically important. Indeed, none of the BDD-, CEGAR- or PDR-based sound and complete model checkers are guaranteed to terminate while dealing with programs with infinite state space. In contrast, when the state space is bounded to contain all program states reachable within k transitions starting from the initial state then it can always be possible to find all counter-examples with the length at most k. This approach to symbolic model checking, called *Bounded Model Checking* (BMC) [21], also relies on the power of the state-of-the art SAT and SMT solvers and thus is widely used in academia and industry.

Unlike this rich line of research on automated formal verification of the individual programs, the area of incremental program verification is still a young and underdeveloped field. Existing automated and semi-automated techniques to conduct *equivalence checking* are based on Hoare Logic [87; 14], BDDs [110], BMC [43; 67; 92], CEGAR [35], symbolic execution [112; 137] and translation validation [109; 108] (to be discussed in Sect. 4.4). However, these approaches are not designed to efficiently cooperate with existing formal verification techniques, and thus, do not benefit from the analysis already performed for individual programs. In this dissertation, we focus on the equivalence checking problem from the model checking point of view, and propose several innovative solutions that make the FIV approach practical and general for reuse. The remaining of this chapter discusses the important challenges and our contributions.

1.2 Challenges and Contributions

In the previous section, we gave an overview of the techniques on the cutting edge of automated formal verification used for validation of individual programs. The overall contribution of this dissertation is a collection of new Formal Incremental Verification (FIV) techniques that allow migrating the verification certificates across sequences of program versions. The classification of the individual contributions with respect to the state-of-the-art verification technologies is depicted in Fig. 1.1. For each considered verification approach (i.e, Bounded Model Checking, Simulation Discovery, and Unbounded Model Checking), we conducted research in two directions (represented by the left and the right arrows). First, we manipulated an individual program version in order to obtain a reusable specification. Second, we manipulated a pair of program versions in or-

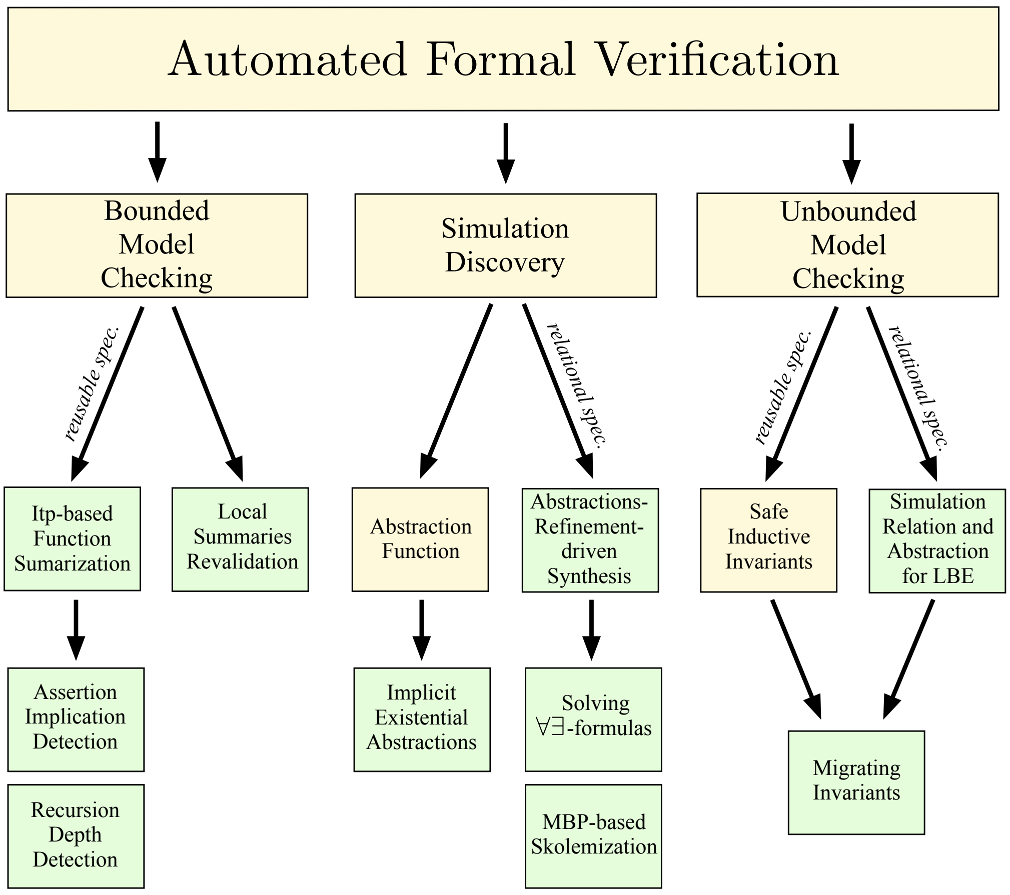

Automated Formal Verification
Bounded Model Checking
Simulation Discovery
Unbounded Model Checking
reusable spec.
relational spec.
reusable spec.
relational spec.
Itp-based Function Sumarization
Local Summaries Revalidation
Abstraction Function
Abstractions-Refinement-driven Synthesis
Safe Inductive Invariants
Simulation Relation and Abstraction for LBE
Assertion Implication Detection
Recursion Depth Detection
Implicit Existential Abstractions
Solving ∀∃-formulas
MBP-based Skolemization
Migrating Invariants

Figure 1.1. Main contributions of the dissertation (green boxes) built on the top of existing techniques (yellow boxes).

der to revalidate the reusable specification or establish a relational specification. In the following sections, we discuss the contributions in more detail.

1.2.1 SAT-Based Bounded Model Checking by means of Function Summarization

BMC is one of the most successful formal techniques in academia and industry. In a nutshell, a classical BMC tool proceeds in 3 main steps. First, it unwinds all loops and recursive function calls up to a given number of iterations and a given recursion depth respectively. This phase results in the unwound program represented by the so called Static Single Assignment (SSA) form in which all variables are assigned at most once. The SSA form is then conjoined with the negation of the assertion. Second, the constructed SSA form is encoded to a so called BMC formula and sent to the appropriate SAT or SMT solver. Finally, the rest of the job is done by the solver: if the formula is proven unsatisfiable then the program is safe up to the given bound; otherwise each model of the formula witnesses a counter-example.

The biggest challenge in BMC related to FIV is searching for a reusable specification. Unlike CEGAR- or PDR-based model checking, BMC is not typically driven by maintaining a safe inductive invariant. Thus, the synthesis of the reusable specification should be performed after the verification has terminated. We propose to exploit the fact that the bounded safety of the program is indicated by the unsatisfiability of the BMC formula. In the context of SAT-based BMC, we show that the proof of unsatisfiability can be further used to discover over-approximating function summaries that gather all important information of the function's behavior to prove the bounded safety. Finally, we present a novel technique based on Craig interpolation to achieve cheap and flexible function summarization. The details of this contribution are in Chap. 2, Sect. 2.2.

The next challenge in BMC related to FIV is searching for an algorithm that effectively reuses the summaries synthesized after the verification of one program version to verify another program version. We propose to revalidate the existing summaries locally in order to prevent re-verification of the entire code from scratch. If the check fails for some function call, it needs to be propagated by the call tree traversal to the caller of the function. If the check fails for the root of the call tree (i.e., the "main" function of the program), the whole program should be verified from scratch. On the other side, if for each function call there exists an ancestor function with the valid old summary then the new program version is safe up to the predefined bound. Finally, for functions whose old summaries are not valid any longer, the new summaries are synthesized using Craig interpolation. The details of this contribution are in Chap. 2, Sect. 2.3.

As a consequence, there is a straightforward challenge of constructing function summaries of a better quality in cases when the program is supplied with a sequence of pre-defined assertions. Based on the natural reliance of the summaries on the assertions, we observe that function summarization can be performed iteratively, i.e., checking each assertion at a time and refining the summaries if needed. Furthermore, if there exist some redundant assertions then some model checking tasks might be even skipped. Thus, for programs with multiple assertions, in the preprocessing step to BMC, we propose to perform a lightweight analysis of the set of assertions to optimize the generation of summaries. The details of this contribution are in Chap. 2, Sect. 2.2.7.

In addition, function summarization offers solutions for the other challenges in BMC that are not directly related to the FIV problem, namely speeding-up BMC for an individual program version and finding a proper loop and recursion bound for BMC. The bigger bound the more counter-examples can be found. Moreover, if the bound is sufficiently big, it indicates that no bugs are missed after the analysis has terminated. We propose an algorithm which uses function summaries for a fully automated detection of recursion depth. It proceeds in an iterative manner by refining the computed function summaries until the bound is detected or it becomes clear that the bound detection is infeasible. The details of this contribution are in Chap. 2, Sect. 2.2.3.

We refer the reader to Sect. 5.1-5.2 for the description of the tools implementing all the proposed algorithms. The new techniques and their evaluation have been published in the following papers: [126], [124], [125], [58], [59], [60], and [56].

1.2.2 SMT-Based Simulation Discovery

Simulation [104] is one of the oldest concepts in program analysis, introduced as early as Hoare logic. A simulation relation represents a condition under which the complete set of behaviors of one program (later referred to as *source* and denoted by S) is included into the set of behaviors of another program (later referred to as *target* and denoted by T). Simulation discovery is a useful procedure for the automated FIV since it does not require any assertions to be specified at the programs. If S is simulated by T then all assertions that hold in T will also hold in S. Thus, a discovered simulation provides a more precise verification certificate (namely, relational specification) than the function-summarization BMC approach from Sect. 1.2.1. When synthesized, this relational specification is an important ingredient for another class of FIV (to be discussed in Sect. 1.2.3), since it allows lifting the safe inductive invariants from T to S.

In realistic applications, there might be a sufficient semantic gap between S and T that essentially raises two challenges of finding an appropriate simulation relation between S and T: (1) the challenge of constructing a total simulation relation between two programs, and (2) whenever the target T does not simulate the source S, the challenge of finding an abstraction of the target T that simulates the source S.

We propose an SMT-based solution for these challenges. Our algorithm uses an abstraction-refinement reasoning which proceeds by guessing a relation and checking whether it is a simulation relation. We propose to reduce the problem of checking simulation to deciding validity of formulas of the form $\forall x \,.\, S(x) \implies \exists y \,.\, T(x, y)$. Intuitively, the formulas say "for each behavior of S there exists a corresponding behavior of T". We manipulate implicit abstractions of T by introducing existential quantifiers to the right-hand-side of the $\forall\exists$-formulas. We present a novel algorithm AE-VAL for deciding validity of $\forall\exists$-formulas that is based on efficient computations of the Model-Based Projections. In addition, AE-VAL extracts a Skolem relation to witness the existential quantifiers. This Skolem relation is the key to refine the considered abstractions of T, and therefore requires to be minimized and factored. As a solution to this challenge, we propose a technique to post-process the results of AE-VAL that results in a Skolem relation of the appropriate form.

We refer the reader to Chap. 3 for the detailed description of the contributions listed in this section, and to Sect. 5.3 for the description of the tools implementing all the proposed algorithms. These contributions have been published in the paper [57].

1.2.3 SMT-Based Unbounded Model Checking via Abstract Simulation

The CEGAR- or PDR-based approaches to Software Model Checking reduce the verification tasks to finding safe inductive invariants, as pointed out in Sect. 1.1. The safe inductive invariants play an important role of *proof certificates* and over-approximate all safe behaviors of the program. Therefore, these techniques are served to provide sound analysis of the programs with unbounded (and possibly nested) loops. To compactly represent such complex programs, model checkers use a "large-block" encoding (LBE) that collapses the control-flow graph (CFG) into the Cut-Point Graph (CPG). In CPG, the nodes represent heads of the loops (called cutpoints), and the edges represent the longest loop-free program fragments. Whenever a program is proven safe, the CPG is labeled by predicates, such that for each CPG-edge the labeling of the corresponding in- and out- nodes constitutes a valid Hoare triple.

We refer to the challenge of constructing a FIV technique for Unbounded

Model Checking as establishing a *Property Directed Equivalence* (PDE) between programs, i.e., to check whether the programs both satisfy the same property (and consequently, are happy with the same safe inductive invariant). Clearly, in contrast to BMC-based FIV (outlined in Sect. 1.2.1), PDE does not have a challenge for synthesizing a reusable specification, since the safe inductive invariants perfectly fit this goal. However, there still remains a challenge of migrating the existing invariant between two programs (the *already verified* one, and the *modified* another one). We propose a solution based on the concept of *Abstract Simulation* outlined in Sect. 1.2.2. We contribute an algorithm that performs an iterative abstract-refinement reasoning to automatically derive an abstraction of the already verified program that simulates the precise modified program .

One important feature of the contributed algorithm is that it guides the abstraction generation by the safe inductive invariant. If a simulation for such a *proof-based abstraction* is found then the proof can be migrated directly. Another distinguishing feature of the algorithm is the ability to migrate the invariants through abstractions even if the abstractions do not preserve safety. It attempts to lift as much information from the invariant as possible, and then strengthens it using a Horn-based unbounded model checker.

We refer the reader to Chap. 4 for the detailed description of the contributions listed in this section, and to Sect. 5.3 for the description of the tools implementing all the proposed algorithms. These contributions have been published in the following papers: [55], and [57].

Chapter 2

SAT-Based Bounded Model Checking by means of Function Summarization

Incremental verification by means of Bounded Model Checking (BMC) is one of the most widely-used bug-hunting techniques in academia and industry. It is served to trade-off completeness of the state-space exploration for finding as many counter-examples as allowed by the predefined time- and resource constraints. BMC problem can be reduced to a SAT problem and be efficiently solved using state-of-the-art decision procedures. We start this chapter with an overview of the recent advances in SAT solving (Sect. 2.1) that can be exploited in order to make BMC reusable.

In Sect. 2.2, we propose the algorithms to create and use function summaries by means of Craig interpolation, a well-known technique to obtain abstractions from the proof of unsatisfiability of a SAT formula. We propose solutions for several challenges occurring in this task, namely: detecting a sufficient depth for recursive function calls, refinement of summaries with respect to a sequence of assertions and exploiting the dependencies of assertions.

Finally, Sect. 2.3 proposes a solution to the FIV problem in the BMC setting. Our algorithm reuses the summaries synthesized after the verification of one program version to verify another program version. The main novelty of the approach is its efficient way to revalidate the already existing summaries locally in order to prevent re-verification of the entire code from scratch.

The results reported in this chapter have been published in the following papers: [126] (co-authored with Ondrej Sery and Natasha Sharygina), [124] (co-authored with Ondrej Sery and Natasha Sharygina), [125] (co-authored with

Ondrej Sery and Natasha Sharygina), [58] (co-authored with Ondrej Sery and Natasha Sharygina), [59] (co-authored with Ondrej Sery and Natasha Sharygina), [60] (co-authored with Natasha Sharygina), and [56] (co-authored with Andrea Callia D'Iddio, Antti Eero Johannes Hyvärinen and Natasha Sharygina). All the presented algorithms are implemented and undergone rigorous evaluation that is shown in the further chapter, and in particular, in Sect. 5.1 and Sect. 5.2.

2.1 Background

2.1.1 SAT Solving and Craig Interpolation

Given a finite set of propositional variables, a *literal* is a variable p or its negation $\neg p$. A *clause* is a finite set of literals and a formula ϕ in *conjunctive normal form* (CNF) is a set of clauses. We also refer to a clause as the disjunction of its literals and a CNF formula as the conjunction of its clauses. A variable p occurs in the clause C, denoted by the pair (p, C), if either $p \in C$ or $\neg p \in C$. The set $var(\phi)$ consists of the variables that occur in the clauses of ϕ. A *truth assignment* μ is a function that assigns a Boolean value ($\top$ or $\bot$) to each variable p. A truth assignment μ is a *satisfying assignment* (or a *model*, denoted $\mu \models \phi$) of a formula ϕ if ϕ evaluates to $\top$ under μ.

The propositional satisfiability problem (SAT) aims at determining whether there exists a model of a CNF formula ϕ. A formula ϕ *implies* a formula ϕ', denoted $\phi \implies \phi'$, if every model of ϕ is a model of ϕ'. If there is no model for a formula ϕ, ϕ is called *unsatisfiable*, denoted $\phi \implies \bot$.

For two clauses C^+, C^- such that $p \in C^+$, $\neg p \in C^-$, and for no other variable q both $q \in C^- \cup C^+$ and $\neg q \in C^- \cup C^+$, a *resolution step* is a triple C^+, C^-, $(C^+ \cup C^-) \setminus \{p, \neg p\}$. The first two clauses are called the *antecedents*, the latter is the *resolvent* and p is the *pivot* of the resolution step. A *resolution refutation* (also referred to as *proof of unsatisfiability*) R of an unsatisfiable formula ϕ is a directed acyclic graph where the nodes are clauses and the edges are directed from the antecedents to the resolvent. The nodes of a refutation R with no incoming edge are the clauses of ϕ, and the rest of the clauses are resolvents derived with a resolution step. The unique node with no outgoing edges is the empty clause $\bot$.

Definition 1 (**cf.** [48]). *Given formulas A and B such that $A \wedge B \implies \bot$,* Craig interpolant *of* (A, B) *is a formula* I *such that* $A \implies I$*,* $I \wedge B \implies \bot$*, and* $var(I) \subseteq var(A) \cap var(B)$*.*

For a pair of inconsistent formulas (A, B), an interpolant always exists [115]. Furthermore, interpolation can be efficiently handled over the resolution refutation witnessing $A \wedge B \implies \bot$. As shown in [98], an interpolant formula can be synthesized by traversing the resolution refutation bottom-up and analyzing whether the pivot variables are contained only in A or also in B. Thus, the size of interpolants is linear in the number of nodes of the resolution refutation (but the size of the resolution refutation can be exponential in the size of variables of A and B).

A commonly used framework for computing the interpolant from a given resolution refutation is the *labeled interpolation system* (LIS) [51], a generalization of several interpolation algorithms (including the widely used algorithms *McM* (McMillan [98]), *Pud* (Pudlák [115]), and *McP* ([51], dual to *McM*) parameterized by a *labeling function*. Given a labeling function and a refutation, LIS uniquely determines the interpolant. LIS induces a partial order over the labeling functions which relates the corresponding interpolants by strength. That is, the collection of the interpolating algorithms represents a complete lattice, where *McM* is the greatest element (i.e., it generates the strongest interpolant), *McP* is the least (i.e., it generates the weakest interpolant) and *Pud* is in between. Furthermore, various adjustments of the labeling function and the resolution refutation can be made in order to produce smaller interpolants [119; 83; 7].

Classical interpolation can be generalized so the partitions of an unsatisfiable formula naturally correspond to a tree structure. Let $\Phi = \bigwedge_{i=1}^{n} \phi_i$ and $K \subseteq \{0 \ldots n\}$, then a partitioning of Φ with respect to K allows representing Φ in the following form: $\Phi \equiv \Phi_K \wedge \overline{\Phi_K}$, where $\Phi_K = \bigwedge_{k \in K} \phi_k$ and $\overline{\Phi_K} = \bigwedge_{\ell \notin K} \phi_\ell$. An *interpolation system* is a function that, given Φ and K, returns an interpolant $I_{\Phi,K}$, a formula implied by Φ_K, inconsistent with $\overline{\Phi_K}$ and defined over the common language of Φ_K and $\overline{\Phi_K}$.

Now consider a tree $T = (V, E)$ with n nodes $V = \{0, \ldots, n\}$ imposed on Φ. Then, the subsets K_i of V can be identified with respect to the tree structure: $K_i = \{j \mid i \sqsubseteq j\}$, where $i \sqsubseteq j$ denotes that j is a descendant node of i. Formally, a sequence of n interpolation systems has the T*-tree interpolation* property [72] iff for any node i and all its descendant nodes j, $\bigwedge_{(i,j) \in E} I_{\Phi,K_j} \wedge \phi_i \implies I_{\Phi,K_i}$. Notice that for the *root* node in T, $K_{root} = V$ and $I_{\Phi,V} \implies \bot$.

Rollini et al. [120; 70] proved that the interpolants generated from the same resolution refutation using the algorithm *Pud* have the tree-interpolation property. Furthermore, this property propagates to other algorithms (e.g., *McM*) which are stronger than *Pud*.

2.1.2 Interpolants in Model Checking

Craig interpolation is commonly used as a means of abstraction in various automated formal methods [98; 74; 99; 33; 130; 72; 100; 88; 4; 102; 5; 34; 131; 101; 132; 123; 2]. While it was originally exploited in a purely propositional context (e.g., as formalized in the previous Sect. 2.1.1), it is the rapid rise of state-of-the-art model checkers that demanded interpolants to be efficiently created and manipulated within first-order logic. However, in this section we give a brief historical overview of the interpolation-based formal methods regardless of the background theory and a particular algorithm with which interpolants were created.

The first application of interpolation to formal verification was pioneered by McMillan [98] in his complete technique for finite-state model checking (e.g., for hardware designs). This BMC-based approach unrolls the entire program up to first k steps and encodes this unrolling to a SAT formula to be further checked for satisfiability. If the formula is unsatisfiable, proving that no counter-example of length k exists, interpolation is used to over-approximate the set of states reachable within k steps. Thus, the synthesized interpolant can be treated as a safe invariant. If, in addition, this invariant is also inductive then the program is correct. Otherwise, the model checker increments k and iterates. Nowadays, this method is considered highly influential to model checking in general. It inspired a number of followers who proposed several ways of its improvement [33; 130; 34].

In the later work [99], McMillan turned this approach into the abstraction-refinement fashion [41], thus allowing to verify infinite-state systems (e.g., programs with loops). As in [98], interpolants are served to over-approximate the sets of reachable states, but now interpolation is applied on demand and only to individual program paths. Therefore, no unrolling of the entire program is needed, and some paths may remain abstracted away as long as it does not prevent proving safety. This idea of *lazy abstraction with interpolants* (LAWI) also attracted the great attention in the model checking community and gave rise to a number of extensions, e.g., to handle bit-vector operations [88], function calls [100], recursion [72; 4], arrays [6], numerical [5] and heap-manipulating programs [2].

An alternative way of applying interpolants to the CEGAR loop was proposed in the scope of predicate abstraction [74]. It helped deriving new predicates to make an abstraction more precise, while the creation of the initial abstraction was performed without interpolation. In particular, sets of predicates are extracted from interpolants of the formulas corresponding to a prefix and suffix of a spurious error trace. This results in predicates associated with program locations along the spurious error trace yielding a more fine-grained abstraction. The authors also propose reordering of a path formula to generate interpolants with local variables suitable for inter-procedural analysis. This method of refinement is implemented in the software model checker BLAST [76] and later in CPACHECKER [19].

Recently interpolants were applied to IC3/PDR [80; 23] to leverage the generalization procedure. Similarly to the refinement in CEGAR-based techniques, whenever a spurious counter-example is found, interpolation is used to refine the sets of reachable states. The orthogonal work [38] proposes to combine PDR with predicate abstraction, in which the refinement is also performed using interpolation. Finally, a new technique AVY [131; 132] proposes to interleave the forward BMC-style global reasoning and the backward PDR-style local reasoning. As in [98], AVY interpolates the unfeasible fixed-length unrolling of the entire program to construct a candidate invariant which is important for two purposes. First, if successfully generalized to become inductive, it could indicate that the program is correct. Second, it could be conjoined with the encoding of a longer program unrolling in the next steps of the algorithm, and thus, accelerate the SAT-solving phases of the technique and its entire convergency.

Notably, model checking is not the only formal approach in which interpolants are used (and describe the sets of good states). Simultaneously they can be produced to describe also the sets of bad states [123]. This application is particularly useful in software debugging when a certain error trace needs to be analyzed. Interpolants are generated over the unfeasible symbolic encoding of such a trace providing so called *error invariants*. The computed error invariants explain the error and shrink the search space of the program statements to be potentially fixed.

In this dissertation, we contribute yet another application of interpolants: to over-approximate behaviours of the program functions and reuse them for checking safety of sequences of software versions. We construct interpolants from a resolution refutation generated by a SAT solver for an unsatisfiable BMC formula. Such BMC formula encodes unreachability of an error in one version of a given program unwound up to a given bound. The key insight in our contribution is to delegate checking whether the error is unreachable in another version of the program to checking whether the previously constructed interpolants still over-approximate the new program behaviors unwound up to the same bound. Details of our advanced interpolation-based techniques can be found in Sect. 2.2 and 2.3, while the basic BMC encoding is detailed in the rest of this section.

2.1.3 Software Bounded Model Checking and Symbolic Execution

Bounded Model Checking (BMC) [22] is aimed at searching for errors in a program within the given bound of loop iterations and recursion depth. Thus, BMC can only show existence of counter-examples, but it is not suitable for complete verification. However, as shown in the previous Sect. 2.1.2, it is a subroutine in several complete interpolation-based model checking algorithms. Furthermore, due to its performance, scalability and ease of use, BMC without a doubt is one of the most successful verification approaches in safety analysis of software and has been implemented in a dozen of tools including CBMC [40], ESBMC [47], SATURN [136], VERISOFT [82], CALYSTO [9], LLBMC [103] and LAV [133].

In this section, we provide conceptual and engineering details of Software SAT-based BMC. Given the unwinding bound ν, BMC unrolls the loops and recursion up to ν, encodes the program symbolically and delegates the checking to a SAT solver. If the number is not provided a priori, BMC may go into an infinite loop and not terminate. Typically in the absence of the bound number or when the number is set too high, a predefined timeout is used to cope with this problem.

BMC encodes the program into the Static Single Assignment (SSA) [49] form, in which every variable is assigned exactly once. Instead of program variables, the SSA form operates by *versioned* logical variables that are distinguished from the original program variables by the subscripts. In addition, each return value of nondeterministically treated functions is also encoded into a separate subscripted logical variable. Such a representation allows to track changes of the value of

```
f(int a) {
  if (a < 10)
    return f(a + 1);
  return a - 10;
}

main() {
  int y = 1;
  int x = nondet();

  if (x > 5)
    y = f(x);

  assert(y >= 0);
}
```

(a) C code

```
y0 = 1;
x0 = nondet0;
if (x0 > 5) {
  a0 = x0;
  if (a0 < 10)
    ...
    ret0 = ...;
  else
    ret1 = a0 - 10;
  ret2 = phi(ret0,
      ret1, a0 < 10);
  y1 = ret2;
}
y2 = phi(y0, y1,
      x0 > 5);
assert(y2 >= 0);
```

(b) SSA form

$$\begin{aligned}
&y_0 = 1 \land \\
&x_0 = nondet_0 \land \\
&a_0 = x_0 \land \\
&\boldsymbol{ret_0 = \ldots \land} \\
&\boldsymbol{\ldots \land} \\
&\boldsymbol{ret_1 = a_0 - 10 \land} \\
&\boldsymbol{(x_0 > 5 \land a_0 < 10 \implies} \\
&\qquad \boldsymbol{ret_2 = ret_0) \land} \\
&\boldsymbol{(x_0 > 5 \land a_0 \geq 10 \implies} \\
&\qquad \boldsymbol{ret_2 = ret_1) \land} \\
&\boldsymbol{y_1 = ret_2 \land} \\
&(x_0 > 5 \implies y_2 = y_1) \land \\
&(x_0 \leq 5 \implies y_2 = y_0) \land \\
&y_2 < 0
\end{aligned}$$

(c) BMC formula

Figure 2.1. BMC formula generation

each variable during program execution. Whenever a variable is assigned, the correspondent subscript is incremented, and a new logical variable is introduced.

Given an unwinding bound ν, a `while` loop is transformed into a chain of ν nested `if` conditionals in order to represent at most ν iterations of the loop. To encode values which depend on `if` branches and `while` loops we use the `phi`-function, which returns its first or second argument depending on the truth value of its third argument as follows:

$$\texttt{phi}(e_1, e_2, e_3) = \begin{cases} e_1 \text{ if } e_3 \text{ is true;} \\ e_2 \text{ otherwise.} \end{cases}$$

Example 1. *Fig. 2.1(a)-2.1(b) show an example of the SSA encoding of a simple C program. The program consists of two functions* `main` *and* `f` *in addition to a nondeterministically treated function* `nondet`*. Notably,* `f` *is recursive and is unwound 5 times. During unwinding, a call of the function is substituted by its body. Thus, in the example there will be five such nested substitutions, and the sixth call is simply skipped.*

Finally, the SSA form is encoded into a logical formula, called a *BMC formula* (we refer to Fig. 2.1(c) as an example) represented by a conjunction of clauses in first-order Logic. Each clause corresponds to a separate SSA step. The last conjunct in the BMC formula is negation of the assertion condition guarded by its path condition. Then the BMC formula is bit-blasted and checked for satisfiability using an off-the-shelf SAT solver. If a satisfying assignment is detected, it identifies an error trace. Otherwise, the program is safe up to the predefined unwinding bound. Notably, this unwinding bound may not be sufficient for complete verification. A program can be proven safe for the unwinding bound ν, but buggy for the unwinding bound $\nu + 1$.

In order to check completeness of the current unwinding, BMC tools automatically derive and implant so called *unwinding assertions*. Unwinding assertions are located after an unwound loop (or a recursive function call) and contain the negation of the loop (or call) condition as the argument. If all of the unwinding assertions hold then the corresponding unwinding bound is enough to guarantee completeness of the verification.

Classical BMC algorithms use a monolithic BMC formula (e.g., [40; 103; 82]). However, for advanced BMC-based algorithms that are based on Craig interpolation, it is convenient to use a so called *Partitioned* BMC formula, which is presented in Sect. 2.2.

2.2 Bounded Model Checking by means of Function Summarization

This section proposes a new approach to BMC which is aimed not only to verify the given program, but also to extract function summaries by means of Craig Interpolation. By *function summary* we mean an abstraction of the function behavior defined as a relation over its input and output variables, which over-approximates the precise behavior of the given function.

In contrast to classic BMC, we propose encoding the program to a so called *partitioned* BMC (PBMC) representation where each partition represents the body of a function called in a specific context. If the PBMC formula is unsatisfiable then the program is correct. Using the proof of unsatisfiability, Craig interpolation can be iteratively applied to generate an interpolant for each partition in the PBMC formula. By construction, such interpolants are the summaries of all the function calls in the program. If the PBMC formula is satisfiable then the program is buggy and a satisfying assignment witnesses an error trace reported back to the user.

We proceed by presenting how the constructed function summaries can be gradually while checking a sequence of assertions. First, we define the iterative *summarization-refinement* procedure to deal with function summaries. Second we propose a technique to decrease the number of possible refinements before the actual verification of a program with respect to a sequence of assertions.

2.2.1 Handling (Recursive) Functions

Definition 2. *An* unwound program *for a depth ν is a tuple $P_\nu = (\hat{F}_\nu, \hat{f}_{main}, child)$, s.t. $\hat{F}_\nu$ is a finite set of function calls, unwound up to the depth ν, $\hat{f}_{main} \in \hat{F}_\nu$ is a program entry point and $child \subseteq \hat{F} \times \hat{F}$ relates each function call $\hat{f}$ to all function calls invoked directly from it.*

There is a fixed set F to represent functions declared in the program and a possibly unbounded set $\hat{F}$ to represent function calls. $\hat{F}$ together with the relation *child* can be represented by a corresponding call tree with a root marked by $\hat{f}_{main}$. We also use relation *subtree* $\subseteq \hat{F} \times \hat{F}$, a reflexive transitive closure of *child*.

A subset $\hat{F}_\nu \subseteq \hat{F}$ collects all function calls that are located on the depth ν from the root of the cal-tree $\hat{f}_{main}$ or closer. A call $\hat{f} \in \hat{F}$ corresponds to a call of a target function, determined by a mapping *target* : $\hat{F} \rightarrow F$. There is exactly one call of function f_{main}, but there may be several calls of the other functions. For simplicity, later we will use primes (i.e., $\hat{f}'$, $\hat{f}''$,..) and subscripts (i.e., $\hat{f}_1$, $\hat{f}_2$,..) to differentiate the calls of the same function $f \in F$ in the unwound program.

Definition 3. *A function f is* recursive *if for every call $\hat{f}_i$, there is another call $\hat{f}_i'$ in its subtree, and target($\hat{f}_i$) = target($\hat{f}_i'$) = f.*

A recursive function f is said to be unwound ν times if there is a chain of function calls $\{\hat{f}_i\}$ (i.e., sequence $\hat{f}_0$, $\hat{f}_1$,.. $\hat{f}_\nu$), $1 \leq i \leq \nu$, *target*($\hat{f}_i$) = f, and each $\hat{f}_{i+1}$ is in the subtree of $\hat{f}_i$.

Example 2. *Fig. 2.2a shows an example with a single recursive function f called two times, once from function g and once from function f_{main}. In this example, the call tree contains two chains of calls of function f: the first one consisting of one function call $\{\hat{f}_2\}$, the other consisting of ν calls: $\{\hat{f}_1, \hat{f}_2', ..\hat{f}_\nu\}$, where the numbers* 1 *and ν are recursion depths.*

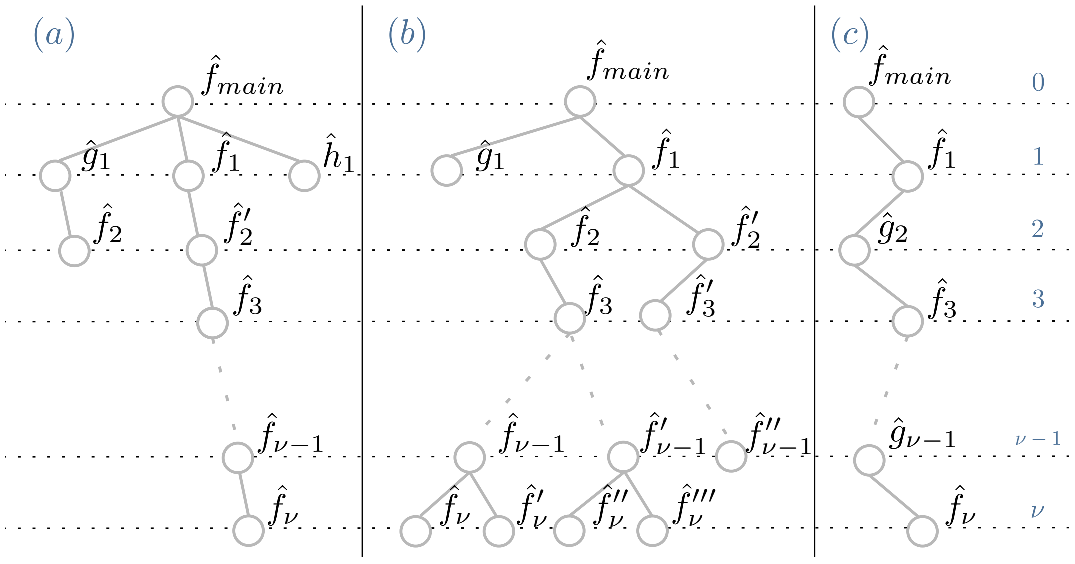


Figure 2.2. A program call tree with recursive functions unwound at most ν times: a) single recursion; b) multiple recursion; c) indirect recursion

Fig. 2.2b shows an example with a recursive function f called multiple times from itself (in the example, it is called 2 times). There are many chains of function calls possible for such scenario, and every one consists of at most ν calls of f, as demonstrated by a sample unwinding in the example. Notably, their unwinding depths can be different (and our algorithm will be able to detect the longest ones and stop exploring the chains for which the smaller depth is sufficient for verification).

Fig. 2.2c shows an example with indirect recursive functions f and g, such that each function is called not by itself, but by another function that it called. In the example, both f an g are unwound at most $\lfloor \frac{\nu}{2} \rfloor$ times (i.e., ν times altogether).

2.2.2 PBMC Encoding

The formula in classical BMC is generally constructed in a monolithical manner. As we discussed in Sect. 2.1.3, all the function calls are inlined, and the variables from the calling context tend to leak into the formulas of the called function as a part of the path condition. For example in Fig. 2.1(c), the variable x_0, which is local in the function `main`, appears in the encoding of the body of the function `f`.

To achieve the desired form, we generate the parts of the formula corresponding to the individual functions in separation and bind them together using a boolean variable $callstart_{\hat{f}}$ for each function call $\hat{f}$. Intuitively, $callstart_{\hat{f}}$ evalu-

$$
\begin{aligned}
& y_0 = 1 \land \\
& x_0 = nondet_0 \land \\
& a_0 = x_0 \land \\
& x_0 > 5 \Leftrightarrow callstart_{\hat{f}_1} \land \\
& y_1 = ret_0 \land \\
& (x_0 > 5 \Rightarrow y_2 = y_1) \land \\
& (x_0 \le 5 \Rightarrow y_2 = y_0)
\end{aligned}
$$

(a) formula $\phi_{\hat{f}_{main}}$

$$y_2 \ge 0 \Leftrightarrow \pi$$

(b) assertion formula π

$$
\begin{aligned}
& (a_0 < 10 \Leftrightarrow callstart_{\hat{f}_2}) \land \\
& a_1 = a_0 + 1 \land \\
& ret_1 = ret_3 \land \\
& ret_2 = a_0 - 10 \land \\
& (callstart_{\hat{f}_1} \land a_0 < 10 \Rightarrow \\
& \qquad ret_0 = ret_1) \land \\
& (callstart_{\hat{f}_1} \land a_0 \ge 10 \Rightarrow \\
& \qquad ret_0 = ret_2)
\end{aligned}
$$

(c) formula $\phi_{\hat{f}_1}$

Figure 2.3. PBMC formula generation

ates to $\top$ iff the corresponding function call $\hat{f}$ is reached. Another special variable is π, which encodes unreachability of an error in the entire program. We call the resulting formula a *partitioned bounded model checking* (PBMC) formula.

Example 3. *Fig. 2.3 demonstrates creation of a PBMC formula equivalent to the BMC formula from Ex. 1. In the example program, unwound 5 times, the partitions for function calls* `f`$_1$, `f`$_2$,.. `f`$_5$ *and* `main` *are generated separately. Note that the assertion π is not encoded inside $\phi_{\hat{f}_{main}}$, as in classical BMC, but separated from the rest of the formula, such that it helps interpolation.*

Formula $\phi_{\hat{f}_1}$ that encodes the function call `f`$_1$ *aims to symbolically represent the function output argument ret_0 by means of the function input argument a_0, symbolically evaluated in $\phi_{\hat{f}_{main}}$. At the same time, $\phi_{\hat{f}_1}$ relies on the value of ret_3 defined in $\phi_{\hat{f}_2}$ by means of a_1. Similar reasoning is applied to create each of the following partitions: $\phi_{\hat{f}_2}$,.. $\phi_{\hat{f}_5}$.*

2.2.3 Bounded Model Checking with Automated Detection of Recursion Depth

In reality BMC tools often rely on unwinding bounds that are not proven to be complete. That is, model checkers need an extra sufficiency check for this bound. In this section, we present a simple abstraction-refinement algorithm that employs PBMC encoding to iteratively find (if possible) such bound. While focusing

Algorithm 1: BMC with automatic detection of recursion depth

```
Input: Initial recursion depth: $\nu$; program unwound $\nu$ times:
       $P_\nu = (\hat{F}_\nu, \hat{f}_{main}, child)$; assertion to be checked: $\pi$
Output: Verification result: {SAFE, BUGGY, TIMEOUT}; detected recursion
        depth: $\nu$; error trace: $\epsilon$
Data: PBMC formula: $\phi$; set of precisely encoded function calls: $\mathbb{P}$; set of
      refinement candidates $\mathbb{R}$
1  $\mathbb{P} \leftarrow \hat{F}_\nu$;
2  while ($\neg$TIMEOUT) do
3      $\mathbb{R} \leftarrow \{\hat{g} \notin \mathbb{P} \mid child(\hat{f}, \hat{g}), \hat{f} \in \mathbb{P}\}$;    ▷ get refinement candidates
4      $\phi \leftarrow \neg\pi \wedge \bigwedge_{\hat{f}\in\mathbb{P}}$ CREATEFORMULA($\hat{f}$) $\wedge \bigwedge_{\hat{g}\in\mathbb{R}}$ NONDET($\hat{g}$);
5      $result, sat\_assignment \leftarrow$ SOLVE($\phi$);    ▷ SAT-solve the PBMC formula
6      if ($result$ = UNSAT) then
7          return SAFE, $\nu, \emptyset$;
8      else
9          $\epsilon \leftarrow$ EXTRACTCE($sat\_assignment$);    ▷ extract an error trace
10         $\mathbb{R} \leftarrow \mathbb{R} \cap$ EXTRACTCALLS($\epsilon$);    ▷ filter out calls which do not affect the satisfiability
11         if ($\mathbb{R} = \emptyset$) then
12             return BUGGY, $\nu$, $\epsilon$;
13         else
14             $\mathbb{P} \leftarrow \mathbb{P} \cup \mathbb{R}$;    ▷ unwind the call tree on demand
15             $\nu \leftarrow$ MAXCHAINLENGTH($\mathbb{P}$);    ▷ update the depth
16 end
17 return TIMEOUT;
```

on recursive programs, we assume they are loop-free (e.g., all loops are converted to recursion in the pre-processing step of the algorithm).

An overview of the algorithm is depicted in Alg. 1. The algorithm starts with a preset recursion depth ν (for simplicity, $\nu = 1$) and iterates until it detects the actual recursion depth, needed for complete proof of the program correctness, or a predefined timeout is reached. Notably, in each iteration of the algorithm, ν gets updated and is equal to the length of the longest unwinding chain of recursive function calls. In the end of the algorithm, all recursive calls are unwound exactly same number of times as they would be called during the execution of the program.

The details of the computation are given below. First, the algorithm aims at constructing a PBMC formula ϕ using the sets $\mathbb{P}$ and $\mathbb{R}$. Every function call $\hat{f} \in \mathbb{P}$

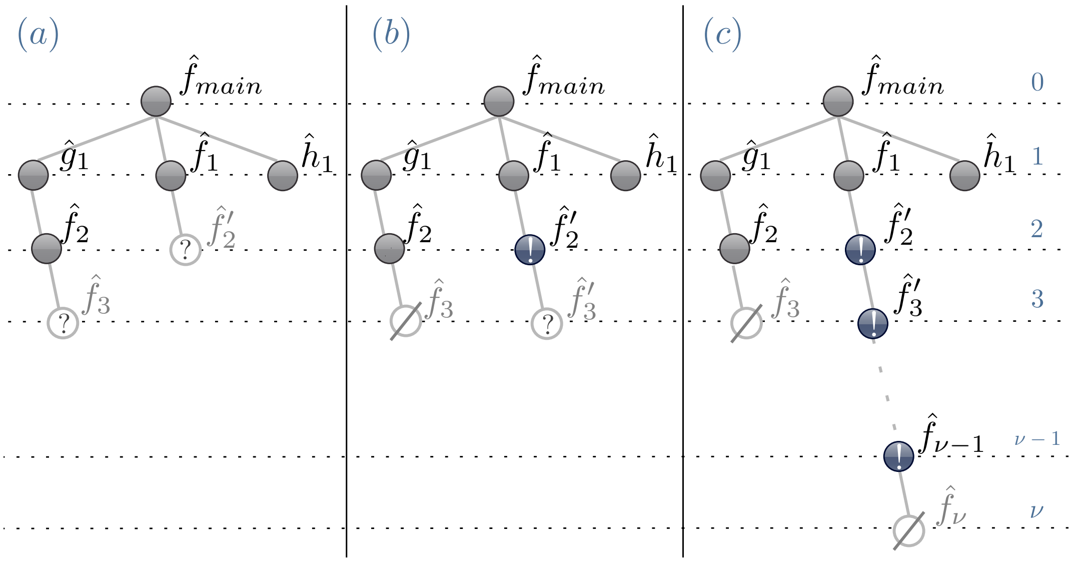


Figure 2.4. Illustration of the individual steps of the Alg. 1 on the example with a single recursive function f, called twice.

is encoded precisely, every function call $\hat{g} \in \mathbb{R}$ is treated nondeterministically. In particular, bodies of function calls from set $\hat{f} \in \mathbb{P}$ are encoded into the SSA forms (i.e., method CREATEFORMULA) and put together into separate partitions (one partition per each function call) of ϕ (line 4). At the same time, all function calls from $\mathbb{R}$ are replaced by empty formulas $\top$ (i.e., method NONDET). In total, ϕ encodes a program abstraction containing precise and over-approximated parts, conjoined by negation of an assertion π (line 4). In each iteration of Alg. 1, $\mathbb{P}$ represents a subset of the full unrolling of the program created with respect to a current value of ν (i.e., $\mathbb{P} \subseteq \hat{F}_\nu$).

After the PBMC formula ϕ is constructed, the algorithm passes it to a SAT solver. If ϕ is satisfiable, and the SAT solver returns a satisfying assignment (line 8), function calls from $\mathbb{R}$ are considered as candidate calls to be refined. To refine, the satisfying assignment is used to restrict $\mathbb{R}$ on the calls, appeared along the error trace ϵ (i.e., in the satisfying assignment) (line 10). In the next iteration of the algorithm, the calls from $\mathbb{R}$ are encoded precisely in the updated PBMC formula. Technically, the algorithm extends $\hat{F}_\nu$ by adding function calls from $\mathbb{R}$ (line 14), as shown, for example, on Fig. 2.4b. There, $\hat{f}'_2$ appears along ϵ and therefore it has to be refined; $\hat{f}_3$ does not appear in ϵ, so it will be encoded nondeterministically. If $\mathbb{R} = \varnothing$ then no nondeterministically treated recursive calls were found along the error trace, so the real bug is found (line 12), and the algorithm terminates.

If the SAT solver proves unsatisfiability of ϕ then the program abstraction, and consequently the program itself, are safe (line 7). This case is represented on Fig. 2.4c. The final recursion depth ν is detected, and the algorithm terminates.

Example 4. *Fig. 2.4 demonstrates a call tree of a program with a single recursive function called twice in different iterations of Alg. 1. In the example,* $\mathbb{P} = \{\hat{f}_{main}, \hat{g}_1, \hat{h}_1, \hat{f}_1, \hat{f}_2\}$ *(grey nodes) are encoded precisely, and* $\mathbb{R} = \{\hat{f}_3, \hat{f}'_2\}$ *(white nodes) are treated nondeterministically.*

Iteration 1: $\mathbb{P} = \{\hat{f}_{main}, \hat{g}_1, \hat{h}_1, \hat{f}_1, \hat{f}_2\}$ *(grey nodes) are encoded precisely,* $\mathbb{R} = \{\hat{f}_3, \hat{f}'_2\}$ *(white "?" nodes) are treated nondeterministically; the initial recursion depth is equal to 1.*

Iteration 2: solver returns SAT *(corresponding to the error trace* $\epsilon = \{\hat{f}_{main}, \hat{f}_1, \hat{f}'_2\}$*), set* $\mathbb{R}$ *is updated to contain only one function call* $\{\hat{f}'_2\}$ *(black "!" nodes). All calls from* $\mathbb{R}$ *are added to current* $\hat{F}_\nu$*. The current recursion depth is incremented, and equal to 2.*

Iteration ν*: solver returns* UNSAT *or* $\mathbb{R} = \varnothing$*, the detected recursion depth is equal to* $\nu - 1$.

Theorem 1. *Given the program P and an assertion* π*, if Alg. 1 terminates with an answer* SAFE *(*BUGGY*) then* π *holds (does not hold) for P.*

Proof. In case if ϕ is unsatisfiable, the formula ϕ represents some abstraction of P which contains precise and over-approximated components (as described in Sect. 2.2.3). Thus, any further strengthening of ϕ (e.g., by conjoining with new partitions that precisely encode function calls, treated nondeterministically in ϕ) is also unsatisfiable that implies the assertion π holds.

In case if ϕ is satisfiable, there exists a satisfying assignment representing an error trace. At the same time, the algorithm did not detect any nondeterministically treated recursive function calls along the error trace (line 11). It means that π is indeed violated within the current recursion depth. □

The algorithm is guaranteed to terminate within a given timeout when it finds an error or proves that the assertion holds. Similar to classical BMC, Alg. 1 terminates if the recursion depth is sufficient to disprove the assertion. Classical

BMC can prove the assertion up to some fixed recursion depth, but the result might be incomplete if the recursion depth is insufficient. In contrast, by Th. 1, if our algorithm does not yield a timeout, it guarantees that the detected recursion depth is complete to prove the assertion. The other benefit of our algorithm is that it does not require the recursion depth to be given a priori, but instead it is detected automatically.

Based on our observations, termination of Alg. 1 depends on the termination of the recursive program it was applied to. For example, the program with one single recursive function from Fig. 2.1(a) terminates for any values of input data. The recursion termination condition, `¬(a < 10)` defines the upper bound 10 for the value of `a`, and at the same time the function `f` monotonically increments the value of `a`. Hence, the recursive function `f` is called a fixed number of times and the program eventually terminates. Clearly, for complete analysis of this program it is enough to consider the maximum possible number of recursive function calls for every initial value of `a` which in this example is equal to 5. At the same time, it introduces an upper bound for the size of the constructed PBMC formula which is a sufficient condition to the SAT solver to terminate while solving it.

The underlying idea of Alg. 1, i.e., abstracting away some function calls and refining them on demand, also appeared in other model checking approaches proposed, e.g., in [9; 93]. However, in contrast to its competitors, our Alg. 1 can be enhanced to support verification of different assertions by means of constructing and reusing function summaries, on whose we elaborate in the remaining sections.

Alg. 1 is implemented and evaluated within a tool FunFrog. We discuss it in details in Sect. 5.1.1

2.2.4 Function Summaries

A function summary relates input and output arguments of a function. Therefore, a notion of arguments of a function is necessary. For this purpose, we expect to have a set of program variables $\mathbb{V}$ and a domain function $\mathbb{D}$ which assigns a domain (i.e., set of possible values) to every variable from $\mathbb{V}$.

Definition 4. *For a function f, sequences of variables $args^f_{in} = \langle in_1, \ldots, in_m \rangle$ and $args^f_{out} = \langle out_1, \ldots, out_n \rangle$ denote the input and output arguments of f, where $in_i, out_j \in \mathbb{V}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. In addition, $args^f = \langle in_1, \ldots, in_m, out_1, \ldots, out_n \rangle$*

denotes all the arguments of f. As a shortcut, we use $\mathbb{D}(f) = \mathbb{D}(in_1) \times \ldots \times \mathbb{D}(in_m) \times \mathbb{D}(out_1) \times \ldots \times \mathbb{D}(out_n)$.

In the following, we expect that functions do not have other than input and output arguments, which include also the return value. Note that an in-out argument (e.g., a parameter passed by reference) is split into one input and one output argument. Similarly, a global variable accessed by a function is rewritten into the corresponding input or/and output argument, depending on the mode of access (i.e., read or/and write).

Precise behavior of a function can be defined as a relation over values of input and output arguments of the function as follows.

Definition 5 (Relational Representation). *Let f be a function, then the relation $R^f \subseteq \mathbb{D}(f)$ is the* relational representation *of the function f, if R^f contains exactly all the tuples $\vec{v} = \langle v_1, \ldots, v_{|args^f|} \rangle$ such that the function f called with the input values $\langle v_1, \ldots, v_{|args^f_{in}|} \rangle$ can finish with the output values $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|} \rangle$.*

Note that Def. 5 admits multiple combinations of values of the output arguments for the same combination of values of the input arguments. This is useful to model nondeterministic behavior, and for abstraction of the precise behavior of a function. In this work, the summaries are applied in BMC. For this reason, the rest of the text in the section will be restricted to the following bounded version of Def. 5.

Definition 6 (Bounded Relational Representation). *Let f be a function and ν be a bound, then the relation $R^f_\nu \subseteq R^f$ is the* bounded relational representation *of the function f, if R^f_ν contains only the tuples representing computations with all loops and recursive calls unwound up to ν times.*

Then a summary of a function is an over-approximation of the set of precise behaviors of the given function under the given bound. In other words, each bounded function behavior is captured by a summary, but not necessarily each summary behavior belongs to the bounded function.

Definition 7 (Summary). *Let f be a function and ν be a bound, then a relation S such that $R^f_\nu \subseteq S \subseteq \mathbb{D}(f)$ is a* summary *of the function f.*

The relational view on a function behavior is intuitive but impractical for implementation. Typically, these relations are captured by means of logical formulas. Def. 8 makes a connection between these two views.

Definition 8 (Summary Formula)**.** *Let f be a function, ν a bound, σ_f a formula with free variables only from $args^f$, and S a relation induced by σ_f as $S = \{\vec{v} \in \mathbb{D}(f) \mid \vec{v} \models \sigma_f\}$. If S is a summary of the function f and bound ν, then σ_f is a* summary formula *of the function f and bound ν.*

A summary formula of a function can be directly used during construction of the PBMC formula to represent a function call. This way, the part of the SSA form corresponding to the subtree of a called function does not have to be created and converted to a part of the PBMC formula. Moreover, the summary formula tends to be more compact.

Example 5. *Considering the PBMC formula in Fig. 2.3, a formula $a_0 > 5 \Rightarrow ret_0 > 0$ represents a summary of the first (upper) call of function* `f`*.*

An important property of the resulting PBMC formula is that if it is unsatisfiable (as in Ex. 5) then also the formula without summaries (in Ex. 3) is unsatisfiable. Therefore, no errors are missed due to the use of summaries.

Lemma 1. *Let ϕ be a BMC formula of an unwound program P for a given bound ν, and let ϕ' be a BMC formula of P and ν, with some function calls substituted by the corresponding summary formulas bounded by ν. If ϕ' is unsatisfiable then ϕ is unsatisfiable as well.*

Proof. Without loss of generality, suppose that there is only one summary formula σ_f substituted in ϕ' for a call to a function f. If multiple summary formulas are substituted, we can apply the following reasoning for all of them.

Suppose that ϕ' is unsatisfiable and ϕ is satisfiable. From the satisfying assignment of ϕ, we get values $\langle v_1, \ldots, v_{|args^f|}\rangle$ of the arguments to the call to the function f. Assuming correctness of construction of the BMC formula ϕ, the function f given the input arguments $\langle v_1, \ldots, v_{|args^f_{in}|}\rangle$ can finish with the output arguments $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|}\rangle$ and with all loops and recursive calls unwound at most ν times. Therefore, by definition of the summary formula, the values $\langle v_1, \ldots, v_{|args^f|}\rangle$ also satisfy σ_f. Since the rest of the formulas ϕ and ϕ' is the same, the satisfying assignment of ϕ is also a satisfying assignment of ϕ' (up to SSA version renaming). Thus, we achieved a contradiction. □

2.2.5 Interpolation-Based Function Summaries

Among different possible ways to obtain a summary formula, we consider a way to synthesize summary formulas using Craig interpolation. To use interpolation, we use the PBMC obtained as described in Sect. 2.2.2. That is, the PBMC formula ϕ should have the form $\neg\pi \wedge \bigwedge\limits_{\hat{f}\in\hat{F}} \phi_{\hat{f}}$ such that every $\phi_{\hat{f}}$ symbolically represents the body of the function call $\hat{f}$. Moreover, the symbols of $\phi_{\hat{f}}$ shared with the rest of the formula correspond only to the input and output program variables.

If the PBMC formula is unsatisfiable, i.e., the program is safe, we can proceed with interpolation. The function summaries are synthesized as interpolants from a proof of unsatisfiability of the PBMC formula. In order to generate an interpolant, for each function call $\hat{f}$ the PBMC formula is split into two parts. First, $\phi_{\hat{f}}^{subtree}$ corresponds to the partitions representing the function call $\hat{f}$ and all the nested function calls. Second, $\phi_{\hat{f}}^{env}$ corresponds to the context of the call $\hat{f}$, i.e., to the rest of the encoded program.

$$\phi_{\hat{f}}^{subtree} = \bigwedge_{\hat{g}\in\hat{F}_\nu:\, subtree(\hat{f},\hat{g})} \phi_{\hat{g}}$$

$$\phi_{\hat{f}}^{env} = \neg\pi \wedge \bigwedge_{\hat{h}\in\hat{F}_\nu:\, \neg subtree(\hat{f},\hat{h})} \phi_{\hat{h}}$$

Therefore, for each function call $\hat{f}$, the summary-construction method separates the PBMC formula into $\phi_{\hat{f}}^{subtree} \wedge \phi_{\hat{f}}^{env}$ and synthesizes an interpolant $I_{\hat{f}}$ as follows:

$$\begin{cases} \phi_{\hat{f}}^{subtree} & \implies I_{\hat{f}} \\ I_{\hat{f}} \wedge \phi_{\hat{f}}^{env} & \implies \bot \end{cases}$$

Notably, by definition of Craig interpolation, the only free variables of $I_\nu^{\hat{f}}$ are from $var(\phi_{\hat{f}}^{subtree}) \cap var(\phi_{\hat{f}}^{env}) = args^f$. Thus the interpolant $I_{\hat{f}}$ is a summary formula for the function f.

The synthesized interpolant summaries are associated with the function calls by a mapping σ, i.e., $\sigma(\hat{f}) = I_{\hat{f}}$. Notably, we consider only a single summary per a function call for the sake of simplicity. This still means multiple summaries per a single function called multiple times.

The quality of summaries might depend on several different aspects. An interpolant (i.e., a summary) is not unique, and might be stronger or weaker than another interpolant. In our particular setting, a summary constructed by *McMillan*'s algorithm is more accurate and closer to the precise representation of the function body than a summary constructed by *Pudlák*'s algorithm. In addition to the choice of the interpolation algorithm, the SAT solver can also guide proof construction and post-process created proofs, e.g., reduce the proof and remove redundancies in its structure. The size and structure of a formula might affect the performance of a SAT solver while dealing with these formulas again: while verifying the same code with respect to different assertions (Sect. 2.2.6) or verifying another code with respect to the same assertions (Sect. 2.3). Thus, in our implementation we use the SAT solver PeRIPLO [119] which is able to perform the necessary manipulations with proofs and interpolation algorithms.

2.2.6 Using Function Summaries to Verify Different Assertions

We propose to reuse function summaries between verification tasks of checking the same program with respect to different assertions. If a summary of a function call exists before a verification run, we propose to substitute it instead of encoding the function body from scratch. Such substitution might fit well and the verification might succeed (therefore we expect performance speedup), or (due to an over-approximating nature of summaries) it might introduce spurious bugs which have to be immediately identified. A refinement of summaries is needed in the latter case.

Alg. 1 considers the case of a checking a given program with respect to a single assertion. Suppose, now we are dealing with a sequence of the assertions to be checked: $\{\pi\}_n^0$. It is often needed to check each of them separately to realize which of them holds and which of them does not. And it is essential to reuse function summaries, generated after a successful verification of assertion π_i, to check assertion π_{i+1}.

However, Alg. 1 is not given with summaries, but proceeds only by treating some function calls nondeterministically. Notably, this is equivalent to always substitute the weakest possible summary $\top$. Thus there is a need to enhance the method CreateFormula in Alg. 1 to support different substitution scenarios.

A mapping *substitution scenario* $\Omega : \hat{F}_\nu \to \{inline, sum, havoc\}$ determines how each function call should be handled. Initially, Ω depends on existence of function summaries. If a summary of a function $\hat{f}$ exists, it substitutes the body, i.e., $\Omega(\hat{f}) = sum$. If not, $\hat{f}$ is either represented precisely, i.e., $\Omega(\hat{f}) = inline$ or abstracted away, i.e., $\Omega(\hat{f}) = havoc$, depending on the current unwinding number ν. Let *dom* denote the domain of a mapping, an *initial scenario* Ω_0 is formalized as follows:

$$\Omega_0(\hat{f}) = \begin{cases} \mathit{sum}, & \text{if } \hat{f} \in \mathit{dom}(\sigma) \\ \mathit{inline}, & \text{if } \hat{f} \text{ is not recursive} \\ \mathit{inline}, & \text{if } \hat{f} \text{ is recursive and } \nu \text{ is not exceeded} \\ \mathit{havoc}, & \text{if } \hat{f} \text{ is recursive and } \nu \text{ is exceeded} \end{cases}$$

When a substitution scenario Ω_i leads to a satisfiable PBMC formula, a *refinement strategy* either shows that the error is real or looks for another substitution scenario Ω_{i+1}. In the latter case, Ω_{i+1} represents a tighter approximation, i.e., it refines Ω_i.

Counter-example guided refinement. We introduce a notion of Counter-example guided refinement (CEGR) for function summarization that generalizes the refinement procedure explained in Sect. 2.2.3. CEGR is based on analysis of the error trace, determined by a satisfying assignment. By construction of the PBMC formula, a variable $\mathit{callstart}_{\hat{f}}$ is evaluated to $\top$, if and only if the satisfying assignment represents a trace that includes the function call $\hat{f}$. If no function summary appeared along the error trace, the error is real.

$$\Omega_{i+1}(\hat{g}) = \begin{cases} \mathit{inline}, & \text{if } \Omega_i(\hat{f}) \neq \mathit{inline} \wedge \mathit{callstart}_{\hat{f}} = \top \\ \Omega_i(\hat{f}), & \text{otherwise} \end{cases}$$

Finally, we present Alg. 2, a revised version of Alg. 1 that is designed to be run in a loop verifying each of the assertions $\{\pi_i\}$ individually. In order to do it, the algorithm performs interpolation-based summary construction and CEGR. In addition to the set of inputs to Alg. 1 (including assertion π_i), Alg. 2 is given a mapping of the function calls by the summaries σ, synthesized after successful verification of assertions $\pi_0 \dots \pi_{i-1}$. Depending on σ, the algorithm initializes the substitution scenario Ω (line 2). In order to simplify the use of Ω, we merge the sets $\mathbb{P}$ and $\mathbb{R}$ (that were used to distinguish function calls to be inlined from the ones to be treated nondeterministically in Alg. 1) into $\mathbb{P}$ and apply the mapping Ω for each function call from $\mathbb{P}$ while constructing the PBMC formula (line 5). The substitution scenario can further be refined (line 16) due to CEGR strategy.

When the algorithm terminates with SAFE answer, method INTERPOLATE (line 7) is run to update summaries for each function call (as described in Sect. 2.2.5). Notably, the existing summaries remain valid but they are not necessarily accurate enough to prove the given assertion. Thus, the algorithm does not drop them, but conjoins with the newly generated ones.

```
Algorithm 2: Interpolating BMC for verifying different assertions

Input: Initial recursion depth: $\nu$; program unwound $\nu$ times:
       $P_\nu = (\hat{F}_\nu, \hat{f}_{main}, child)$; assertion to be checked: $\pi_i$; summaries
       mapping: $\sigma$
Output: Verification result: {SAFE, BUGGY, TIMEOUT}; detected recursion
        depth: $\nu$; error trace: $\epsilon$
Data: PBMC formula: $\phi$; current unrolling: $\mathbb{P}$; set of refinement
      candidates $\mathbb{P}$; current substituting scenario: $\Omega$
1  $\mathbb{P} \leftarrow \hat{F}_\nu \cup \{\hat{g} \notin \hat{F}_\nu \mid child(\hat{f}, \hat{g}), \hat{f} \in \hat{F}_\nu\}$;   ▷ initialize unrolling
2  $\Omega \leftarrow$ INIT($\mathbb{P}, \sigma$);   ▷ initialize substitution scenario
3  while ($\neg$TIMEOUT) do
4      $\phi \leftarrow \neg\pi_i \wedge \bigwedge_{\hat{f} \in \mathbb{P}: \Omega(\hat{f})=inline}$ CREATEFORMULA($\hat{f}$) $\wedge$
           $\bigwedge_{\hat{g} \in \mathbb{P}: \Omega(\hat{g})=sum}$ SUMMARY($\hat{g}$) $\wedge$
               $\bigwedge_{\hat{h} \in \mathbb{P}: \Omega(\hat{h})=havoc}$ NONDET($\hat{h}$);
5      $result, sat\_assignment \leftarrow$ SOLVE($\phi$);   ▷ SAT-solve the PBMC formula
6      if ($result$ = UNSAT) then
7          foreach ($\hat{f} \in \mathbb{P}$) do $\sigma(\hat{f}) \leftarrow \sigma(\hat{f}) \wedge$ INTERPOLATE($proof, \hat{f}$);
8          return SAFE, $\nu, \varnothing$;
9      else
10         $\epsilon \leftarrow$ EXTRACTCE($sat\_assignment$);   ▷ extract an error trace
11         $\mathbb{R} \leftarrow$ EXTRACTCALLS($\epsilon$);   ▷ get function calls that affect the satisfiability
12         if ($\mathbb{R} = \varnothing$) then
13             return BUGGY, $\nu$, $\epsilon$;
14         else
15             $\mathbb{P} \leftarrow \mathbb{P} \cup \mathbb{R} \cup \{\hat{g} \notin \mathbb{R} \mid child(\hat{f}, \hat{g}), \hat{f} \in \mathbb{R}\}$;
16             $\Omega \leftarrow$ REFINE($\Omega, \mathbb{P}, \mathbb{R}$);   ▷ refine calls from $\mathbb{R}$
17             $\nu \leftarrow$ MAXCHAINLENGTH($\mathbb{P}$);   ▷ update the depth
18 end
19 return TIMEOUT;
```

Alg. 2 is implemented and evaluated within a tool FunFrog. We discuss it in details in Sect. 5.1.2 and Sect. 5.2.

2.2.7 Detecting and Exploiting the Assertion Implication Relation

In Sect. 2.2.6 we proposed the summarization-based model checking procedure (Alg. 2) for verifying a sequence of assertions $\{\pi\}_n^0$. The main feature of Alg. 2

is that for checking an assertion π_i, it reuses the summaries ($\sigma(\hat{f})$ for each function call $\hat{f} \in \hat{F}_v$) already created while checking the previous assertions $\{\pi\}_{i-1}^{0}$. However, there is no guarantee that $\sigma(\hat{f})$ is accurate enough for checking π_i. Thus, there might be needed to refine summaries, which is expensive since it requires extra efforts on SSA encoding, solving and interpolating. On the other hand, some assertions from the sequence might be redundant, allowing to skip the whole model checking procedure.

We propose to detect dependencies among assertions, using static analysis before actual verification. It allows detecting the Assertion Implication Relation (AIR). AIR specifies a directed acyclic graph, possibly with several connected components that can be used as follows. If the goal is *to prove program correctness*, it is needed to collect the strongest assertions (of all connected components) with respect to AIR and check them in a single verification run. If this check succeeds, all the assertions of the program are valid. If the goal is *to detect all violated assertions*, it makes sense to run incremental analysis, separately for each ordered chain of assertions. In general, it can be done in the forward or backward direction. For example, if the incremental algorithm starts from the bottom of the dependency graph, it checks the weakest assertion first, then proceeds with a stronger until it reaches the strongest one (i.e., the top of the chain). Whenever it finds a violation, it automatically means that all stronger assertions are also violated.

In the rest of the section we elaborate on detecting and exploiting AIR. Without loos of generality, we assume that the given sequence of assertions $\{\pi\}_{n}^{0}$ is ordered by the appearance in the control-flow graph of the program. That is, for any two numbers i, j such that $0 \leq i < j \leq n$, assertion π_i is closer to the program entry point than assertion π_j. Let $\Delta(\pi_i, \pi_j)$ denote the path in the control-flow graph between locations of the two assertions.

Definition 9. *Given a program P_v containing two assertions π_0 and π_1, we say that the assertion π_0* implies *the assertion π_1 iff the following* Hoare triple *is valid:*

$$\{\pi_0\}\,\Delta(\pi_0, \pi_1)\,\{\pi_1\}$$

The validity of the Hoare triple in Def. 9 means that π_1 holds whenever π_0 holds in all executions of the program allowed by $\Delta(\pi_0, \pi_1)$. Given the valid local implication relation between assertions π_0 and π_1, we refer to π_0 as a *stronger* assertion, and to π_1 as a *weaker* assertion. We are interested in determining whether, in a given program with a given sequence of assertions $\{\pi\}_{n}^{0}$ there are

Algorithm 3: Computing AIR

```
Input: Program: $P_\nu$, sequence of assertions: $\{\pi\}_n^0$
Output: Assertion Implication Relation: $AIR$
Data: Disjoint sets of variables corresponding to the variable dependency
      classes: $VD$, the assertion dependency relation: $AD$
1 $VD \leftarrow \textsc{GetDependentVariables}(P_\nu)$;
2 $AD \leftarrow \textsc{GetDependentAssertions}(VD)$;
3 $AIR \leftarrow \{(i,j) \in AD \mid \textsc{implies}(\pi_i, \pi_j) = true\}$;
```

pairs of assertions π_i and π_j such that π_j holds whenever π_i holds in all executions of the program. We compute AIR, which consists of (a subset of) such pairs where the implication follows from the SSA steps between the assertions. We present a high-level overview of the algorithm for detecting assertion implications in Alg. 3.

We compute the implication relation in two phases. First, the classes of *dependent* assertions *AD* are detected using a syntactic analysis on the SSA form. We say that two variables x and y are *dependent* when the value of x potentially affects the value of y. This notion of variable dependencies is exactly the same as in *program slicing* [134]. The dependency relation is reflexive, transitive and symmetric and therefore an equivalence relation which groups all variables into dependency classes. We further extend it to assertions. Two assertions π_0 and π_1 are said to be *dependent* if there exists a variable x in π_0 and a variable x' in π_1 such that x and x' are dependent. Unlike a variable, if an assertion π_0 depends on an assertion π_1, and the assertion π_1 depends on an assertion π_2, this does not imply that π_0 depends on π_2, since the dependencies might result from variables not shared by π_0 and π_2. The assertion dependency relation can be constructed in an iterative procedure over the set of assertion pairs. For each pair, it explores the dependency classes of the variables involved in the assertions. If two assertions contain variables of the same dependency class, the assertions are dependent and are going to be included into the relation *AD*.

In the second phase of constructing *AIR*, the assertion dependency relation is refined to contain only the pairs of assertions (π_i, π_j) such that π_i implies π_j. This is done by constructing the formula corresponding to Def. 9 and invoking the SAT solver through the IMPLIES call in Alg. 3. This is sound up to the number ν of loop iterations and recursive function calls. Finally, the *AIR* defines an *assertion* *implication graph* representing all revealed implication relationships between the guarded assertions. More formally,

Definition 10. *Given a program P_v with a sequence of assertions $\{\pi\}_n^0$ the* assertion implication graph *of P_v is a graph $G_U = (\{\pi\}_n^0, E)$ where $E = \{(\pi_i, \pi_j) \mid \pi_i$ implies π_j in $P_v\}$.*

We propose to traverse G_U before the BMC run to minimize the search for holding assertions and avoid checking all assertions one by one. Our solution is based on the two following ideas: 1) If an assertion π_i is proven to hold, all weaker assertions π_j (i.e., $\{\pi_j \mid (\pi_i, \pi_j) \in E\}$) are implicitly proven to hold. 2) If an assertion π_k is proven to fail, all stronger assertions π_j (i.e., $\{\pi_k \mid (\pi_j, \pi_k) \in E\}$) are implicitly proven to fail.

We further expand these ideas into the two complementing strategies for the efficient detection of assertions which hold in the program. We denote the nodes of G_U that do not have incoming edges as $\{\pi_s\}$. These correspond to the *strongest* assertions in the program. Similarly, we denote the edges with no outgoing edges as $\{\pi_w\}$, and these correspond to the *weakest* assertions in the program.

In the first (*forward*) strategy, a BMC tool traverses G_U starting from $\{\pi_s\}$ in the depth-first order. For each assertion node π_i, if there exists a holding predecessor π_j, the BMC tool concludes that π_i also holds. Otherwise, it verifies the program with respect to π_i. This strategy is efficient in cases when there are many holding assertions in the program.

In the second (*backward*) strategy, a BMC tool traverses G_U in reverse, starting from $\{\pi_w\}$. For each assertion node π_k, if there exists a failing successor π_j, the BMC tool concludes that π_k also fails. Otherwise, it verifies the program with respect to π_k. This strategy is efficient in cases when there are many assertions which fail in the program.

Both strategies and the preprocessing Alg. 3 are implemented and evaluated within a tool FunFrog. We discuss them in details in Sect. 5.1.3.

2.3 Checking Software Versions within Bounded Model Checking

This section presents an incremental SAT-based BMC approach that uses function summarization for checking sequences of software versions. It receives two

```
int g(int a, int b)
{
  if (a < b)
    return a;
  return a - b + 1;
}

int f (int a, int b
{
  return g (a, b);
}

main()
{
  int x = 0;
  int y = nondet();
  int z = nondet();

  if (y > 0)
    x = f(y, z);

  assert(x > 0);
}
```

(a) Original program

```
int g(int a, int b)
{
  if (a < b)
    return a;
  return a - b;
}

int f (int a, int b)
{
  return g (a, b) + 1;
}

main()
{
  int x = 0;
  int y = nondet();
  int z = nondet();

  if (y > 0)
    x = f(y, z);

  assert(x > 0);
}
```

(b) Modified program

Figure 2.5. Two versions of the C program.

versions of a program, an old and a new one, and a bound to be used to unwind the loops and recursive function calls in both program versions. We assume, both versions share the same set of assertions to be checked. Given that the old version satisfy the given assertions up to the predefined bound, the goal of the approach is to verify that the assertions hold in the new version as well. An example of a program change is illustrated in Fig. 2.5. The increment operation is lifted from one function to another one.

Our BMC-based incremental algorithm maintains *function summaries* that, in our case, over-approximate the bounded behavior of the functions, computed by means of Craig interpolation (described in Sect. 2.2.5). The core idea of the algorithm is to check if the old function summaries still over-approximate the bounded behavior of the corresponding functions in the new program version. If the check fails for some function call, it needs to be propagated by the call tree traversal to the caller of the function. If the check fails for the root of the call tree (i.e., the "main" function of the program), the whole program should be verified from scratch. On the other side, if for each function call there exists an ancestor function with the valid old summary then the new program version is safe up to the predefined bound. Finally, for functions whose old summaries are not valid any longer, the new summaries are synthesized using Craig interpolation.

The incremental model checking algorithm implements the refinement strategy for dealing with spurious behaviors that can be introduced during computation of the over-approximated summaries. The refinement procedure for the incremental checks builds on ideas of using various summary substitution scenarios (described in Sect. 2.2.6). We further extend it to simplify the summary validity checks by substituting the summaries of nested function calls which are already proven valid. Failures of such checks may be due to the use of summaries which are not accurate enough. In such case, the refinement is used to expand the involved function calls on demand.

In the rest of the section, we present this basic algorithm in more details, describe its optimization with a refinement loop and prove its correctness.

2.3.1 Incremental BMC Algorithm

We proceed by presenting our algorithm for checking sequences of software versions (Alg. 4). As an input, Alg. 4 takes the unwound program together with the old and new versions of the SSA form for each function call, and a mapping σ from the function calls in the new version to the summaries from the old version. We denote the domain of a mapping by *dom*. If $dom(\sigma) = \varnothing$ (line 1), the algorithm runs verification of the new version from scratch (Alg. 2). Thus, we assume that $dom(\sigma)$ contains at least a summary for $\hat{f}_{main}$.

The algorithm starts with computing a set *changed* that collects the function calls corresponding to the functions, on which the old and the new versions disagree (line 3). In our implementation, we syntactically compare the corresponding SSA forms.

The algorithm maintains a work-list *WL* of function calls that require rechecking. Initially, *WL* is populated by the elements of the previously computed set *changed* (line 4). Then the algorithm repeatedly removes a function call $\hat{f}$ from

Algorithm 4: Incremental BMC algorithm

```
Input: Unwound program: $P_\nu = (\hat{F}_\nu, \hat{f}_{main})$ with function calls $\hat{F}_\nu$, SSA
       forms of both versions of $P_\nu$: $code_{old} : \hat{F}_\nu \implies$ SSA and
       $code_{new} : \hat{F}_\nu \implies$ SSA, summaries mapping: $\sigma : \hat{F}_\nu \implies \mathbb{S}$
Output: Verification result: {SAFE, BUGGY}, actualized summaries
        mapping: $\sigma$
Data: temporary sets of function calls: $changed, WL \subseteq \hat{F}_\nu$, PBMC formula:
      $\phi$, set of invalid summaries: $invalid \subseteq \mathbb{S}$, refutation: $proof$

1  if ($dom(\sigma) = \emptyset$) then
2      VERIFYFROMSCRATCH($P_\nu$);                                   ▷ run Alg. 2
3  $changed \leftarrow \{\hat{f} \mid \hat{f} \in \hat{F}_\nu$, s.t. $code_{old}(\hat{f}) \not\equiv code_{new}(\hat{f})\}$;
4  $WL \leftarrow changed$;
5  $invalid \leftarrow \emptyset$;
6  while ($WL \neq \emptyset$) do
7      choose $\hat{f} \in WL$, s.t. $\forall \hat{g} \in WL : \neg subtree(\hat{f}, \hat{g})$;
8      $WL \leftarrow WL \setminus \{\hat{f}\}$;
9      if ($\hat{f} \in dom(\sigma) \wedge \sigma(\hat{g}) \notin invalid$) then
10         $\phi \leftarrow$ CREATEFORMULA($\hat{f}$);
11         $result, proof \leftarrow$ SOLVE($\phi \wedge \neg\sigma(\hat{f})$);
12         if ($result =$ UNSAT) then
13             foreach ($\hat{g} \in \hat{F}_\nu : subtree(\hat{f}, \hat{g})$) do
14                 $\sigma(\hat{g}) \leftarrow$ INTERPOLATE($proof, \hat{g}$);
15             end
16         else $invalid \leftarrow invalid \cup \{\sigma(\hat{f})\}$;
17     if ($\hat{f} \notin dom(\sigma) \vee \sigma(\hat{f}) \in invalid$) then
18         if ($\hat{f} \neq \hat{f}_{main}$) then
19             $WL \leftarrow WL \cup \{parent(\hat{f})\}$;                ▷ check the parent
20         else
21             return VERIFYFROMSCRATCH($P_\nu$);                     ▷ run Alg. 2
22 end
23 return SAFE, $\sigma$;                                            ▷ new version is safe
```

WL and attempts to check validity of the corresponding summary in the new version. Note that the algorithm picks $\hat{f}$ so that no function call in the subtree of $\hat{f}$ occurs in *WL* (line 7). This ensures that summaries in the subtree of $\hat{f}$ were already analyzed (shown either valid or invalid).

The actual summary validity check happens on lines 10-11. First, the PBMC formula encoding the subtree of $\hat{f}$ (with respect to the new version of the SSA form) is constructed and stored as ϕ. Then, conjunction of ϕ with negated summary of $\hat{f}$ is passed to a SAT solver for the satisfiability check. If unsatisfiable, the summary is still a valid over-approximation of the function's behavior. Here, the algorithm obtains a proof of unsatisfiability which is used later to create new summaries to replace the invalid or missing ones (line 13-15). If satisfiable, the summary is not valid for the new version (line 16). In this case, the check is propagated to the function caller (line 19). If there is no caller to propagate the check (i.e., the old function summary for $\hat{f}_{main}$ is invalid) the entire program should be verified from scratch (line 21).

Note that the algorithm cannot identify the real error by itself, since for this goal it needs to create and solve a formula of the form $\phi \wedge \neg\pi$, which is the typical subroutine of the Alg. 2. However, this is the worst case scenario, and is possible only in case if the old function summary for $\hat{f}_{main}$ is invalid. In all other cases, the algorithm is able to prove the new version safe without running the the verification from scratch.

Note that if the chosen function call $\hat{f}$ has no summary, the check is propagated to the caller immediately (line 17) and the summary of $\hat{f}$ is created later when the check succeeds for some ancestor function call of $\hat{f}$.

Example 6. *As a demonstration of the incremental BMC algorithm, consider a modified version of the program on Fig. 2.5(b). It is created by lifting an increment operator one level up on the call tree (i.e, from function* `g` *to its caller, function* `f`*). Assume the program on Fig. 2.5(a) is verified by Alg. 2 and the following summaries of the functions* `main`*,* `f`*,* `g` *exist:*

$$\sigma(\mathtt{main}) = x_2 > 0 \tag{2.1}$$

$$\sigma(\mathtt{f}) = (f_{a_0} > 0) \implies (f_{ret_0} > 0) \tag{2.2}$$

$$\sigma(g) = (g_{a_0} > 0) \implies (g_{ret_0} > 0) \tag{2.3}$$

Note that the summary of function `main` *is expressed over variable x_2, the only common one between the body of the function and the assertion expression. The summaries of functions* `f` *and* `g` *are expressed over their input and output variables respectively (i.e., the common language of the function and its caller). Furthermore, formulas* (2.2) *and* (2.3) *are semantically equivalent, but syntactically different.*

Our approach first checks if $\sigma(g)$ still over-approximates `g`*.*

In our example, the summary-check of $\sigma(g)$ does not succeed. As a witness of this, one can chose equal positive numbers to be the values for the function parameters. For example, if $g_a = g_b$ then $g_{ret} = 0$ which contradicts the formula (2.3).

Then the algorithm proceeds with checking validity of $\sigma(f)$ and proves it. During this check, the new behavior of g was encoded and its old summary (2.3) *was not used. Given the proof of validity of $\sigma(f)$, we apply interpolation and update $\sigma(g)$ as follows:*

$$\sigma(g) = (g_a > 0) \implies (g_{ret} \geq 0) \tag{2.4}$$

Since there was no change in function `main`*, the algorithm terminates. The updated summary* (2.4) *is going to be stored instead of* (2.3) *and used when another program version arrives.*

2.3.2 Tree Interpolation for SAT Equisatisfiability

The approach presented in Sect. 2.3.1 can also be viewed as a technique for reusing refutation proofs from one unsatisfiable propositional formula to speed up SAT checks for a slightly different formula. For instance, a large number of tools and algorithms within verification and program analysis make many quick calls to SAT solvers. A large number of these calls are often similar. We believe that the SAT view of the algorithm, presented in the previous section, will be applicable in a variety of settings.

Throughout this section, we consider an inconsistent formula $\Phi = \bigwedge_{i=1}^{n} \phi_i$ and targeting to decide satisfiability of another formula $\Psi = \bigwedge_{i=1}^{n} \psi_i$. We require Φ and Ψ, to consist of n conjuncts. Moreover, we assume that Φ and Ψ share a common subformula, i.e., there exist $S \subseteq \{1 \dots n\}$, such that $\forall k \in S$, $\phi_k \iff \psi_k$ and $\forall \ell \notin S$, $\phi_\ell \not\Leftrightarrow \psi_\ell$. Then the check for unsatisfiability of Ψ can proceed by checking unsatisfiability of the subformulas Ψ using interpolants obtained from Φ. We rely on the tree interpolation property formalized in Sect. 2.1.1.

Interpolation-based SAT solving algorithm (outlined in Alg. 5) implements the core idea[1] of Alg. 4. The algorithm is given the formulas Φ and Ψ (Φ is unsatisfiable), the tree T and a family of T-tree interpolation algorithms. Then Alg. 5 decides satisfiability of Ψ.

[1]In order to simplify presentation, we omitted the subroutine to re-construct interpolants in Alg. 5, as done in Alg. 4.

Algorithm 5: Deciding equisatisfiability of two SAT formulas

```
Input: Formulas $\Phi = \bigwedge_{i=1}^{n} \phi_i$ and $\Psi = \bigwedge_{i=1}^{n} \psi_i$, tree $T = (V, E)$, $V = \{1, \ldots, n\}$
       a family of $T$-tree interpolation algorithms that result in $I_{\Phi,K_i}$
Output: Satisfiability of $\Psi$: $\{$SAT, UNSAT$\}$
Data: Temporary sets $WL, K_i \subseteq \{1, \ldots, n\}$
1  $WL \leftarrow \{i \mid 1 \leq i \leq n$ and $\phi_i \not\Longleftrightarrow \psi_i\}$;
2  while ($WL \neq \emptyset$) do
3      choose $i \in WL$, s.t. $\forall j \in WL : j \neq i \implies i \not\sqsubseteq j$;
4      $WL \leftarrow WL \setminus \{i\}$;
5      $K_i = \{j \mid i \sqsubseteq j\}$;
6      if ($\psi_i \wedge \bigwedge_{j \in K_i} \psi_j \not\Longrightarrow I_{\Phi,K_i}$) then
7          if ($i = root(T)$) then
8              return SAT;
9          else
10             $WL \leftarrow WL \cup \{parent(T, i)\}$;
11 end
12 return UNSAT;
```

Similarly to Alg. 4, Alg. 5 maintains the work-list of natural numbers *WL*, initially populated by the numbers that identify non-equal conjuncts of Φ and Ψ. In each iteration, the algorithm choses the least element i of *WL* corresponding the deepest node in T, and checks validity of the interpolant I_{Φ,K_i} (line 3). If the check succeeds, the algorithm removes i from *WL* and goes to the next iteration. If the check does not succeed, *WL* replaces i by the parent node of i in T and goes to the next iteration. The algorithm terminates when *WL* is empty (i.e., Ψ is unsatisfiable) or when the check for the *root* node does not succeed (i.e., Ψ is satisfiable).

The worst case scenario of Alg. 5, i.e., checking satisfiability of the entire formula Ψ, is done also via checking validity of the interpolant. Indeed, in this case, the current element i of *WL* corresponds to the tree *root*, whose interpolant is equal to $\bot$: $I_{\Phi,K_{root}} = \bot$. Finally, if the check $\psi_{root} \wedge \bigwedge_{j \in K_{root}} \psi_j \implies I_{\Phi,K_{root}}$ fails (line 8), then $\Psi = \psi_{root} \wedge \bigwedge_{j \in K_{root}} \psi_j$ is satisfiable. In contrast, the similar scenario of Alg. 4 is not subsumed by the summary validity check. The reason is that

Alg. 4 manipulates the program call tree without respect to the assertion (recall the PBMC encoding and interpolation from Sect. 2.2.5). That is, the root of the program call tree is the “main” function call, and its function summary may not necessarily be $\perp$. Taking it into account, Alg. 5 can be seen as a generalized representative of Alg. 4. It is also more compact, easily understandable and potentially applicable in the contexts not directly related to model checking.

2.3.3 Optimization and Refinement

To optimize Alg. 4 outlined in Sect. 2.3.1, old function summaries can be used to abstract away the function calls. Consider the validity check of a summary of a function call $\hat{f}$. Suppose there exists a function call $\hat{g}$ in the subtree of $\hat{f}$ together with its old summary, already shown valid. Then this summary can be substituted for $\hat{g}$, while constructing the PBMC formula of $\hat{f}$ (line 10). This way, only a part of the subtree of $\hat{f}$ needs to be traversed and the PBMC formula ϕ can be substantially smaller compared to the encoding of the entire subtree.

If the resulting formula is satisfiable, it can be either due to a real violation of the summary being checked or due to too coarse summaries used to substitute some of the nested function calls. In our incremental BMC algorithm, this is handled in a similar way as in the refinement of the standalone verification by analyzing the satisfying assignment. The set of summaries used along the counter-example is identified. Then it is further restricted by dependency analysis to only those possibly affecting the validity. Every summary in the set is marked as *inline* in the next iteration. If the set is empty, check fails and the summary is shown invalid. This refinement loop (replacing lines 10, 11 in Alg. 4) iterates until validity of the summary is decided.

Regarding complexity, in the worst case scenario, i.e. when a major change occurs, the entire subtree is refined one summary at a time for each node of the call tree. This may result in a number of SAT solver calls quadratic in the size of the call tree, where the last call is the verification of the entire program from scratch. In this section, we focus on incremental changes and thus for most cases there is no need for the complete call graph traversal. Moreover, the quadratic number of calls can be easily mitigated by limiting the refinement laziness using a threshold on the number of refinement steps and disabling this optimization when the threshold is exceeded.

2.3.4 Soundness of the Incremental BMC Algorithm

This section proves the correctness of our Alg. 4, i.e., given an unwinding bound ν and an assertion π, the verification of the new version against π always terminates with the correct answer with respect to ν. We expect the same ν and π for the old and new versions. To ensure correctness, if the user increases ν for a specific loop, the corresponding function has to be handled as if modified.

Let the PBMC formula encoding the subtree of a function call $\hat{f}$ is denoted $\phi_{\hat{f}}$. By definition, the entire PBMC formula $\Phi = \neg\pi \wedge \phi^{subtree}_{\hat{f}_{main}}$ is a conjunction of negation of the assertion and the PBMC formulas encoding the bodies of each function call from $\hat{F}_\nu$, thus, $\phi^{subtree}_{\hat{f}_{main}} = \bigwedge_{\hat{f} \in \hat{F}_\nu} \phi_{\hat{f}}$. To simplify further presentation, we assume that the position of each conjunct $\phi_{\hat{f}}$ in the PBMC formula Φ is fixed and determined by $pos(\hat{f}, \Phi)$.

Recall that the summary of $\hat{f}$ generated with respect to the assertion π is specified by the mapping σ, i.e., $\sigma(\hat{f})$. For the interpolating algorithm, we assume that it has the tree interpolating property, i.e., the resulting summaries are at least as strong as those given by *Pud* algorithm (recall discussion in Sect. 2.1.1).

The key insight for proving correctness is that after each successful run of Alg. 4 (i.e., when SAFE is returned), the following two properties are maintained.

$$\neg\pi \wedge \sigma(\hat{f}_{main}) \implies \bot \tag{2.5}$$

Given each function call $\hat{f}$ and its children calls $\hat{g}_1, \ldots, \hat{g}_n$:

$$\sigma(\hat{g}_1) \wedge \ldots \wedge \sigma(\hat{g}_n) \wedge \phi_{\hat{f}} \implies \sigma(\hat{f}) \tag{2.6}$$

In the following two lemmas, we first show that properties (2.5, 2.6) hold after an initial whole-program check. Then we show that the properties are maintained between individual successful checks of software versions.

Lemma 2. *After an initial whole-program check (Alg. 2), the properties (2.5, 2.6) hold over the call tree annotated by the generated interpolants.*

Proof. Recall that the summaries are constructed only when the program is SAFE. In other words, the PBMC formula $\Phi = \neg\pi \wedge \phi^{subtree}_{\hat{f}_{main}}$ us unsatisfiable, i.e., $\Phi \implies \bot$. Thus, by definition of interpolation, $\neg\pi \wedge I_{\hat{f}_{main}}$ is obviously unsatisfiable, i.e., property (2.5) holds.

The program call tree induces a tree $T = (V, E)$, where $V = \{0, \ldots, |\hat{F}_\nu| + 1\}$ and $E = \{(0,1), (0, pos(\hat{f}_{main}, \Phi)\} \cup \{(i,j) \in V \times V \mid \exists \hat{f}, \hat{g} \in \hat{F}_\nu \text{ s.t. } i = pos(\hat{f}, \Phi),$ $j = pos(\hat{g}, \Phi)$, and $child(\hat{f}, \hat{g})\}$. Intuitively, V gathers the positions of conjuncts, each of which encodes body of a function call in the PBMC formula Φ. In addition, V has two nodes, "0" and "1", corresponding to the *root* of T (which can be treated as an invisible $\top$ conjunct of Φ) and $\neg\pi$ respectively. Similarly, E gathers the child-dependencies between the function calls and two additional edges connecting *root* with $\neg\pi$ and *root* with $\phi_{\hat{f}_{main}}$.

Tree T is used by the tree interpolation algorithm to generate summaries, and it applies to a proof of unsatisfiability of Φ. Let for a function call $\hat{f}$, the corresponding conjunct $\phi_{\hat{f}}$ in Φ has the position $i \in V$. Then the conjuncts with the positions $j \in V$, such that $(i, j) \in E$, correspond to the children calls $\hat{g}_1, \ldots, \hat{g}_n$ of $\hat{f}$. Thus, the resulting interpolants (and in turn, summaries) $I(\hat{g}_1) \ldots I(\hat{g}_n)$ satisfy the T-tree interpolation property: $I(\hat{g}_1) \wedge \ldots \wedge I(\hat{g}_n) \wedge \phi_{\hat{f}} \implies I(\hat{f})$, which is exactly property (2.6). □

Lemma 3. *Properties (2.5, 2.6) are reestablished whenever the incremental BMC algorithm (Alg. 4) successfully finishes (i.e.,* SAFE *is returned).*

Proof. Consider two possibilities when the algorithm returns SAFE. In the first case, it is obtained by the verification from scratch (line 1 or 20), immediately allowing us to apply Lemma 2. In the second case, SAFE is obtained after a finite number of summary checks for a subset of function calls (line 23). Therefore, the summary of the function call $\hat{f}_{main}$ remains unchanged, and property (2.5) holds.

In the rest of the proof, we need to show that property (2.6) holds whenever the summaries for a strict subset of function calls $S \subset \hat{F}_\nu$ are recomputed and $\hat{f}_{main} \notin S$. Thus, each function call $\hat{f}$ can either 1) be a parent call of the function call with recomputed summary, or 2) itself have a recomputed summary, or simply 3) be a function call that was never touched by the algorithm. Consider each of these cases:

1) $\hat{f} \notin S$, and some its children call $\hat{g}_i \in S$. It is easy to see (line 14) that a child call of $\hat{f}$ is recomputed only if the summary check of $\hat{f}$ succeeded. Thus, all other children calls $\hat{g}_1, \ldots, \hat{g}_n$ of $\hat{f}$ are also recomputed from the unsatisfiable formula $\neg\sigma(\hat{f}) \wedge \phi_{\hat{f}}^{subtree}$. This lets us proceed in the similar manner, as for the proof of Lemma 2, to show that $I(\hat{g}_1) \wedge \ldots \wedge I(\hat{g}_n) \wedge \phi_{\hat{f}} \implies I(\hat{f})$, where each I is an interpolant constructed for the correspondent partition of the PBMC formula. Furthermore, by definition of the interpolant, $\neg\sigma(\hat{f}) \wedge I(\hat{f})$ is unsatisfiable, and consequently, $I(\hat{f}) \implies \sigma(\hat{f})$. Conjoining the two derived implications, we get property (2.6).

2) $\hat{f} \in S$. Since all the function calls from the subtree of $\hat{f} \in S$ (including $\hat{f}$) are recomputed, thus property (2.6) is guaranteed by definition of the tree interpolation.

3) $\hat{f} \notin S$ and for all the children calls $\hat{g}_1, \ldots, \hat{g}_n \notin S$. Since all the function cals from $\hat{f}$, $\hat{g}_1, \ldots, \hat{g}_n$ have the old valid summaries, property (2.6) is guaranteed by direct application of Lemma 2.

□

We now show that the properties (2.5, 2.6) are strong enough to show that the whole program is safe.

Theorem 2. *When the program call tree annotated by interpolants satisfies the properties (2.5, 2.6), then $\neg\pi \wedge \phi^{subtree}_{\hat{f}_{main}} \implies \bot$ (i.e., the whole program is safe).*

Proof. Property (2.5) yields $\neg\pi \wedge \sigma(\hat{f}_{main}) \implies \bot$. Repeated application of property (2.6) to substitute all interpolants on the right hand side yields the claim $\neg\pi \wedge \phi^{subtree}_{\hat{f}_{main}} \implies \bot$. □

2.4 Related Work to Function Summarization

In this section we give an overview of different approaches to construct function summaries and use them in formal verification. The techniques related to Craig interpolation, our underlying engine for summaries' synthesis, are discussed in Sect. 2.1.2. Likewise, the techniques related to incremental verification are discussed in Sect. 4.4.

Function summaries date back to Hoare logic [79]. Hoare proposed to attach every function F with a pre-condition *pre* and a post-condition *post* (denoted as $\{pre\}F\{post\}$). The pair (*pre*, *post*) can be seen as an over-approximating function summary since it specifies all behaviors of the function that start in a state satisfying *pre* and finish in a state satisfying *post*. Notably, if the Hoare triple $\{pre\}F\{post\}$ is valid then the summary allows all set of exact behaviors of F and possibly more (so does our summary in Def. 7 as well).

Since then a variety of approaches were proposes for computing summaries. For instance, [117] presents an algorithm to compute function summaries explicitly using the data-flow analysis. That is, the control-flow graph of a program is traversed and the sets of initial and final states of each function call are incrementally recorded until no new states is discovered. The idea was further employed in the tool Bebop [12] (a part of the framework SLAM [13]) that use BDDs to represent the program states. Due to their high sensitivity to the number of variables, BDDs were replaced by SAT- and QBF-based techniques in the further work by [15]. As admitted in [15], QBF queries still constitute a major bottleneck. We propose a less expensive procedure to extract multiple function summaries from a single proof of unsatisfiability of a BMC formula.

The raise of SAT-based model checking techniques instigated the research for over-approximating function behaviors instead of explicit enumerating them. In particular, bounded model checkers Saturn [135; 136] and Calysto [9] generate summaries for each function by an iterative discovery of modification of variable values in the function behaviors. Such computation requires many calls to a SAT solver, but can end up with more general summaries than [117]. This is conceptually different approach also to our Alg. 2 that requires a single proof of unsatisfiabililty of the entire program and synthesizes all the summaries at once. Despite all approaches described so far are distinguished by the methods to synthesize summaries, they all agree on the way of using them: once computed, the summaries can substitute other calls of the same function whenever they need to be processed by the model checking algorithm once again.

As we mention in Sect. 2.1.2, there is an extension of LAWI that supports handling function calls [100]. It also uses function summaries. It extends symbolic execution to remember a reason for infeasibility of an execution path, i.e., a blocking annotation. Blocking annotations are used to reject other execution paths as early as possible. Compared to our technique, lazy annotation uses interpolation to derive and propagate the blocking annotations backwards for every program instruction. If the annotation is to be propagated across a function call, a function summary merging blocking annotations from all paths through the function is generated and stored for a later use. Our technique uses interpolation on the whole BMC formula and creates one function summary from one interpolant.

Algorithmically, the closest body of our BMC-based approach with Recursion Depth Detection (Alg. 1) is the Corral [93] model checker. Unlike in our approach, in the Corral, 1) the depth of recursion is bounded by a user-supplied recursion depth and 2) an external tool [62] is used to generate function summaries which in general may not be helpful to verify the given assertion. Our approach is able to generate relevant function summaries by itself. Moreover, it forces summaries to be bit-precise and highly related to the given assertion.

The most recent verification framework that generates and manipulates function summaries is SeaHorn [71]. SeaHorn synthesizes summaries by encoding and solving a system of non-linear Horn clauses (see Sect. 4.1.2 for more detail) that requires developing an appropriate SMT solver that admits quantifier elimination (see Sect. 3.1.1 for more detail) for each particular first-order theory (including bit-vectors). In contrast, we are able to use already existing SAT-based methods and easily allow bit-precise reasoning.

To finalize our brief overview of related methods, we stress the attention again on the over-approximating nature of the function summaries used in model checking. Because of this, the summaries are sometimes referred to as *may*-summaries, i.e., that they are valid for *all* function executions. In contrast, dynamic analysis typically manipulates the *must*-summaries, i.e., that they are valid for *some* function executions. While may-summaries are used to prove the absence of bugs in the program, must-summaries are used to prove the existence of bugs in the program. There are methods to create must-summaries by analyzing concrete behaviors of a program during its execution or analyzing logs after the execution [65; 36]. There are methods that combine may- and must-analysis [66], but still this line of research remains orthogonal to ours.

2.5 Summary of Contributions

In this chapter, we contributed a FIV technique for BMC (left branch in Fig. 1.1) that searches for counter-examples during the bounded exploration of the program search space. Our main contribution is the framework for synthezing a reusable specification of the program safe up to the given bound. In the context of SAT-based BMC, we proposed to use the proof of unsatisfiability of the BMC formula in order to discover over-approximating function summaries that are strong enough to guarantee the bounded safety. We further proposed an algorithm to effectively reuse the summaries synthesized after the verification of one program version to verify another program version. The core feature of the algorithm is that it revalidates the old summaries of the modified function calls locally and repairs the new summaries for functions whose old summaries are not valid any longer.

Furthermore, we contributed techniques for constructing function summaries of a better quality in cases when the program is supplied with a sequence of predefined assertions. We proposed the algorithm to perform a lightweight analysis of a set of assertions to optimize the generation of summaries. In addition, we contributed solutions for accelerating BMC for an individual program version and finding a proper unwinding of recursive calls.

We implemented the interpolation-based function summarization, refinement, automatic recursion depth detection and automatic assertion implication detection in a tool FunFrog extending the CBMC [40] model checker. We implemented the incremental BMC algorithm in a tool eVolCheck extending the FunFrog model checker. We evaluated the tools on the range of academic and industrial benchmarks provided by the PINCETTE EU project and confirmed that incremental changes can be verified efficiently for different classes of programs. More details on the evaluation can be found in Sect. 5.1 and Sect. 5.2.

Chapter 3

SMT-Based Simulation Discovery

Simulation is one of the oldest logical concepts behind program analysis. Introduced by Milner [104], a simulation relation is used to represent a condition under which the complete set of behaviors of one program (called *source* and denoted by S) is included into the set of behaviors of another program (called *target* and denoted by T). The role of simulation relation for FIV is self-explanatory: if S is simulated by T then all possible assertions that hold in T will also hold in S. Thus, an algorithm for discovering a simulation relation may be used for both types of programs: with implanted assertions (as in Chap. 2) or without any assertions at all.

The programs can, however, be substantially different, thus making the task of finding an appropriate simulation relation difficult. To overcome this problem, Milner suggests to abstract some irrelevant details from the target program and thus to improve the chances of the simulation relation to be found. In this chapter, we propose a solution to the problem known for the past half century, and in particular: (1) the challenge of constructing a total simulation relation between two programs, and (2) whenever the target T does not simulate the source S, the challenge of finding an abstraction of the target T that simulates the source S.

In Sect. 3.2, we propose to reduce the problem of simulation discovery to deciding validity of $\forall\exists$-formulas. Intuitively, the formulas say "for each behavior of S there exists a corresponding behavior of T". We manipulate implicit abstractions of T by introducing existential quantifiers to the right-hand-side of the $\forall\exists$-formulas. In Sect. 3.3, we present a novel algorithm AE-VAL for deciding validity of $\forall\exists$-formulas. In addition, AE-VAL extracts a Skolem relation to witness the existential quantifiers. This Skolem relation is the key to refine the considered abstractions of T.

The results reported in this chapter have been published in the paper [57] (co-authored with Arie Gurfinkel and Natasha Sharygina). For the sake of simplicity, this chapter considers loop-free programs only, but the contributed approach scales to programs with loops as well. The further use of simulation relation, as a building block for Property-Directed Equivalence applicable to programs with complicated loop structures, is discussed in Chap. 4 and in Sect. 5.3.

3.1 Background

3.1.1 SMT Solving and Quantifier Elimination

Given a set $\mathcal{X}$ of variables, a set $\mathcal{F}$ of *function symbols*, and a set $\mathcal{P}$ of *predicate symbols*. A function symbol of 0-arity is called a *constant*, and a predicate symbol of 0-arity corresponds to a propositional variable. A *term* is either a constant or an expression constructed from function symbols in $\mathcal{F}$ and variables in $\mathcal{X}$. An *atom* is either a propositional variable or an expression constructed from predicate symbols in $\mathcal{P}$ and terms. A formula is an expression constructed from atoms, propositional operations ($\neg$, $\wedge$, $\vee$), and quantifiers ($\forall$, $\exists$). A *ground* formula is a formula that has no variable occurrences. A formula without free variables is called a *sentence* (or a closed formula). A formula without quantifiers is called quantifier-free.

We refer to a union $\mathcal{X} \cup \mathcal{F}$ as a *signature* Σ. Consider a first-order language with equality (i.e., "="$\in \mathcal{P}$) and a signature Σ. A Σ*-structure* S consists of a *domain of interpretation*, denoted as $|S|$, and an *interpretation function* that assigns elements of $|S|$ to variables, and functions and predicates on $|S|$ to the symbols of Σ. Given a Σ-structure S, a Σ-assignment s is a function mapping each Σ-term to an element in $|S|$. Given a formula φ in the first-order language, we call φ satisfied by a Σ-structure S, denoted $S \models \varphi$, if there exists a Σ-assignment s (called a *model*) under which φ evaluates to $\top$.

A first-order theory $\mathcal{T}$ consists of a signature Σ and a set of Σ-sentences $\mathcal{S}$. A Σ-formula φ is $\mathcal{T}$-satisfiable is there exists a Σ-structure $S \in \mathcal{S}$ such that φ satisfied by S. A Σ-formula φ is $\mathcal{T}$-valid if its negation is $\mathcal{T}$-unsatisfiable. The Satisfiability Modulo Theory (SMT) problem for a given theory $\mathcal{T}$ and a quantifier-free formula φ aims at determining whether φ is $\mathcal{T}$-satisfiable.

Quantifier elimination is a decision procedure that turns a quantified formula into an equivalent quantifier-free formula. In addition, the quantifier elimination algorithms are often able to discover a Skolem function that represents witnesses for the existentially quantified individual variables (e.g., [11; 90; 77; 84]). Various tasks in verification and synthesis [128; 37; 17; 63] rely on efficient techniques to remove existential quantifiers from formulas in first-order logic, thus adjusting the task to be decided by an SMT solver. In particular, *functional synthesis* aims at computing a function that meets a given input/output relation. A function with an input x and an output y, specified by a relation $f(x,y)$, can be constructed as a by-product of deciding validity of the formula $\forall x \exists y \,.\, f(x,y)$. Due to a well-known *AE-paradigm* (also referred to as *Skolem paradigm* [114]), the formula $\forall x \exists y \,.\, f(x,y)$ is equivalent to the formula $\exists sk\ \forall x \,.\, f(x, sk(x))$, which means existence of a Skolem function sk, such that $f(x, sk(x))$ holds for every x. Thus the key feature in modern quantifier elimination approaches is their ability to produce witnessing Skolem function.

3.1.2 Model-Based Projection for Linear Rational Arithmetic

Quantifier elimination of a formula $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$ is an expensive procedure that typically proceeds by enumerating all models of an extended formula $T(\vec{x}, \vec{y})$. However, in some applications, the quantifier-free formula, fully equivalent to $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, is not even needed. Instead, it is enough to operate by (possibly incomplete) sets of models. This idea relies on some notion of projection that under-approximates existential quantification. In this chapter, we consider a concept of Model-Based Projections (MBP), recently proposed by [86; 52].

In the following, we use vector notation to denote sets of variables (and set-theoretic operators of *subset* $\vec{u} \subseteq \vec{x}$, *complement* $\vec{x}_{\vec{u}} = \vec{x} \setminus \vec{u}$, *union* $\vec{x} = \vec{u} \cup \vec{x}_{\vec{u}}$).

Definition 11. *An $MBP_{\vec{y}}$ is a function from models of $T(\vec{x}, \vec{y})$ to $\vec{y}$-free formulas iff:*

$$\text{if } m \models T(\vec{x}, \vec{y}) \text{ then } m \models MBP_{\vec{y}}(m, T) \tag{3.1}$$

$$MBP_{\vec{y}}(m, T) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y}) \tag{3.2}$$

There are finitely many MBPs for fixed $\vec{y}$ and T and different models $m_1, \ldots, m_n$ (for some n): $T_1(\vec{x}), \ldots, T_n(\vec{x})$, such that $\exists \vec{y} \,.\, T(\vec{x}, \vec{y}) = \bigvee_{i=1}^{n} T_i(\vec{x})$.

A possible way of implementing an MBP-algorithm was proposed in [86]. It is based on Loos-Weispfenning (LW) quantifier-elimination method [95] for Linear

Rational Arithmetic (LRA). Consider formula $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, where T is quantifier-free. In our simplified presentation, $\vec{y}$ is singleton, T is in Negation Normal Form (that allows the operator $\neg$ to be applied only to variables), and y appears in the literals only of the form $y = e$, $l < y$ or $y < u$, where l, u, e are y-free. LW states that the equation (3.3) holds:

$$\exists y \,.\, T(\vec{x}) \equiv \Big(\bigvee_{(y=e)\in lits(T)} T[e] \vee \bigvee_{(l<y)\in lits(T)} T[l+\epsilon] \vee T[-\infty] \Big) \tag{3.3}$$

In (3.3), *lits*(T) denote the set of literals of T, $T[\cdot]$ stands for a *virtual substitution* for the literals containing y. In particular, $T[e]$ substitutes exact values of y $(y = e)$, $T[l + \epsilon]$ substitutes the intervals $(l < y)$ of possible values of y, $T[-\infty]$ substitutes the rest of the literals. Consequently, a function *LRAProj*$_T$ is an implementation of the *MBP* function for (3.3):

$$LRAProj_T(m) = \begin{cases} T[e], & \text{if } (y = e) \in lits(T) \wedge m \models (y = e) \\ T[l+\epsilon], & \text{else if } (l < y) \in lits(T) \wedge m \models (l < y) \wedge \\ & \quad \forall (l' < y) \in lits(T) \,.\, m \models \big((l' < y) \implies (l' \leq l)\big) \\ T[-\infty], & \text{otherwise} \end{cases} \tag{3.4}$$

3.1.3 Programs and Abstractions

As in Sect. 3.1.2, we use vector notation to denote sets of *real and boolean* variables (and set-theoretic operations of *subset* $\vec{u} \subseteq \vec{x}$, *complement* $\vec{x}_{\vec{u}} = \vec{x} \setminus \vec{u}$, *union* $\vec{x} = \vec{u} \cup \vec{x}_{\vec{u}}$). For the first-order formulas $\varphi(\vec{x}) \in Expr$ in the chapter, we assume that all free variables $\vec{x}$ are implicitly universally quantified. For simplicity, we omit the arguments and simply write φ when the arguments are clear from the context. Furthermore, for a model m of $\varphi \in Expr$ we write $m \models \varphi$, and for an implication between $\varphi, \psi \in Expr$ we write $\varphi \implies \psi$.

In this chapter, we illustrate our key ideas on simulation synthesis for simple loop-free programs, since 1) it requires deepening to the computational level, but handling loops would make our presentation unnecessarily complicated; 2) it can be seen as a building block for designing an automated FIV solution for complex programs with (possibly nested) loops. Thus, the extension of the concepts and ideas presented here is left to Chap. 4. Throughout the chapter (but except Sect. 3.3), we use the same three example programs shown in Fig. 3.1. For

```
int a = *;
int b = *;
while(*){
  a = a + b;
}
```

(a) The source

```
int a = *;
int b = *;
while(*){
  int c = a - b;
  a = c;
}
```

(b) The target

```
int a = *;
while(*){
  int b = *;
  int c = a - b;
  a = c;
}
```

(c) Abstraction of the target

Figure 3.1. Three programs written in the C programming language.

demonstration purposes, we focus on their loop bodies in an arbitrary iteration.

Definition 12. *A* program *P is a tuple $\langle Var, Init, Tr\rangle$, where $Var \equiv V \cup L \cup V'$ is a set of* input, local *and* output variables*; $Init \in Expr$ encodes the* initial states *over V; and $Tr \in Expr$ encodes the* transition relation *over Var.*

A state $\vec{s} \in \mathcal{S}$ is a valuation to all variables in V. While Tr encodes an entire computation between initial and final states, the values of variables in L explicitly capture all intermediate states along the computation. If for states $\vec{s}, \vec{s'}$ there exists a valuation $\vec{l}$ to the local variables in L, such that $(\vec{s} \cup \vec{l} \cup \vec{s'}) \models Tr$, we call the pair $(\vec{s}, \vec{s'})$ *computable*. V' is used to denote the values of variables in V at the end of the computation. We write $\vec{s'}$ for $\vec{s}(x')$ and $\mathcal{S}'$ for $\{\vec{s'} \mid \vec{s} \in \mathcal{S}\}$.

Definition 13. *Given a program $P = \langle Var, Init, Tr\rangle$, a* transition system *$\mathcal{T}(P) = \langle \mathcal{S}, I, \mathcal{R}\rangle$, where $I = \{\vec{s} \in \mathcal{S} \mid \vec{s} \models Init\}$ is the set of* initial states, *$\mathcal{R} = \{(\vec{s}, \vec{s'}) \mid \vec{s} \in \mathcal{S},\ \vec{s'} \in \mathcal{S}' \,.\, (\vec{s}, \vec{s'})$ is computable$\}$ is a* transition relation.

To simplify the presentation, we use programs and their transition systems interchangeably throughout this chapter.

Definition 14. *Program $P_1 = \langle V_1 \cup L_1 \cup V', Init_1, Tr_1\rangle$ is an* abstraction *of program $P_2 = \langle V_2 \cup L_2 \cup V_2', Init_2, Tr_2\rangle$ iff* (1) *$V_1 \subseteq V_2$,* (2) *$Init_2 \implies Init_1$,* (3) *each $(\vec{s}, \vec{s'})$ that is computable in Tr_2, is also computable in Tr_1.*

Example 7. *Consider an example in Fig. 3.1(b)-3.1(c). The loop bodies of the concrete and abstract programs differ in the sets of input variables: for the abstract one $V_1 = \{a, b\}$, for the concrete one: $V_2 = \{a\}$; and the sets of local variables: $L_1 = \{c\}$, $L_2 = \{b, c\}$, respectively. The difference in the initial states and the transition relations of both programs can be seen in Examples 8,9.*

Definition 15. *Given transition systems S and T, a* left-total *relation $\rho \subseteq \mathcal{S}_S \times \mathcal{S}_T$ is a* simulation relation *if* (1) *every state in I_S is related by ρ to some state in I_T, and* (2) *for all states $\vec{s}$, $\vec{s}'$ and $\vec{t}$, such that $(\vec{s},\vec{t}) \in \rho$ and $(\vec{s},\vec{s}') \in \mathcal{R}_S$ there is some state $\vec{t}'$, such that $(\vec{t},\vec{t}') \in \mathcal{R}_T$ and $(\vec{s}',\vec{t}') \in \rho$.*

We write $S \preceq_\rho T$ to denote that the source S is simulated by the target T via a simulation relation ρ. We write $S \preceq T$ to indicate existence of a simulation between S and T. The identity relation *id*, i.e., pairwise-equivalence of values of common variables, is an example of ρ. Each program S is simulated by a *universal abstraction* $\mathbb{U}$ of any other program T (which is in fact the only common abstraction to all possible programs). Since such cases do not provide any practical significance, in our approach they are algorithmically disqualified.

Note that the programs in scope of the chapter are not required to have an *error* location. Thus, the approach proposed in the following sections is not limited to dealing only with safe programs.

3.2 From Simulation to Validity

The goal of simulation synthesis is to deliver a relation ρ between two programs S and T such that $S \preceq_\rho T$. The high-level idea behind a possible synthesizer of ρ is to enumerate a pool of candidate relations $\rho_1, \ldots, \rho_n$ and check whether one of those relation constitutes a simulation. While such "enumerate-check" procedure can be straightforwardly implemented in a loop, the efficiency and termination of the loop still crucially depends on two challenging points. First, there should be a procedure for checking each candidate relation. Second, there should be a procedure for choosing the next candidate relations to populate the pool.

In this section, we show that deciding whether a given relation ρ is a simulation relation is reducible to deciding validity of $\forall\exists$-formulas. We then show how Skolem functions witnessing the existential quantifiers can be used to provide a new candidate relation ρ' to be iteratively checked for simulation.

3.2.1 Deciding Simulation Symbolically

Let $S(\vec{s},\vec{x},\vec{s}')$, $T(\vec{t},\vec{y},\vec{t}') \in Expr$ encode transition relations of programs, where $\vec{s}$ and $\vec{t}$, $\vec{s}'$ and $\vec{t}'$, $\vec{x}$ and $\vec{y}$ are input, output, and local variables, respectively.

Let $Init_S(\vec{s})$, $Init_T(\vec{t}) \in Expr$ encode the initial states in S and T, respectively. Let $\rho(\vec{s}, \vec{t}) \in Expr$ encode a left-total relation between variables in S and T.

Lemma 4. *T simulates S via relation ρ iff*

$$Init_S(\vec{s}) \implies \exists \vec{t} \,.\, \rho(\vec{s}, \vec{t}) \land Init_T(\vec{t}) \tag{3.5}$$

$$\rho(\vec{s}, \vec{t}) \land \exists \vec{x} \,.\, S(\vec{s}, \vec{x}, \vec{s'}) \implies \exists \vec{t'}, \vec{y} \,.\, T(\vec{t}, \vec{y}, \vec{t'}) \land \rho(\vec{s'}, \vec{t'}) \tag{3.6}$$

Implication (3.5) reflects the matching of initial states in S and T via ρ. The left-hand-side of implication (3.6) reflects the set of all behaviors in S and the set of all input conditions matched via ρ. The right-hand-side of (3.6) reflects the existence of a behavior in T and an output condition matched via ρ.

Example 8. *Consider two programs in Fig. 3.1(a) and Fig. 3.1(b). Assume that constants X, Y are assigned to the input variables as in* (3.7)*, so the computation starts at the identical states. The fragments of the transition relation corresponding to the single loop body are encoded into* (3.8)*:*

$$Init_S \equiv (a_S = X) \land (b_S = Y) \qquad Init_T \equiv (a_T = X) \land (b_T = Y) \tag{3.7}$$

$$S \equiv (a'_S = a_S + b_S) \qquad T \equiv (c_T = a_T - b_T) \land (a'_T = c_T) \tag{3.8}$$

where the subscript indicates in which program the variables are defined.

Let ρ be a relation between variables in S and T:

$$\rho \equiv (a_S = a_T) \land (b_S = b_T) \qquad \rho' \equiv (a'_S = a'_T) \land (b_S = b_T) \tag{3.9}$$

ρ is a simulation relation iff the two formulas are valid:

$$Init_S \implies \exists a_T, b_T \,.\, Init_T \land \rho \qquad \rho \land S \implies \exists c_T, a'_T \,.\, T \land \rho' \tag{3.10}$$

Note that since T is deterministic, the existential quantifiers in (3.10) *are eliminated trivially by substitution. In our example, the left implication of* (3.10) *is valid, but the second implication of* (3.10) *simplifies to* $0 = 1$*. Hence, $S \not\preceq_\rho T$.*

3.2.2 Abstract Simulation

If the complete simulation relation between S and T is not found, we can proceed with checking whether S is simulated by an abstraction αT of T via relation ρ_α. As a key result, we show that such abstract-simulation checking can be done without constructing an abstraction explicitly. We focus on an existential abstraction $\alpha^{\exists}_{\vec{u}} T$ of T that abstracts away a subset of variables $\vec{u} \subseteq \vec{t}$ of T [42].

Definition 16. *$Init_{\alpha^{\exists}_{\vec{u}} T} \equiv \exists \vec{u} \,.\, Init_T(\vec{t})$, and $\alpha^{\exists}_{\vec{u}} T \equiv \exists \vec{u}, \vec{u}' \,.\, T(\vec{t}, \vec{y}, \vec{t'})$.*

Deciding whether $\alpha^{\exists}_{\vec{u}}T$ simulates S via $\rho_\alpha(\vec{s}, \vec{t}_{\vec{u}})$ (where $\vec{t}_{\vec{u}}$ is the complement of $\vec{u}$ in $\vec{t}$) can be done if the variables $\vec{u}$ are treated as locals in T.

Lemma 5. *$\alpha^{\exists}_{\vec{u}}T$ simulates S via relation ρ_α iff*

$$Init_S(\vec{s}) \implies \exists \vec{t}_{\vec{u}}, \vec{u} \,.\, \rho_\alpha(\vec{s}, \vec{t}_{\vec{u}}) \land Init_T(\vec{t}) \tag{3.11}$$

$$\rho_\alpha(\vec{s}, \vec{t}_{\vec{u}}) \land \exists \vec{x} \,.\, S(\vec{s}, \vec{x}, \vec{s'}) \implies \exists \vec{u}, \vec{y}, \vec{t'}_{\vec{u'}}, \vec{u}' \,.\, T(\vec{t}, \vec{y}, \vec{t'}) \land \rho_\alpha(\vec{s'}, \vec{t'}_{\vec{u'}}) \tag{3.12}$$

Recall that in Example 8, the loop body T was shown to not simulate the loop body S via identity relation. Interestingly, this result is still useful to obtain a simulation relation between S and T by creating an implicit abstraction of T and further refining it. We demonstrate this 2-steps procedure in Example 9.

Example 9. *As the first (*abstraction*) step, we create an abstraction of T by choosing a variable (say b) to be existentially quantified. Note that the produced abstraction is equivalent to the program in Fig. 3.1(c). Instead of encoding initial states $Init_{\alpha T}$ and a transition relation of αT from scratch (similarly to* (3.7) *and* (3.8)*), we let $Init_{\alpha T} \equiv \exists b_T \,.\, Init_T$ and $\alpha T \equiv \exists b_T \,.\, T$. Relation* (3.9) *(disproven to be a simulation between S and T) is weakened in correspondence with αT:*

$$\rho_\alpha \equiv (a_S = a_T) \qquad\qquad \rho'_\alpha \equiv (a'_S = a'_T) \tag{3.13}$$

ρ_α is a simulation relation between S and αT iff the following formulas are valid:

$$Init_S \implies \exists a_T, b_T \,.\, Init_T \land \rho_\alpha \qquad\qquad \rho_\alpha \land S \implies \exists c_T, a'_T, b_T \,.\, T \land \rho'_\alpha \tag{3.14}$$

Clearly, (3.14) *are valid iff there is a Skolem function for the existentially quantified variable b_T. Note that $sk_{b_T}(b_S) = -b_S$ is such function, and* (3.15) *are valid.*

$$\begin{aligned} Init_S \implies (b_T = -b_S) \implies \exists a_T \,.\, Init_T \land \rho_\alpha \\ \rho_\alpha \land S \implies (b_T = -b_S) \implies \exists c_T, a'_T \,.\, T \land \rho'_\alpha \end{aligned} \tag{3.15}$$

*As the second (*refinement*) step, sk_{b_T} is used to strengthen the simulation relation* (3.13) *between S and αT to become* (3.16).

$$\rho^{ext}_\alpha \equiv (a_S = a_T) \land (b_S = -b_T) \qquad \rho'^{ext}_\alpha \equiv (a'_S = a'_T) \land (b_S = -b_T) \tag{3.16}$$

Note that ρ^{ext}_α is a simulation relation between S and T.

3.2.3 Refining Simulation by Skolem Relations

Definition 17. *Given a formula $\exists y \,.\, f(x, y)$, a relation $Sk_y(x, y)$ is a* Skolem relation *for y iff* (1) $Sk_y(x, y) \implies f(x, y)$, (2) $\exists y \,.\, Sk_y(x, y) \iff \exists y \,.\, f(x, y)$.

In Def. 17, we allow Sk_y to be a relation between x and y such that (1) Sk_y maps each x to a value of y that makes f true, (2.1) if for a given x, Sk_y maps x

to some value of y then there is a value of y that makes f valid for this value of x, (2.2) if for a given x, there is a value of y such that f holds, then Sk_y is not empty. A Skolem relation Sk_y is *functional* iff it is of the form $Sk_y(x,y) \equiv y = f_y(x)$ (also known as a *Skolem function*, as in [127]). $Sk_{\vec{y}}$ is *Cartesian* iff it is a Cartesian product of Skolem relations of individual variables from $\vec{y}$. $Sk_{\vec{y}}$ is *guarded* iff it is a guarded disjunction of Cartesian Skolem relations.

In other words, validity of a $\forall\exists$-formula is equivalent to existence of an appropriate total Skolem relation. As sketched in Example 9, our use of a Skolem relation Sk witnessing the validity of the formulas (3.11,3.12) is to refine an abstract simulation relation ρ_α to $\rho_\alpha^{ext} = \rho_\alpha \wedge Sk$. However, ρ_α^{ext} is guaranteed to be a simulation relation only in case if the corresponding formulas (3.11,3.12) are valid, thus requiring an extra simulation check.

Theorem 3 formalizes the condition, under which a Skolem relation for valid formulas (3.11,3.12) refines an abstract simulation relation.

Theorem 3. *Let $S(\vec{s},\vec{x},\vec{s'})$ and $T(\vec{t},\vec{y},\vec{t'})$ be two programs, such that $S \preceq_\rho T$. Let $\vec{u}$ be a subset of variables in T ($i.e.,\vec{u} \subseteq \vec{t}$), such that $S \preceq_{\rho_\alpha} \alpha^{\exists}_{\vec{u}}(T)$ and $\rho \implies \rho_\alpha$. Then, there exists a relation $Sk(\vec{s},\vec{u})$ such that* (1) *$\rho_\alpha \wedge Sk$ is a simulation relation between S and T and* (2) *Sk is a Skolem relation for $\vec{u}$ in* (3.11) *and* (3.12).

Proof. Let $\vec{t}_{\vec{u}}$ be the complement of $\vec{u}$ in $\vec{t}$, and $\vec{t'}_{\vec{u}'}$ be the complement of $\vec{u}'$ in $\vec{t'}$. Having in mind $\rho \implies \rho_\alpha$, let Sk be a left-total relation over $\vec{s}$, $\vec{s'}$, $\vec{u}$ and $\vec{u}'$, such that:

$$\begin{aligned} \rho(\vec{s},\vec{t}) &\equiv \rho_\alpha(\vec{s},\vec{t}_{\vec{u}}) \wedge Sk(\vec{s},\vec{u}) \\ \rho(\vec{s'},\vec{t'}) &\equiv \rho_\alpha(\vec{s'},\vec{t'}_{\vec{u}}) \wedge Sk(\vec{s'},\vec{u}') \end{aligned} \tag{3.17}$$

Substituting (3.17) into (3.5) and (3.6), we get (3.18) and (3.19) respectively:

$$Init_S(\vec{s}) \implies \exists \vec{t}_{\vec{u}}, \vec{u} \,.\, \rho_\alpha(\vec{s},\vec{t}_{\vec{u}}) \wedge Sk(\vec{s},\vec{u}) \wedge Init_T(\vec{t}) \tag{3.18}$$

$$\begin{aligned} &\rho_\alpha(\vec{s},\vec{t}_{\vec{u}}) \wedge Sk(\vec{s},\vec{u}) \wedge S(\vec{s},\vec{s'}) \implies \\ &\qquad \exists \vec{t'}_{\vec{u}}, \vec{u}', \vec{y} \,.\, T(\vec{t}_{\vec{u}}, \vec{u}, \vec{y}, \vec{t'}_{\vec{u}'}, \vec{u}') \wedge \rho_\alpha(\vec{s'},\vec{t'}_{\vec{u}'}) \wedge Sk(\vec{s'},\vec{u}') \end{aligned} \tag{3.19}$$

Then from (3.18) and (3.19) it easy to see that $Sk(\vec{s},\vec{u})$ satisfies Def. 17 (it is a Skolem relation) for $\vec{u}$, respectively in (3.20) and in (3.21).

$$Init_S(\vec{s}) \implies \exists \vec{t}_{\vec{u}}, \vec{u} \,.\, \rho_\alpha(\vec{s},\vec{t}_{\vec{u}}) \wedge Sk(\vec{s},\vec{u}) \wedge Init_T(\vec{t}) \tag{3.20}$$

$$\begin{aligned} &\exists \vec{u} \,.\, \rho_\alpha(\vec{s},\vec{t}_{\vec{u}}) \wedge S(\vec{s},\vec{s'}) \implies \\ &\qquad \exists \vec{t'}_{\vec{u}}, \vec{u}', \vec{y} \,.\, T(\vec{t}_{\vec{u}}, \vec{u}, \vec{y}, \vec{t'}_{\vec{u}'}, \vec{u}') \wedge \rho_\alpha(\vec{s'},\vec{t'}_{\vec{u}'}) \wedge Sk(\vec{s'},\vec{u}') \end{aligned} \tag{3.21}$$

Consequently, $Sk(\vec{s},\vec{u})$ is a Skolem relation for $\vec{u}$ in (3.23), that is logically weaker than (3.21); and in (3.22), that is logically weaker than (3.20).

$$Init_S(\vec{s}) \implies \exists \vec{t}_{\vec{u}}, \vec{u} \,.\, \rho_\alpha(\vec{s}, \vec{t}_{\vec{u}}) \wedge Init_T(\vec{t}) \tag{3.22}$$

$$\exists \vec{u} \,.\, \rho_\alpha(\vec{s}, \vec{t}_{\vec{u}}) \wedge S(\vec{s}, \vec{s}') \implies \exists \vec{t}'_{\vec{u}'}, \vec{u}, \vec{u}', \vec{y} \,.\, T(\vec{t}_{\vec{u}}, \vec{u}, \vec{y}, \vec{t}'_{\vec{u}}, \vec{u}') \wedge \rho_\alpha(\vec{s}', \vec{t}'_{\vec{u}}) \tag{3.23}$$

Finally, (3.22) and (3.23) are equivalent respectively to the simulation-checking-formulas (3.11) and (3.12) for S and $\alpha^{\exists}_{\vec{u}}(T)$. □

The automated discovery of simulation for realistic programs based on the aforementioned ideas and its application to FIV is left for Sect. 4.2.2 and Sect. 4.3.2 respectively. In the next section, we focus on the main solving routine that makes our FIV solutions possible.

3.3 Validity and Skolem Extraction

We present AE-VAL, a novel algorithm for deciding validity of ∀∃-formulas and constructing witnessing Skolem relations. Without loss of generality, we restrict the input formula to have the form $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, where S has no universal quantifiers, and T is quantifier-free.

3.3.1 Deciding Validity of ∀∃-Formulas

Our algorithm is based on a notion of Model-Based Projection (MBP), defined in Def. 11. Recall that an $MBP_{\vec{y}}$ is a function from models of $T(\vec{x}, \vec{y})$ to $\vec{y}$-free formulas. Given a model m, it returns a $\vec{y}$-free formula, such that (1) m is also a model of the image $MBP_{\vec{y}}(m, T)$, and (2) $MBP_{\vec{y}}(m, T)$ is an under-approximation of $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$. There are finitely many MBPs for fixed $\vec{y}$ and T and different models $m_1, \ldots, m_n$ (for some n): $T_1(\vec{x}), \ldots, T_n(\vec{x})$, such that $\exists \vec{y} \,.\, T(\vec{x}, \vec{y}) = \bigvee_{i=1}^{n} T_i(\vec{x})$.

Additionally, we assume that for each projection T_i, the MBP-algorithm gives a condition ϕ_i under which T is equisatisfiable with T_i:

$$\phi_i(\vec{x}, \vec{y}) \implies \big(T_i(\vec{x}) \iff T(\vec{x}, \vec{y})\big) \tag{3.24}$$

Such a relation ϕ_i is a natural by-product of the MBP-algorithm in [86]. Intuitively, each ϕ_i captures the substitutions made in T to produce T_i. We assume that each ϕ_i is in the Cartesian form, i.e., a conjunction of terms, in which each $y \in \vec{y}$ appears at most once. That is, for $y \in \vec{y}$ and $\sim \in \{<, \leq, =, \geq, >\}$,

$$\phi_i(\vec{x}, \vec{y}) = \bigwedge_{y \in \vec{y}} \big(y \sim f_y(\vec{x})\big) \tag{3.25}$$

Algorithm 6: AE-VAL $\left(S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x},\vec{y})\right)$

```
Input: $S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x},\vec{y})$
Output: return value $\in$ {VALID, INVALID } of $S(\vec{x}) \Longrightarrow \exists \vec{y} \,.\, T(\vec{x},\vec{y})$
Data: SMTSOLVER, counter $i$, models $\{m_i\}$, MBPs $\{T_i(\vec{x})\}$, conditions
      $\{\phi_i(\vec{x},\vec{y})\}$
1  SMTADD($S(\vec{x})$);
2  $i \leftarrow 0$;
3  forever do
4      $i$++;
5      if (ISUNSAT(SMTSOLVE())) then return VALID;
6      SMTPUSH();
7      SMTADD($T(\vec{x},\vec{y})$);
8      if (ISUNSAT(SMTSOLVE())) then return INVALID;
9      $m_i \leftarrow$ SMTGETMODEL();
10     $(T_i, \phi_i(\vec{x},\vec{y})) \leftarrow$ GETMBP($\vec{y}, m_i, T(\vec{x},\vec{y})$));
11     SMTPOP();
12     SMTADD($\neg T_i$);
```

$$
\begin{aligned}
S &\equiv (a = b + 2)\\
T &\equiv (a' > a) \wedge (b = 1 \implies b' = b) \wedge\\
&\quad (b = 2 \implies b' > b) \wedge\\
&\quad (b = 3 \implies b' < b)\\
m_1 &\equiv \{a = 0, b = -2, a' = {}^1/_2, b' = -{}^5/_2\}\\
m_2 &\equiv \{a = 4, b = 2, a' = {}^9/_2, b' = {}^5/_2\}\\
m_3 &\equiv \{a = 3, b = 1, a' = {}^7/_2, b' = 1\}
\end{aligned}
$$

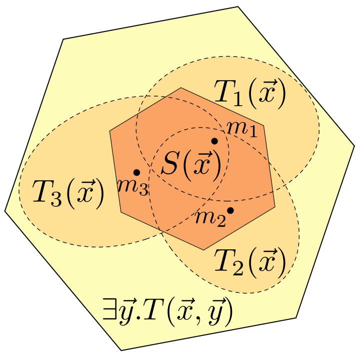


Figure 3.2. (a) S and T for $\vec{x} \equiv \{a, b\}$, $\vec{y} \equiv \{a', b'\}$, models of $S \wedge T$ and (b) the correspondent Venn diagram.

We write $(T_i, \phi_i) \leftarrow$ GETMBP$(\vec{y}, m_i, T(\vec{x},\vec{y}))$ for the invocation of the MBP-algorithm that takes a formula T, a model m_i of T and a vector of variables $\vec{y}$, and returns a projection T_i of T based on m_i and the corresponding relation ϕ_i.

AE-VAL is shown in Alg. 6. Given formulas $S(\vec{x})$ and $\exists \vec{y} \,.\, T(\vec{x},\vec{y})$, it decides validity of $S(\vec{x}) \implies \exists \vec{y}.T(\vec{x},\vec{y})$. AE-VAL enumerates the models of $S \wedge T$ and blocks them from S. In each iteration i, it first checks whether S is non-empty (line 3) and then looks for a model m_i of $S \wedge T$ (line 9). If m_i is found, AE-VAL gets a projection T_i of T based on m_i (line 10) and blocks all models contained in T_i from S (line 12). The algorithm iterates until either it finds a model of S that can not be extended to a model of T (line 8), or all models of S are blocked (line 5). In the first case, the input formula is invalid. In the second case, every model of S has been extended to some model of T, and the formula is valid.

Three possible iterations of AE-VAL are depicted graphically in Fig. 3.2. In the first iteration, AE-VAL selects a model m_1 and generalizes it to a projection $MBP_{\vec{y}}(m_1, T) = T_1$. Then, it picks a model m_2 that is not contained in T_1 and generalizes it to $MBP_{\vec{y}}(m_2, T) = T_2$. Finally, it picks a model m_3 that is contained neither in T_1 nor in T_2, and generalizes it to $MBP_{\vec{y}}(m_3, T) = T_3$. At this point, all models of S are covered by $\vec{y}$-free implicants of $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, and the algorithm terminates. We demonstrate this further in the following example.

Example 10. *Let S and T be as defined in Fig. 3.2. We use Φ_i to denote the formula in the SMT context at the beginning of iteration i of* AE-VAL. *Initially, $\Phi_1 = S$. The first model is m_1, and* GETMBP$(\vec{y}, m_1, T)$ *returns:*

$$T_1 \equiv (b \neq 1) \wedge (b \neq 2) \qquad \phi_1 \equiv (a' > a) \wedge (b' < b)$$

In the iteration 2, $\Phi_2 = \Phi_1 \wedge \neg T_1$, GETMBP$(\vec{y}, m_2, T)$ *returns:*

$$T_2 \equiv (b \neq 1) \wedge (b \neq 3) \qquad \phi_2 \equiv (a' > a) \wedge (b' > b)$$

In the iteration 3, $\Phi_3 = \Phi_2 \wedge \neg T_2$, GETMBP$(\vec{y}, m_3, T)$ *returns:*

$$T_3 \equiv (b \neq 2) \wedge (b \neq 3) \qquad \phi_3 \equiv (a' > a) \wedge (b' = b)$$

In the iteration 4, $\Phi_4 = \Phi_3 \wedge \neg T_3$ is unsatisfiable, and consequently AE-VAL *returns* VALID *and terminates.*

AE-VAL is similar to other algorithms for deciding validity of quantified formulas presented in the earlier works [105; 113; 52; 118]. However, it is the only one known to generate a fine-grained Skolem relation, on which we elaborate in the following Sect. 3.3.2.

3.3.2 Extracting Skolem Relation

AE-VAL is designed to construct a Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$, that maps each model of $S(\vec{x})$ to a corresponding model of $T(\vec{x}, \vec{y})$. We use a set of projections $\{T_i(\vec{x})\}$ for $T(\vec{x}, \vec{y})$ and a set of conditions $\{\phi_i(\vec{x}, \vec{y})\}$ that make the corresponding projections equisatisfiable with $T(\vec{x}, \vec{y})$.

Lemma 6. *For each i, the relation $\phi_i(\vec{x}, \vec{y})$ is a Skolem relation for $\vec{y}$ in formula $S(\vec{x}) \wedge T_i(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$.*

Proof. The proof follows from (3.2), (3.24), and (3.25). Recall, by definition (3.24), $\phi_i(\vec{x}, \vec{y}) \implies \big(T_i(\vec{x}) \iff T(\vec{x}, \vec{y})\big)$. Unfolding Def. 17, we need to show that:
1) $\phi_i(\vec{x}, \vec{y}) \implies (S(\vec{x}) \wedge T_i(\vec{x}) \implies T(\vec{x}, \vec{y}))$ – by definition (3.24).
2) $\exists y \,.\, \phi_i(\vec{x}, \vec{y}) \iff (S(\vec{x}) \wedge T_i(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y}))$ – since by construction (3.25), ϕ_i is total relation, and an implication in the definition of MBP (3.2) is valid. □

Intuitively, ϕ_i maps each model of $S \wedge T_i$ to a model of T. Thus, in order to define the guarded Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ it is enough to match each ϕ_i against the corresponding T_i, as proposed in (3.26).

$$Sk_{\vec{y}}(\vec{x}, \vec{y}) \equiv \begin{cases} \phi_1(\vec{x}, \vec{y}) & \text{if } T_1(\vec{x}) \\ \phi_2(\vec{x}, \vec{y}) & \text{else if } T_2(\vec{x}) \\ \cdots & \text{else } \cdots \\ \phi_n(\vec{x}, \vec{y}) & \text{else } T_n(\vec{x}) \end{cases} \tag{3.26}$$

The following theorem states that $Sk_{\vec{y}}(\vec{x}, \vec{y})$ satisfies Def. 17 for the chosen model of $\vec{x}$, and $Sk_{\vec{y}}(\vec{x}, \vec{y})$ is defined for all models of $\vec{x}$. It follows immediately from Lemma 6.

Theorem 4. *If $Sk_{\vec{y}}(\vec{x}, \vec{y})$ is defined in (3.26) then $Sk_{\vec{y}}(\vec{x}, \vec{y})$ is a Skolem relation for $\vec{y}$ in formula $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$.*

Note that not all Skolem relations are equal. In practice, we often like a Skolem relation that minimizes the number of variables on which each partition depends. We leave the problem of finding the minimum partitioning for future and elaborate on this question in Sect. 4.2.3 of the next chapter.

3.3.3 Towards Minimal Skolem Refinement

In this section we elaborate on extracting Skolem functions from the Skolem relation returned by AE-VAL. Given $Sk_{\vec{y}}(\vec{x}, \vec{y})$ for $\vec{y}$, we want to factor $Sk_{\vec{y}}(\vec{x}, \vec{y})$ into the product of Skolem functions for individual variables $y_j \in \vec{y}$, for $0 < i \leq n$. This is easy whenever $Sk_{\vec{y}}(\vec{x}, \vec{y})$ can be transformed into the form $\bigwedge_i (y_j = f_i(\vec{x}))$. However, such a factoring of $Sk_{\vec{y}}(\vec{x}, \vec{y})$ is not feasible in general because

$Sk_{\vec{y}}(\vec{x},\vec{y})$ might contain interdependencies between variables in $\vec{y}$. Our goal is then to find a sequence of Skolem functions $f_1(\vec{x}),\ldots,f_n(\vec{x})$, such that every tuple $\langle\vec{x},f_1(\vec{x}),\ldots,f_n(\vec{x})\rangle$ is in $Sk_{\vec{y}}(\vec{x},\vec{y})$. Hence, the conjunction $sk_{\vec{y}}(\vec{x},\vec{y}) \equiv \big(\bigwedge_{0<i\leq n}(y_j = f_i(\vec{x}))\big)$ is a Skolem relation for $\vec{y}$, and $sk_{\vec{y}}(\vec{x},\vec{y}) \implies Sk_{\vec{y}}(\vec{x},\vec{y})$. We say that $sk_{\vec{y}}(\vec{x},\vec{y})$ *refines* $Sk_{\vec{y}}(\vec{x},\vec{y})$.

In practice, constructing a refinement $sk_{\vec{y}}(\vec{x},\vec{y})$ for (3.26) is still difficult. Instead, it is worth constructing a *partial* refinement – not necessarily a product of Skolem functions. If for some $v \in \vec{y}$, for every local Skolem relation $\phi_i(\vec{x},\vec{y})$, there exists a partial local Skolem refinement $\phi_i^v(\vec{x},v)$:

$$\phi_i^v(\vec{x},\vec{y}) \equiv (v = f_i(\vec{x})) \wedge \phi_i^{\vec{y}\setminus v}(\vec{x},\vec{y}\setminus v) \quad \text{s.t.} \quad \phi_i^v(\vec{x},\vec{y}) \implies \phi_i(\vec{x},\vec{y}) \tag{3.27}$$

then there exists a partial global Skolem refinement $Sk_{\vec{y}}^v(\vec{x},\vec{y})$ for v:

$$Sk_{\vec{y}}^v(\vec{x},\vec{y}) \equiv Sk_v(\vec{x},v) \wedge Sk_{\vec{y}\setminus v}(\vec{x},\vec{y}\setminus v) \quad \text{s.t.} \quad Sk_{\vec{y}}^v(\vec{x},\vec{y}) \implies Sk_{\vec{y}}(\vec{x},\vec{y}) \tag{3.28}$$

In the rest of the section, we propose techniques for (a) maximizing the number of factors in each local Skolem refinement (3.27), and (b) flattening the factors in the global Skolem refinement (3.28). Our techniques lead to significant minimization of the Skolem relation independently of whether a complete refinement of the Skolem relation into the product of Skolem functions was constructed.

We propose an algorithm GLOBALFACTOR (shown in Alg. 7) to re-factor the global Skolem refinement for a given variable $y_j \in \vec{y}$. The algorithm uses the procedure LOCALFACTOR to extract factors from partial local Skolem refinements (line 2). GLOBALFACTOR is independent of a particular implementation of LOCALFACTOR. We show a simple implementation of LOCALFACTOR in Alg. 8. Given a local Skolem $\phi(\vec{x},\vec{y})$ and a variable y_j, Alg. 8 returns a local Skolem function $y_j = f(\vec{x})$. Note that Alg. 7 fails (line 3) whenever Alg. 8 does so (which in turn depends on other subroutines of Alg. 8).

Whenever LOCALFACTOR returns a set of local Skolem functions $\{f_i(\vec{x})\}$ for each local Skolem $\phi_i(\vec{x},\vec{y})$, GLOBALFACTOR constructs the set of equivalence classes: $T_i(\vec{x})$ and $T_j(\vec{x})$ belong to the same equivalence class if $f_i(\vec{x}) = f_j(\vec{x})$. Then it is enough to collapse the *ite* statement (3.26) representing the global Skolem relation (line 7) by merging all the cases projections T_i belonging to the same equivalence class (line 6).

Note that if Alg. 7 is applied to every variable $y_j \in \vec{y}$ and returns a non-empty factor $Sk_{y_j}(\vec{x},y_j)$ (i.e., does not exit in line 3), then the product $\bigwedge_{y_j\in\vec{y}}(Sk_{y_j}(\vec{x},y_j))$ is factored partial global Skolem refinement for $\vec{y}$.

```
Algorithm 7: GlobalFactor($y_j, S(\vec{x}), \{(T_i(\vec{x}), \phi_i(\vec{x}, \vec{y}))\}$)
  Input: variable $y_j$, formula $S(\vec{x})$, set of pairs $\{(T_i(\vec{x}), \phi_i(\vec{x}, \vec{y}))\}$
  Output: factor $Sk_{y_j}(\vec{x}, y_j)$
  Data: partitions $\{P_k(\vec{x})\}$, equivalence classes $Df$
1 forall $(\phi_i(\vec{x}, \vec{y}))$ do
2     $(y_j = f_i(\vec{x})) \leftarrow$ LocalFactor($y_j, \phi_i(\vec{x}, \vec{y})$);
3     if $(f_i(\vec{x}) \in \varnothing)$ then return $\varnothing$ ;
4 $Df \leftarrow \{f_j\}/\equiv$; $n \leftarrow |Df|$;
5 for $(k = 1;\ k \leq n;\ k++)$ do
6     $P_k \leftarrow \bigvee_{\{i \mid f_i = [Df_k]\}}(T_i(\vec{x}))$;
7 return GlobalSkolem($\{(P_k(\vec{x}), [Df_k]\}$);
```

```
Algorithm 8: LocalFactor($y_j, \phi(\vec{x}, \vec{y})$)
  Input: $y_j \in \vec{y}$, local Skolem relation
         $\phi(\vec{x}, \vec{y}) = \bigwedge_{y_j \in \vec{y}}(\psi_{y_j}(\vec{x}, y_j, \ldots, y_n))$
  Output: factor of the local Skolem refinement $y_j = f_j(\vec{x})$
  Data: known functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$

1 $\pi_{y_j}(\vec{x}, y_j) \leftarrow$ Substitute($\psi_{y_j}(\vec{x}, y_j, y_{j+1}, \ldots, y_n), f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$);
2 if $(\pi_{y_j}(\vec{x}, y_j) \in \varnothing)$ then
3     return $\varnothing$;
4 $\bigwedge_k \big(y_j \sim f_j^k(\vec{x})\big) \leftarrow$ Rewrite($\pi_{y_j}(\vec{x}, y_j)$);
5 return MinimalSkolem$\Big(\bigwedge_k \big(y_j \sim f_j^k(\vec{x})\big)\Big)$;
```

Example 11. *Let $\vec{x} \equiv \{a, b\}$, $\vec{y} \equiv \{c, d\}$, $S \equiv (a > b)$, and $T \equiv (d > -1) \land \big(a \geq 0 \implies (c \leq 0 \land d \leq c)\big) \land \big(a < 0 \implies (c < b \land d = 0)\big)$.* AE-VAL *proves validity of $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$ in 2 iterations, in which it produces 2 pairs of MBPs and the local Skolem relations $\{(T_1, \phi_1), (T_2, \phi_2)\}$:*

$$T_1 \equiv (a < 0) \qquad \phi_1 \equiv (d = 0) \land (c \leq a) \land (c < b) \land (c < 0)$$
$$T_2 \equiv (a \geq 0) \lor (b > 0) \qquad \phi_2 \equiv (d = 0) \land (c \geq 0)$$

The global factor for d is a constant Skolem function: $Sk_d(\vec{x}, d) \equiv (d = 0)$. The global factor for c is in terms of the ternary ite-operator:
*$Sk_c(\vec{x}, c) \equiv$ **if** $(a < 0)$ **then** $\big((c \leq a) \land (c < b) \land (c < 0)\big)$ **else** $(c \geq 0)$.* □

Let us now discuss the algorithm LocalFactor. By construction, each local

Skolem relation $\phi(\vec{x},\vec{y})$ has a form $\bigwedge_{y_j \in \vec{y}}(\psi_{y_j}(\vec{x}, y_j, \ldots, y_n))$. Since quantifier elimination in AE-VAL is applied iteratively for each variable $y_j \in \vec{y}$, y_j may depend on the variables of $y_{j+1}, \ldots, y_n$ that are still not eliminated in the current iteration j. Each $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n)$ is the conjunction $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n) = \bigwedge_i (cl_i(\vec{x}, y_j, \ldots, y_n))$, where each cl_i is an (in)equality.

For each $y_j \in \vec{y}$, our goal is to find a Skolem function $f_{y_j}(\vec{x})$, such that $(y_j = f_{y_j}(\vec{x})) \implies \exists y_{j+1}, \ldots, y_n \,.\, \psi_{y_j}(\vec{x}, y_j, \ldots, y_n)$. The idea is presented in Alg. 8. The algorithm is applied separately for each $y_j \in \vec{y}$, starting from y_n till y_1. For each y_j, assume, we already established Skolem functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ for variables $y_{j+1}, \ldots, y_n$ in the previous runs of the algorithm.

First, the algorithm substitutes each appearance of variables $y_{j+1}, \ldots, y_n$ in ψ_{y_j} by $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ (line 1). If for some variable there is no Skolem function to substitute, the algorithm halts with nothing (line 3). Second, the algorithm normalizes $\pi_{y_j}(\vec{x}, y_j)$ into the form $\bigwedge_k \big(y_j \sim f_k(\vec{x})\big)$, i.e., conjunction of expressions, left-hand-sides of which are reserved for y_j and $\sim \in \{<, \leq, =, \geq, >\}$. For this, it uses the method REWRITE (line 4) that rewrites each clause using the following rule (where g, h - are functions over $\vec{x}$, p, q - rational numbers, sgn - a function, returning the sign of the rational number):

$$\Big((g(\vec{x}) + p \times y_j) \sim (h(\vec{x}) + q \times y_j)\Big) \implies \Big((sgn(p-q) \times y_j) \sim \big(-\frac{g(\vec{x})}{|p-q|} + \frac{h(\vec{x})}{|p-q|}\big)\Big)$$

Finally the algorithm gets rid of inequalities (whenever applicable). Method MINIMALSKOLEM rewrites each clause using the following rule:

$$(y_j \leq g(\vec{x}) \land y_j \geq g(\vec{x})) \implies (y_j = g(\vec{x}))$$
$$(y_j \leq h(\vec{x}) \land -y_j \leq -h(\vec{x})) \implies (y_j = h(\vec{x}))$$

Example 12. *Let $\vec{x} = \{a\}$, $\vec{y} = \{c, d\}$, and $\phi(\vec{x},\vec{y}) \equiv (0 \leq -c - 1 + d) \land (d = a) \land (-c + d \leq 1)$. First,* LOCALFACTOR *is run for d and returns its local Skolem factor $d = a$. Second,* LOCALFACTOR *is applied for c. Method* SUBSTITUTE *rewrites $(0 \leq -c - 1 + d) \land (-c + d \leq 1)$ with $d = a$ and gets $(0 \leq -c - 1 + a) \land (-c + a \leq 1)$. Then, method* REWRITE *rewrites it to $(c \leq -1 + a) \land (-c \leq 1 - a)$; and* MINIMALSKOLEM *produces the local Skolem factor for c: $(c = -1 + a)$.* □

Note that our current implementation of MINIMALSKOLEM extracts Skolem function only in a case when there is a pair of adjacent inequalities. This is sound, since for all $\vec{x}$, and for all total functions $g(\vec{x})$, $(y_j \leq g(\vec{x})) \land (y_j \geq g(\vec{x})) \iff (y_j = g(\vec{x}))$. In future work, we plan to support extracting Skolem functions from sets of arbitrary inequalities. For example, if $\psi_c(a,b,c) = \big((c \leq a) \land (c < b) \land (c < 0)\big)$, as in Example 11, then there might be many possible local Skolem factors for c, including $c = min(a,b,0) - 1$. Another example, if $\psi_d(a,b,d) = \big((d > a) \land (d < b) \land (a < b)\big)$, then a local Skolem factor for d might be $d = \frac{b-a}{2}$.

3.4 Related Work to Simulation Synthesis

The notion of simulation relation between programs dates back to the seminal paper of Milner [104]. Apart from the first algebraic formalization of simulation, this work also proposed an idea of abstracting some details from two programs in order to prove that they realize the same algorithm. Since then, this concept has been widely used in verification and other areas of computer science.

The first symbolic automatic construction of simulation relations was proposed by Dill et al. in [50]. However, that work is based on BDDs, and for quantifier elimination it uses Shannon Expansion [30] (i.e., replaces the quantification by a disjunction of all possible assignments). Another approach to check simulation relations is *game-theoretic*: the state space of the source and the target programs is traversed by the evader and the pursuer players. For instance, Henzinger et al. [73] apply it to prove validity of a simulation relation between infinite graphs. We target to solve this problem by exploiting recent advancements in SMT and thus allowing synthesis of non-trivial simulation relations.

The problem of constructing and checking simulation relations arises when there is a need to prove equivalence between two programs for more detailed overview see Sect. 4.4). Necula [109] proposes to check correctness of compiler optimizations by constructing simulation relations heuristically. Namjoshi et al. in [108; 64] propose a more precise way to construct simulation relations, which requires augmenting a particular optimizer. Ciobâcă et al. [39] develop a parametric proof system for proving mutual simulation between programs written in different programming languages. These approaches do not deal with cases when programs are not equivalent (i.e., there exists only an abstract simulation or when there is some other form of simulation relation rather than identity). Our approach goes beyond these limitations.

Simulation relation is a sort of relative specifications: it describes how the behaviors of programs relate to each other, but not how they behave individually. Inferring other types of relative specifications were studied in [92; 61]. Lahiri et al. [92] propose to search for *differential errors*: whether there exist two behaviors of S and T starting from the same input, such that the former is non-failing and the latter is failing. The proposed solution is by composing S and T into one program and running an off-the-shelf invariant generator on it. Felsing et.al [61] aims at synthesizing *coupling predicates*, a stronger relationship than a simulation relation, since it does not allow programs to have unmatched behaviors. In contrast to our approach, their synthesis method is restricted to deal only with deterministic and terminating programs and does not require quantifier elimination.

Functional synthesis [24] is the process to construct a logical function (which in turn can formalize some program) that fulfils a given specification. Counter-Example-Guided Inductive Synthesis (CEGIS) [128] maintains a pool of candidate functions and checks whether the current candidate is a solution to the synthesis problem. Our approach for simulation relation synthesis can be seen as an instantiation of CEGIS, but it simultaneously handles two candidate pools, for simulation and for abstraction. While solving a simulation-relation-checking formula, our algorithm decides whether the current *candidate relation* constitutes a simulation with respect to the current *candidate abstraction*. If the formula is valid, the pools are populated with the help of discovered Skolem relation; otherwise, by applying existential abstraction function. We leave the further discussion on possible extensions of this idea (i.e., to provide an iterative synthesis algorithm) for the next chapter.

3.5 Summary of Contributions

In this chapter, we contributed a solution to the problem of simulation discovery between two programs (central branch in Fig. 1.1) that do not necessarily contain assertions. Our main contribution is the approach to synthesize both, abstractions and simulations, between the source and the target programs. If the target does not simulate the source, we proposed to detect an abstraction of the target that simulates the source. In contrast to existing techniques, our solution is based on deciding validity of $\forall\exists$-formulas iteratively, and abstractions are created implicitly, by existential quantification of program variables.

The second contribution of the chapter is AE-VAL, the decision procedure that extracts Skolem relations from valid $\forall\exists$-formulas in LRA. We proposed to guide Skolemization procedure in AE-VAL by iterative construction of MBPs that under-approximate existential quantification. Finally, we presented an algorithm to factorize Skolem relations and make it more suitable for the further use. We implemented AE-VAL on the top of the SMT solver Z3 [106]. AE-VAL relies on the external procedure [86] to obtain MBPs.

Despite this chapter contributes the algorithm for simulation synthesis only for the loop-free programs, it can be generalized to overcome this restriction (and is discussed in Chap. 4). Evaluation of AE-VAL and the algorithm for simulation synthesis is performed within a NIAGARA framework that is discussed in Sect. 5.3.

Chapter 4

SMT-Based Unbounded Model Checking via Abstract Simulation

Model checking of programs handling unbounded loops is nowadays reduced to finding safe inductive invariants (or shortly, proofs) that over-approximate the sets of reachable states, but precise enough to prove unreachability of the given assertion. This problem is known to be undecidable in general, so the individual model-checking solutions based on CEGAR and PDR are not guaranteed to deliver an appropriate invariant. On the other hand, in cases if the model checking succeeded, the synthesized invariant provides an important specification that comes in handy whenever the program gets modified. However, in order to migrate the invariant between evolution boundaries, some relational specification, such as the simulation relation, is needed.

This chapter extends our idea of simulation synthesis presented in Chap. 3 to a general case. We consider programs with complicated loop structures, i.e., those that involve communication of two or more loop-free fragments encoded as independent transition relations. We propose to discover simulation relations gradually: 1) on the level of loop structures, (to find a matching between loop-free fragments), and 2) on the level of pairs of the matched loop-free fragments. This procedure is inductive since each discovered relation of each matched loop-free fragments requires checking for the compatibility with the other pairs of matched loop-free fragments. In Sect. 4.2, we propose an algorithm that implements a complete *Simulation-Abstraction-Refinement Loop* and enables such inductive reasoning.

In Sect. 4.3, we address a challenge of migrating a safe inductive invariant between programs. However, for the efficiency reasons, a precise simulation relation between concrete programs might be sacrificed here. Instead, it might be enough to find some abstraction of the already verified program that simulates the new program and to prove that the given invariant is safe for this abstraction. We propose to derive such abstractions from the invariants. If the abstract simulation is found then the proof can be lifted directly. Finally, we present a technique to lift the proof through abstractions even if they do not preserve safety. Our solution tries to lift as much information from the invariant as possible, and then strengthens it using an induction-based unbounded model checker.

The results reported in this chapter have been published in the following papers: [55] (co-authored with Arie Gurfinkel and Natasha Sharygina) and [57] (co-authored with Arie Gurfinkel and Natasha Sharygina). All the presented algorithms are implemented within an LLVM-based framework NIAGARA, but are discussed outside of this chapter, in Sect. 5.3.

4.1 Background

4.1.1 Large-Block Encoding for Unbounded Model Checking

In this chapter, we consider "large-block" encoding (LBE) [18] of programs that allows representing complex control-flow graphs compactly.

Definition 18. *A* program *is a tuple $P = \langle Vars, CP, en, err, E, \tau \rangle$, where $Vars$ is a set of program variables, CP is a set of cutpoints (i.e., program locations which represent heads of loops); $en, err \in CP$ are designated locations of the program entry and the error, respectively; $E \subseteq CP \times CP$ is the control-flow relation (to represent loop-free program fragments), and $\tau : E \to Expr(Vars)$ maps control edges to formulas in first-order logic that encode a transition relation of the correspondent loop-free program fragment. We refer to the graph $\langle CP, E \rangle$ as a* cutpoint graph *(CPG) of the program P.*

Throughout the chapter, we consider only variables that appear as source- and destination arguments for the edges E of the program P. In the formulas encoding transition relations τ, the other (local) variables are implicitly existentially quantified. Let $V : E \to 2^{Vars}$ be the function that, given a cutpoint, returns a set of variables live at that cutpoint. We use primed notation for $Vars'$ to distinguish between the source and the destination arguments of each edge.

To enable quantification over variables in a formula $e \in \mathit{Expr}$, we explicitly declare the variable sets over which e is expressed. For example, if e is expressed over $\vec{x}$ and $\vec{y}$, we write $e(\vec{x},\vec{y})$. If, in addition, $\vec{x}$ is existentially quantified in e, we write $\exists \vec{x} \,.\, e(\vec{x},\vec{y})$. However, when clear from the context, the variables declarations can be omitted from the formulas.

The goal of unbounded model checking is to check whether the location *err* is unreachable by any program behavior starting at the location *en*. One of the most common ways of proving safety of a program is to construct an *inductive invariant* that over-approximates the sets of reachable states in the program, and to prove the unreachability of *err* for the invariant. In the context of LBE, (safe) inductive invariants are represented by a labeling of the cutpoints with logical formulas, such that the condition(s) of the following definition hold.

Definition 19. *Given a program P, a mapping $\psi : CP \to \mathit{Expr}(\mathit{Vars})$ is an* inductive invariant *if:*

$$\forall (u,v) \in E \,.\, \Big(\psi(u)(\vec{x}) \wedge \tau(u,v)(\vec{x},\vec{x}') \implies \psi(v')(\vec{x}')\Big) \tag{4.1}$$

ψ is a proof *(or a* safe *inductive invariant) of P if additionally:*

$$\psi(err)(\vec{x}') \implies \bot \tag{4.2}$$

In (4.1), $\psi(u)$ is expressed over the source arguments of the cutpoint-edge (u,v) (denoted as $\vec{x}$). Similarly, $\psi(v')$ that stands for the primed version of $\psi(v)$, is expressed over the destination arguments of the cutpoint-edge (u,v) (denoted as $\vec{x}'$). Throughout the chapter, we add the following mnemonic notation to emphasize whether (4.2) hold for an inductive invariant: $|\psi|$ (with vertical bars) to indicate that (4.1) holds *alone*, and $\widehat{\psi}$ (with a hat) to indicate that *both*, (4.1) and (4.2), holds. If ψ is used without this mnemonic notation then in the current context it does not matter if (4.1) holds alone, or together with (4.2).

Since an inductive invariant over-approximates the sets of reachable states for each cutpoint of a program P, it allows more behaviors of P than specified by its transition relation τ. It can be used to represent programs that share the CPG-structure with P, but have less accurate transition relations. We say that such programs are the *abstractions* of P and describe them formally as follows.

Definition 20. *Given two programs $P = \langle \mathit{Vars}, CP, en, err, E, \tau\rangle$ and $\alpha P = \langle \mathit{Vars}, CP, en, err, E, \tau_\alpha\rangle$, αP is an* abstraction *of P if for some inductive invariant ψ of P,*

$$\forall (u,v) \in E \,.\, \Big(\psi(u)(\vec{x}) \wedge \tau(u,v)(\vec{x},\vec{x}') \implies \tau_\alpha(u,v)(\vec{x},\vec{x}')\Big) \tag{4.3}$$

The use of an inductive invariant ψ in (4.3) makes the way of creating abstractions more flexible. Indeed, for each cutpoint $u \in CP$, the formula $\psi(u)$ might bring any additional information about the pre-states at the edge $(u, v) \in E$ learned inductively from the dependent cutpoint-edges. Notably, $\tau(u, v)$ might be incomparable (and even inconsistent) with $\psi(u)$.

The simplest way to construct a program abstraction from the given inductive invariant ψ is to assign the transition relation of each cutpoint-edge by the invariant at the post-state of that edge. Thus, an abstraction of P can be constructed directly from ψ, and in rest of the chapter we will refer to it as to $\alpha^{\psi}P$. The following lemma assures that $\alpha^{\psi}P$ satisfies Def. 20.

Lemma 7. *Given $P = \langle Vars, CP, en, err, E, \tau\rangle$ and its invariant ψ, let $\alpha^{\psi}P = \langle Vars, CP, en, err, E, \tau^{\psi}_{\alpha}\rangle$ be defined as:*

$$\forall (u, v) \in E \,.\, \Big(\tau^{\psi}_{\alpha}(u, v)(\vec{x}, \vec{x}') \triangleq \psi(v')(\vec{x}')\Big) \tag{4.4}$$

Then $\alpha^{\psi}P$ is an abstraction of P.

If ψ is not trivial (i.e., $\exists u \in CP \,.\, \psi(u) \neq \top$) and some abstraction αP is as accurate as $\alpha^{\psi}P$ then αP provides a particular interest for incremental verification that is explained in Sect. 4.3. However, for the sake of completeness of presentation, we must admit that Def. 20 also allows other types of abstractions, whose abstract transition relation does not necessarily satisfy $\psi(v')$ for all post-states at (u, v).

Example 13. *Consider a program P_0 shown in Fig. 4.1 that increments a counter x, initially assigned to 0. The CPG of P_0 consists of $CP = \{en, CP_0, err\}$ and $E = \{(en, CP_0), (CP_0, CP_0), (CP_0, err)\}$. Fig. 4.2 shows: (1) transition relation τ_{P_0} labeling each edge in E, (2) the proof $\widehat{\psi}$ labeling each node in CP, (3-4) transition relations of two abstractions αP_0 and βP_0 respectively. Compared to αP_0, βP_0 allows variable x to be equal to 13 in the cutpoint err.*

4.1.2 Verification Based on Horn Solving

As we discussed in the previous Sect. 4.1.1, the verification conditions in LBE can be represented as a system of constraints which involves uninterpreted predicates. Consequently, the verification results rely on a decision procedure that is able to find an interpretation for the predicates that provides a suitable solution

```
int x = 0;
while ( * )
  { x++; }
if ( x < 0 )
  { error(); }
```

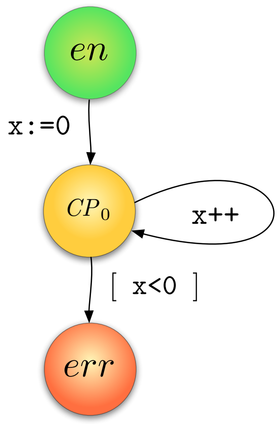


Figure 4.1. Program P_0 and its CPG.

$$\tau_{P_0} = \begin{cases} (en, CP_0) \mapsto (x' = 0) \\ (CP_0, CP_0) \mapsto (x' = x + 1) \\ (CP_0, err) \mapsto (x' = x \wedge x' < 0) \end{cases} \qquad \widehat{\psi} = \begin{cases} (en) \mapsto \top \\ (CP_0) \mapsto x \geq 0 \\ (err) \mapsto \bot \end{cases}$$

$$\tau_{\alpha P_0} = \begin{cases} (en, CP_0) \mapsto (x' \geq 0) \\ (CP_0, CP_0) \mapsto (x' \geq x) \\ (CP_0, err) \mapsto (x < 0) \end{cases} \qquad \tau_{\beta P_0} = \begin{cases} (en, CP_0) \mapsto (x' \geq 0) \\ (CP_0, CP_0) \mapsto (x' \geq x) \\ (CP_0, err) \mapsto (x < 0) \vee (x = 13) \end{cases}$$

Figure 4.2. Transition relation τ_{P_0}, proof $\widehat{\psi}$, transition relations for two abstractions of P_0: $\tau_{\alpha P_0}$ and $\tau_{\beta P_0}$.

for the corresponding system. In first-order logic, there is a recently established area of Horn solving that can be applied to perform this task.

Let us fix a vocabulary of interpreted symbols that consists of sets $\mathcal{F}$ and $\mathcal{P}$, that are fixed-arity function and predicate symbols, respectively. Suppose, we are given a set $\mathcal{R}$ of uninterpreted fixed-arity relation symbols, disjoint from $\mathcal{F}$ and $\mathcal{P}$, and a set $\mathcal{X}$ of variables.

Definition 21. *A Horn clause is a formula:*

$$S \wedge I_1 \wedge I_2 \wedge \ldots \wedge I_n \implies H \tag{4.5}$$

where S is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$; each I_i is an application of a relation symbol $r \in \mathcal{R}$ to first-order terms over $\mathcal{F}, \mathcal{X}$; H is either an application of $r \in \mathcal{R}$ to first-order terms over $\mathcal{F}$, or a constraint $\bot$.

The right-hand-side H of the implication (4.5) is called the *head* of the clause. The left-hand-side of (4.5) is called the *body* of the clause. A clause is called a *query* if its head is $\mathcal{R}$-free, and otherwise, it is called a *rule*. A rule with body $\top$ is called a *fact*. A clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. A set of Horn clauses HC over predicates $\mathcal{R}$ is called satisfiable (or solvable) if there is an interpretation of the predicates $\mathcal{R}$ such the universal closure of every clause $c \in HC$ holds.

Horn solving was shown applicable to synthesize generalized forms of Craig interpolation [121], such as classical binary interpolants, inductive sequences of interpolants, tree and DAG interpolants. Taking into account the role of interpolants in the state-of-the-art verification (overviewed in Sect. 2.1.2), Horn solving is becoming a highly valuable technique.

As said before, Horn solving is used to find inductive invariants of the programs. The conditions of Def. 19 can be encoded into the system of linear Horn clauses HC. In HC, constraints (4.1) are the rules, a constraint (4.2) is the query, and each $\psi(u)$ is a new uninterpreted predicate. Thus, if HC is solvable then the considered program is safe. Motivated by its use in model checking, the area of Horn solving has been being actively developed. The solving methods for Horn systems are based on predicate abstraction [69; 122], PDR [80], and sampling [129].

In the rest of the chapter, we exploit a Horn solving procedure as a black box. The solver is given a freedom for an underlying computation engine, as long as it is able to synthesize a safe inductive invariant. In the following Sect. 4.2, we will present another application for the Horn solving in synthesis of simulation relations applied for programs in LBE.

4.2 Simulation Relations for Proof Lifting

This section proposes a solution to the problem of automated discovery of simulations between programs with complicated loop structures. Our main contribution is SIMABS, a novel algorithm to automatically synthesize both, abstractions and simulations, between the source and the target programs. If the target does not simulate the source, SIMABS iteratively performs abstraction-refinement reasoning to detect an abstraction of the target that simulates the source. SIMABS operates by deciding validity of a sequence of $\forall\exists$-formulas, thus iterating over the basic method for simulation relation synthesis presented in Chap. 3.

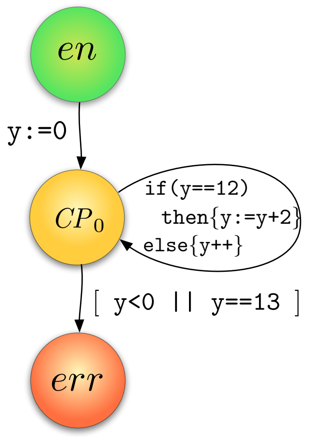
```
int y = 0;
while ( * ) {
  if ( y == 12 ) {
     y = y + 2; }
  else { y++; }
}
if ( y < 0 || y == 13 ) {
  error();
}
```

(a) Q_0

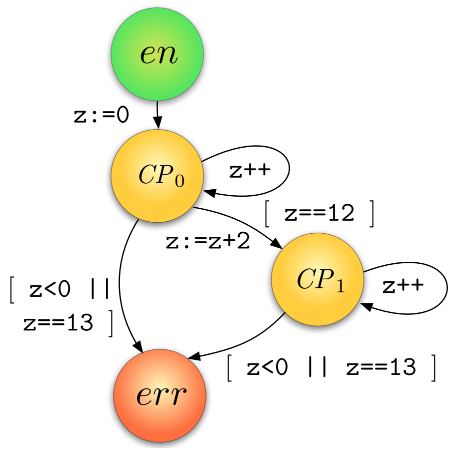
```
int z = 0;
while ( * && z < 12 ) {
  z++;
}
if ( z == 12 ) {
  z = z + 2;
}
while ( * && z > 12 ) {
  z++;
}
if ( z < 0 || z == 13 ) {
  error();
}
```

(b) Q_1

Figure 4.3. Program Q_0, and its loop-splitting optimization (+ variable renaming).



(a) Q_0



(b) Q_1

Figure 4.4. CPGs of Q_0, and Q_1.

4.2.1 Simulation Relations in Large-Block Encoding with Invariants

Given a pair of programs $P = \langle Vars_P, CP_P, en_P, err_P, E_P, \tau_P \rangle$ and $Q = \langle Vars_Q, CP_Q, en_Q, err_Q, E_Q, \tau_Q \rangle$. A simulation relation between P and Q specifies a matching of every program behavior of Q by some program behavior of

$$\tau_{Q_0} = \begin{cases} (en, CP_0) \mapsto (y' = 0) \\ (CP_0, CP_0) \mapsto \big((y = 12 \wedge y' = y + 2) \vee (y \neq 12 \wedge y' = y + 1)\big) \\ (CP_0, err) \mapsto \big((y' = y) \wedge (y' < 0 \vee y' = 13)\big) \end{cases}$$

$$\sigma = \begin{cases} en \mapsto en \\ CP_0 \mapsto CP_0 \\ err \mapsto err \end{cases} \qquad \rho = \begin{cases} (en, \sigma(en)) \mapsto \top \\ (CP_0, \sigma(CP_0)) \mapsto (x = y) \\ (err, \sigma(err)) \mapsto (x = y) \end{cases}$$

$$|\varphi| = \begin{cases} en \mapsto \top \\ CP_0 \mapsto \exists x \,.\, (x = y \wedge x \geq 0) \\ err \mapsto \exists x \,.\, (x = y \wedge x = 13) \end{cases} \qquad \widehat{\varphi} = \begin{cases} en \mapsto \top \\ CP_0 \mapsto (y \geq 0 \wedge y \neq 13) \\ err \mapsto \bot \end{cases}$$

Figure 4.5. Simulation relation between P_0 and $\tau_{\beta P_0}$, and lifting invariants.

$$\tau_{Q_1} = \begin{cases} (en, CP_0) \mapsto (z' = 0) \\ (CP_0, CP_0) \mapsto (z < 12 \wedge z' = z + 1) \\ (CP_0, CP_1) \mapsto (z = 12 \wedge z' = z + 2) \\ (CP_1, CP_1) \mapsto (z > 12 \wedge z' = z + 1) \\ (CP_0, err) \mapsto \big((z' = z) \wedge (z' < 0 \vee z' = 13)\big) \end{cases} \qquad \sigma = \begin{cases} en \mapsto en \\ CP_0 \mapsto CP_0 \\ CP_1 \mapsto CP_0 \\ err \mapsto err \end{cases}$$

$$\rho = \begin{cases} (en, \sigma(en)) \mapsto \top \\ (CP_0, \sigma(CP_0)) \mapsto (y = z) \\ (CP_0, \sigma(CP_0)) \mapsto (y = z) \\ (err, \sigma(err)) \mapsto (y = z) \end{cases} \qquad \widehat{\pi} = \begin{cases} en \mapsto \top \\ CP_0 \mapsto (z \geq 0 \wedge z \neq 13) \\ CP_1 \mapsto (z \geq 0 \wedge z \neq 13) \\ err \mapsto \bot \end{cases}$$

Figure 4.6. Simulation relation between Q_1 and Q_0, and lifting invariants.

P. In LBE, finding simulation relations is the two-steps procedure. First, it requires finding a simulation σ at the level of CPGs. Second, it requires finding a simulation ρ at the level of pairs of cutpoint-edges.

Definition 22. *Given two programs P and Q, we say that the CPG(P) simulates the CPG(Q) iff there exists a left-total relation $\sigma : CP_Q \to CP_P$, such that:*

$$\begin{aligned} \forall u_Q, v_Q \in CP_Q, u_P \in CP_P \,.\, (u_Q, v_Q) \in E_Q \wedge u_P = \sigma(u_Q) \implies \\ \exists v_P \in CP_P \,.\, (u_P, v_P) \in E_P \wedge v_P = \sigma(v_Q) \end{aligned} \tag{4.6}$$

When clear from the context, we omit the subscripts from u_Q, v_Q, etc.

Definition 23. *Program P simulates program Q iff* (1) *CPG(P) simulates the CPG(Q) via some σ, and* (2) *there exists a left-total relation $\rho : CP_Q \times CP_P \to$*

$Expr(Vars_Q \cup Vars_P)$, such that for some inductive invariant ψ of P:

$$\forall (u,v) \in E_Q \,.\, \Big(\psi(\sigma(u))(\vec{y}) \wedge \rho(u,\sigma(u))(\vec{x},\vec{y}) \wedge \tau_Q(u,v)(\vec{x},\vec{x}') \implies \exists \vec{y}' \,.\, \rho(v',\sigma(v'))(\vec{x}',\vec{y}') \wedge \tau_P\big(\sigma(u),\sigma(v)\big)(\vec{y},\vec{y}')\big)\Big) \tag{4.7}$$

For each edge (u,v) in (4.7), the existential quantifier in front of $\vec{y}'$ is served to encode existence of a valuation of the variables in $V'_P\big(\sigma(v)\big)$. In contrast, valuations of the variables $\vec{x}$, $\vec{x}'$, $\vec{y}$ respectively of $V_Q(u)$, $V_Q(v)$ and $V_P\big(\sigma(u)\big)$ are implicitly universally quantified. Thus, for each $\vec{x}$ and $\vec{y}$ matched by $\rho(u,\sigma(u))$ and $\vec{x}'$, there should exists $\vec{y}'$, such that $\vec{x}'$ and $\vec{y}'$ are matched by $\rho(v',\sigma(v'))$. Additionally, the pairs $\vec{x}$ and $\vec{x}'$, and $\vec{y}$ and $\vec{y}'$ should belong to valid behaviors corresponding to their transition relations $\tau_Q(\sigma(u),\sigma(v))$ and $\tau_P(u,v)$ respectively. Notably, those transition-relation formulas are conjoined with the different sides of the implication, so the validity of the $\forall\exists$-formula means that each behavior of $\tau_Q(\sigma(u),\sigma(v))$ is matched by a behavior of $\tau_P(u,v)$ (but it is still allowed to have unmatched behaviors of $\tau_P(u,v)$). For this, the simulation relation induced by formulas $\rho(u,\sigma(u))$ and $\rho(v',\sigma(v'))$ is required to be left-total.

In contrast to the classical definition of simulation used in Chap. 3, this definition of simulation relation exploits an inductive invariant ψ of P that over-approximates the sets of reachable states for each cutpoint of P. In particular, for each cutpoint-edge (u,v) of Q, the condition of Def. 23 restricts the set of pre-states of $\tau_P(\sigma(u),\sigma(v))$ on $\psi(\sigma(u))$. Such restriction is sound, since it does not drop any behavioral information of P that can be potentially useful while constructing and checking a simulation of Q. Furthermore, for each behavior of Q requiring to be matched by some behavior of P, the invariant ψ reduces the search space of this matching.

Whenever for a given pair of programs P and Q, there exists the pair of relations $\langle\sigma,\rho\rangle$, such that P simulates Q, we write $Q \preceq_{\sigma,\rho} P$, or simply $Q \preceq P$ if $\langle\sigma,\rho\rangle$ are clear from the context.

An important practical aspect that makes simulation relations useful for incremental verification is their ability to lift the proofs between programs. In fact, if the error location err_P is proven unreachable in a program P, and the program P simulates another program Q, then the error location err_Q is unreachable in Q. Interestingly, this fact can be further propagated to the level of inductive invariants [107] making the following lemma hold:

Lemma 8. *Given programs P and Q, let ψ be a (safe) inductive invariant of P and $Q \preceq_{\sigma,\rho} P$. Consider a mapping $\varphi : CP_Q \to Expr(Vars_Q)$ defined for each $u \in CP_Q$ such that:*

$$\varphi(u)(\vec{x}) \triangleq \exists \vec{y} \,.\, \rho(u, \sigma(u))(\vec{x}, \vec{y}) \wedge \psi(\sigma(u))(\vec{y}) \tag{4.8}$$

Then φ is a (safe) inductive invariant of Q.

Example 14. *Suppose that P_0 (shown in Fig. 4.1) evolved to a "lucky" program Q_0 (shown in Fig. 4.3(a), 4.4(a)) such that the counter jumps over the value "13": the new variable y appeared instead of x, and the program fragment corresponding to the looping edge (CP_0,CP_0) is replaced by* `if (y==12) then {y=y+2} else {y++}`*. More importantly, the property in Q_0 is stronger than in P_1: in addition to be positive, y is restricted to be not equal to 13. The CPG of P_0 and the CPG of Q_0 are identical. Notably, $Q_0 \not\preceq P_0$, $Q_0 \not\preceq \alpha P_0$, but $Q_0 \preceq \beta P_0$. Fig. 4.5 shows: (1) transition relation τ_{Q_0}, (2) relation ρ for Q_0 and βP_0, (3) lifted inductive (but not safe) invariant $|\varphi|$ labeling each node in CP of Q_0, (4) proof $\widehat{\varphi}$ of Q_0 obtained from $|\varphi|$ by inductive strengthening.*

In order to obtain the inductive invariant $|\varphi|$ for Ex. 14, we first need to weaken $\widehat{\psi}$ (as in Ex. 13) to be an inductive invariant of βP_0. Weakening can be done, e.g., by replacing the labeling $\widehat{\psi}(err) = \perp$ by a formula $x = 13$ (since $\perp$ obviously implies anything including $x = 13$). Finally, $|\varphi|$ can be further strengthened to be $\widehat{\varphi}$ using an unbounded model checker (and in turn, a Horn solver) to become safe.

Example 15. *Consider a loop-splitting optimization Q_1 (shown in Fig. 4.3(b), 4.4(b)) of Q_0 (shown in Fig. 4.3(a), 4.4(a)). It is produced by inserting an* `if`*-conditional outside of the* `while`*-loop and renaming of the variables. As a result, an extra loop (and an extra cutpoint CP_1) appeared in Q_1, and both loops were simplified to contain only an increment operator z++. Notably, $Q_1 \preceq Q_0$. Fig. 4.6 shows: (1) transition relation τ_{Q_1}, (2) relation σ for Q_1 and Q_0, (3) relation ρ for Q_1 and Q_0, (4) lifted inductive (and safe) invariant $\widehat{\pi}$ of Q_1.*

For now, we omit the details of computing the relations as in Ex. 14-15. We return to explaining methods for simulation relation synthesis in Sect. 4.2.2, while the methods for lifting proofs are left for Sect. 4.3.

4.2.2 SimAbs: Simulation-Abstraction-Refinement Loop

This section generalizes and automatizes the approach of the symbolic simulation discovery introduced in Chap. 3 to programs with non-trivial control-flow graphs.

We give a sketch of SimAbs in Alg. 9. The algorithm starts with iterative search of some unrolling of P, CPG of which represents a supergraph of CPG (Q). SimAbs handles a number of heuristics to iteratively find a proper unrolling of P (method Unroll) and a relation for the CPGs σ. It traverses both graphs and tries unrolling different loops in a branch-and-bound manner. Checking whether one graph is a supergraph of another one is reduced to checking validity of the $\forall\exists$-formula (4.6), but it requires some relation σ to be guessed (e.g., in our implementation we exploit similarities of the names of variables used in the given cutpoints). If the formula is valid then the proper σ is found. This routine is not guaranteed to terminate (i.e., if there is no simulation on the level of CPGs, SimAbs will keep iterating forever), so the algorithm is parametrized by a method CanUnroll (either by using some maximal unrolling depth or a timeout).

The further reasoning of SimAbs follows the abstraction-refinement paradigm. The algorithm proceeds to discover an abstraction αP and refine it as much as possible. If $\alpha P \neq \mathbb{U}$ and αP satisfies some quality metric $\mathcal{Q}$ (e.g., if it preserves some safety property of P), SimAbs also returns a simulation relation ρ_α^{ext}, such that $Q \preceq_{\sigma,\rho_\alpha^{ext}} \alpha P$. SimAbs uses Abstract (which in turn uses Synthesize) to guess an initial relation ρ. The initial guess can be an arbitrary total relation between the variables of each cutpoint $v \in CP_Q$ and a corresponding cutpoint $\sigma(v) \in CP_P$. In our implementation (line 2 of Abstract), for every cutpoint v, we take $\rho(v)$ to be a conjunction of equalities between the *live* variables of Q and P respectively at v and $\sigma(v)$ that have identical names.

Abstract (outlined in Alg. 10) iteratively constructs a $\forall\exists$-formula (4.7) for each edge $(u,v) \in E_Q$ (line 4), and checks its validity. If the check succeeds for all edges, ρ is returned to the main loop of SimAbs to be further refined. Otherwise, Abstract chooses an abstraction αP of P using the method Weaken (line 5), and repeats the check for Q and αP (line 7). Note that in the next iteration of Abstract, ρ will be weakened correspondingly, since Synthesize in that iteration is given Q and αP.

Weaken introduces non-determinism to P. Since based on inductive invariant ψ of P provided to the algorithm, the simplest implementation of Weaken would immediately return a ψ-safe abstraction $\alpha^\psi P$ (defined in Lemma 7). Another implementation (shown in Alg. 13) existentially abstracts away a subset of

Algorithm 9: SIMABS(Q, P)

```
Input: programs $Q$ and $P$, abstraction quality metric $\mathcal{Q} : \alpha \rightarrow \{\top, \bot\}$,
       inductive invariant $\psi$ of $P$
Output: an abstraction $\alpha^{ext}P$, a CPG relation $\sigma$, and a simulation relation
        $(\sigma, \rho_\alpha^{ext})$, s.t. $Q \preceq_{\sigma,\rho_\alpha^{ext}} \alpha^{ext}P$
Data: universal abstraction $\mathbb{U}$
1  forever do
2      $\sigma \leftarrow$ GUESS$(P, Q)$;                                ▷ Guess a mapping between cutpoints
3      if (CPG$(Q) \preceq_\sigma$ CPG$(P)$) then break;                 ▷ If $\sigma$ is valid, go to line 7
4      else if (CANUNROLL$(P)$) then
5          $P \leftarrow$ UNROLL$(P)$;                                   ▷ If not, iteratively replace $P$ by its unrolling
6      else return $\mathbb{U}, \varnothing, \varnothing$;               ▷ Until no unrolling is possible
7  $\alpha^{ext}P \leftarrow P$;
8  forever do                                                          ▷ Use $\sigma$ to synthesize $\rho$
9      $\alpha_{pre}P \leftarrow \alpha^{ext}P$;
10     $\alpha P, \rho_\alpha \leftarrow$ ABSTRACT$(Q, P, \sigma, \psi)$;  ▷ Guess, check and abstract $\rho$ (and correspondingly, $P$)
11     if ($\alpha P \neq \mathbb{U}$) then return $\mathbb{U}, \varnothing, \varnothing$;
12     $\alpha^{ext}P, \rho_\alpha^{ext} \leftarrow$ REFINE$(Q, \alpha P, \sigma, \rho_\alpha, \psi)$;  ▷ Attempt to refine $\rho_\alpha$ and $\alpha P$
13     if ($\mathcal{Q}(\alpha^{ext}P) \vee (\alpha_{pre}P = \alpha^{ext}P)$) then
14         return $\alpha^{ext}P, \sigma, \rho_\alpha^{ext}$;
```

input variables of the edge (u, v) for which the simulation check has failed. The further possible weakening would be based on predicate abstractions.

REFINE (outlined in Alg. 11) constructs a refinement ρ_α^{ext} of simulation relation ρ_α, and the corresponding strengthening $\alpha^{ext}P$ of abstraction αP. REFINE maintains a work-list *WL* of the cutpoint edges to be processed. Initially, *WL* is populated with E_Q (line 3). In each iteration, while processing the edge (u, v), REFINE adds a Skolem relation *Sk* to $\rho_\alpha(u)$ (line 10). *Sk* is produced for the existentially abstracted input variables in $(\sigma(u), \sigma(v))$ and furthermore is used to strengthen $\alpha^{ext}P$ (line 11). At the same time, method STRENGTHEN (Alg. 14) performs an opposite action to WEAKEN with the current abstraction αP, i.e., removes nondeterminism from αP using *Sk*.

Finally, REFINE updates *WL* with the outgoing edges from u and other incoming edges to u (line 12) and iterates until *WL* is empty (line 14). If in some iteration a strengthening is impossible, REFINE returns the last successful values for ρ_α^{ext} and $\alpha^{ext}P$ (line 13).

Algorithm 10: ABSTRACT(Q, P, σ, ψ)

```
Input: programs $Q$ and $P$, CPG relation $\sigma$, inductive invariant $\psi$ of $P$
Output: abstraction $\alpha P$, simulation relation $\rho_\alpha$, such that $Q \preceq_{\rho_\alpha} \alpha P$
Data: $\forall\exists$-formula: $sim$

1  for each $(u, v) \in E_Q$ do
2      $\rho(v) \leftarrow$ SYNTHESIZE$(\tau_Q(u, v), \tau_P(\sigma(u), \sigma(v)))$;
3      $sim \leftarrow$ CREATEFORMULA$(\tau_Q(u, v), \tau_P(\sigma(u), \sigma(v)), \rho, \psi)$;
4      if $(\neg$ISVALID$(sim))$ then
5          $\alpha P \leftarrow$ WEAKEN$(P, V(\sigma(u)) \cup V(\sigma(v)))$;
6          if $(\alpha P \neq \mathbb{U})$ then
7              return ABSTRACT$(Q, \alpha P, \sigma, \psi)$;
8          else return $\mathbb{U}, \varnothing$;
9  return $P, \rho$;
```

Algorithm 11: REFINE$(Q, \alpha P, \sigma, \rho_\alpha, \psi)$

```
Input: program $Q$, abstraction $\alpha P$, CPG relation $\sigma$, simulation relation
       $\rho_\alpha$, inductive invariant $\psi$ of $P$
Output: abstraction $\alpha^{ext} P$, simulation relation $\rho_\alpha^{ext}$
Data: $\forall\exists$-formula: $sim$

1  $\rho_\alpha^{ext} \leftarrow \rho_\alpha$;
2  $\alpha^{ext} P \leftarrow \alpha P$;
3  $WL \leftarrow E_Q$;
4  while $(WL \neq \varnothing)$ do
5      $(u, v) \leftarrow$ GETEDGE $(WL)$;
6      $WL \leftarrow WL \setminus \{(u, v)\}$;
7      $sim \leftarrow$ CREATEFORMULA$(\tau_Q(u, v), \tau_P(\sigma(u), \sigma(v)), \rho_\alpha^{ext}, \psi)$;
8      if (ISVALID$(sim)$) then
9          $Sk \leftarrow$ SKOLEM$(sim)$;
10         $\rho_\alpha^{ext}(u) \leftarrow \rho_\alpha^{ext}(u) \wedge Sk$;
11         $\alpha^{ext} P \leftarrow$ STRENGTHEN$(\alpha P, Sk)$;
12         $WL \leftarrow WL \cup \{(u, x) \in E \mid x \in CP\} \cup \{(y, u) \in E \mid y \in CP\}$;
13     else return $\alpha P, \rho_\alpha$;
14 return $\alpha^{ext} P, \rho_\alpha^{ext}$;
```

Algorithm 12: SYNTHESIZE(S, T)

```
  Input: loop-free programs $S, T$
  Output: candidate relation $\rho$
1 return $\bigwedge_{a'_S \in V'_S, a'_T \in V'_T} (a'_S = a'_T)$;
```

Algorithm 13: WEAKEN(P, U)

```
  Input: program $P$, $U \subseteq V_P$
  Output: abstraction $\alpha P$
1 guess $U' \subseteq U$;
2 return $\alpha^{\exists}_{U'}(P)$;
```

Algorithm 14: STRENGTHEN($\alpha P, Sk$)

```
  Input: abstraction $\alpha P$, relation $Sk$
  Output: abstraction $\alpha^{ext} P$
1 $U^{ext} \leftarrow V(\alpha P) \cup V(Sk)$;
2 return $\alpha^{\exists}_{V(P) \setminus U^{ext}}(P)$;
```

For the progress of the algorithm, it is enough to note that in each iteration of SIMABS, ABSTRACT is given a concrete program P, and always constructs a new abstraction from scratch. Thus, if the space of possible abstractions is finite (which is the case for existential abstraction) the algorithm always terminates.

4.2.3 Horn Solving for Skolem Extraction

It is worth reminding that for every iteration of ABSTRACT, as well as for every iteration of REFINE, there is a need to decide validity of a set of simulation-abstraction-checking formulas (4.7). For this goal, SIMABS invokes AE-VAL (presented in Chap. 3, Alg. 6), a decision procedure for $\forall\exists$-formulas of the form $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$ strengthened with Skolem extracting capabilities. Algorithm SKOLEM, used as a subroutine of REFINE, is an essential extension of AE-VAL that produces a Skolem relation.

AE-VAL is based on a notion of MBP described in Sect. 3.1.2, that under-approximates existential quantification. Given a formula $\varphi(\vec{x}, \vec{y})$, let $m \models \varphi(\vec{x}, \vec{y})$

denote a model of φ. An $MBP_{\vec{y}}$ is a function that given a model m, returns a $\vec{y}$-free formula, such that (1) m is also a model of the image $MBP_{\vec{y}}(m,\varphi)$, and (2) $MBP_{\vec{y}}(m,\varphi)$ is an under-approximation of $\exists \vec{y} \,.\, \varphi(\vec{x},\vec{y})$. AE-VAL proceeds by iterative enumerating models of the quantified formula until all of them are distributed in a finite number of projections $\{T_i(\vec{x})\}$, where each $T_i(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x},\vec{y})$. In order to construct a Skolem relation, AE-VAL first obtains a *local* Skolem relation $\phi_i(\vec{x},\vec{y})$ for each partition and composes a global Skolem relation $Sk_{\vec{y}}(\vec{x},\vec{y})$ (as in (4.9)), by matching the local Skolem relations with guards represented by $\vec{y}$-free predicates $I_i(\vec{x})$. In Chap. 3, we used the simplest interpretation of the guards, by the projections themselves, i.e., for each i, $I_i(\vec{x}) = T_i(\vec{x})$, i.e., as in (4.9).

$$Sk_{\vec{y}}(\vec{x},\vec{y}) \equiv \begin{cases} \phi_1(\vec{x},\vec{y}) & \text{if } I_1(\vec{x}) \\ \phi_2(\vec{x},\vec{y}) & \text{else if } I_2(\vec{x}) \\ \cdots & \quad\text{else } \cdots \\ \phi_n(\vec{x},\vec{y}) & \qquad\text{else } I_n(\vec{x}) \end{cases} \tag{4.9}$$

Intuitively, ϕ_i maps each model of $S \wedge T_i$ to a model of T. However, $\{T_i\}$ are not disjoint, and each conjunction $S \wedge T_i$ could be simplified and minimized. In the context of simulation relation, there are additional restrictions for the variables over whose the guards are expressed. That is, we do not want a Skolem relation for local variables, but only for existentially abstracted input variables in the cutpoint edge under consideration.

Thus, to adjust the Skolem relation Sk, we need to find another partitioning $\{I_i\}_{i=1}^n$ of S, such that each partition I_i must be associated with an appropriate ϕ_i and expressed over some subset $\vec{x}^{(i)} \subseteq \vec{x}$. The constraints on the partitions I_i are as follows. First, a partition I_i must cover all models of T_i that are not already covered by $I_1 \ldots I_{i-1}$. Second, it should not include any models that are not contained in T_i. Writing these requirements formally, we get the system of constraints (4.10).

$$\begin{cases} S(\vec{x}) \wedge T_1(\vec{x}) \implies I_1(\vec{x}^{(1)}) \\ S(\vec{x}) \wedge T_2(\vec{x}) \wedge \neg T_1(\vec{x}) \implies I_2(\vec{x}^{(2)}) \\ \dots \\ S(\vec{x}) \wedge T_n(\vec{x}) \wedge \neg T_1(\vec{x}) \wedge \dots \wedge \neg T_{n-1}(\vec{x}) \implies I_n(\vec{x}^{(n)}) \\ S(\vec{x}) \wedge I_1(\vec{x}^{(1)}) \wedge \neg T_1(\vec{x}) \implies \bot \\ \dots \\ S(\vec{x}) \wedge I_n(\vec{x}^{(n)}) \wedge \neg T_n(\vec{x}) \implies \bot \end{cases} \tag{4.10}$$

Note that in (4.10), S and $\{T_i\}$ are the first-order formulas, and $\{I_i\}$ are the uninterpreted predicates. The set of constrains corresponds to a system of non-linear Horn clauses. Thus, we can find an interpretation of the predicates $\{I_i\}$ using a Horn solver. In our implementation, we use the solver of Z3, but other solutions, for example, based on interpolation, are also possible. The solution for $\{I_i\}$ is then substituted to (4.9) that provides a more fine-grained guarded Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$.

A simple way to find a minimal solution for $\{I_i\}$ is to iteratively restrict the number of variables $\vec{x}^{(i)}$ in each partition in (4.10) until no smaller solution can be found. We leave the problem of finding the minimum partitioning for future work.

4.3 Migrating Proofs between Programs

The main target of this section is to establish a so called *Property Directed Equivalence* (PDE) between programs, i.e., to check whether the programs P and Q both satisfy the same property (and consequently, are happy with the same proof ψ). Here we show how the simulation relations and abstractions synthesized by means of SimAbs can be used to lift the proofs between programs.

4.3.1 Abstract Simulations for Proof Lifting

Given an abstraction αP of P and a proof $\widehat{\psi}$ of P, we say that αP is *$\widehat{\psi}$-safe* iff $\widehat{\psi}$ is also a proof of αP. Not every abstraction of P is $\widehat{\psi}$-safe, but there might exist several $\widehat{\psi}$-safe abstractions of P of different precision, and the most precise one of those is P itself. Formally, it is reflected in the following definition.

Definition 24. *Given a program P and a proof $\widehat{\psi}$. An abstraction $\alpha P = \langle Vars, CP, en, err, E, \tau_\alpha \rangle$ of P is $\widehat{\psi}$-safe iff the following holds:*

$$\forall (u,v) \in E \,.\, \widehat{\psi}(u)(\vec{x}) \wedge \tau_\alpha(u,v)(\vec{x},\vec{x}') \implies \widehat{\psi}(v')(\vec{x}') \tag{4.11}$$

Definition 25. *Programs P and Q are $\widehat{\psi}$-equivalent iff there exists another program R, such that $P \preceq R$ and $Q \preceq R$ and $\widehat{\psi}$ is a proof of R.*

In Def. 25, we allow R to be either P or Q, depending on whether $\widehat{\psi}$ is a proof of P or Q, respectively. Similarly, R is allowed to be an abstraction of P or Q.

Example 16. *Programs P_0 and Q_0 (shown in Fig. 4.1 and Fig. 4.3(a) respectively) are not $\widehat{\psi}$-equivalent, since we can not find a $\widehat{\psi}$-safe abstraction of P_0 (αP_0 is $\widehat{\psi}$-safe, but $Q_0 \not\preceq \alpha P_0$, and βP_0 is not $\widehat{\psi}$-safe). In contrast, Q_0 and Q_1 (shown in Fig. 4.3(a) and Fig. 4.3(b) respectively) are $\widehat{\psi}$-equivalent, since we have shown in Ex. 15 that $Q_1 \preceq Q_0$.*

The FIV problem for P, Q and $\widehat{\psi}$ can be stated as establishing a $\widehat{\psi}$-equivalence between P and Q. In this chapter, we want to provide not only a generic, but also an efficient solution to the FIV problem. One crucial obstacle on the way towards efficiency is that the simulation synthesis in general requires more efforts for solving than verifying Q from scratch. However, for ProofAdapt it is not required to have Q simulated by the precise program P via some precise $\langle \sigma, \rho \rangle$. Instead, it is enough to find a $\widehat{\psi}$-safe abstraction αP that simulates Q via some abstract $\langle \sigma, \rho_\alpha \rangle$. Detecting ρ_α is expected to be not so hard as detecting ρ, and to have more chances to converge.

Theorem 5. *Given programs P, αP and Q. Let $\widehat{\psi}$ be a proof of P and αP be a $\widehat{\psi}$-safe abstraction of P. If $Q \preceq \alpha P$ then P and Q are $\widehat{\psi}$-equivalent.*

The tie that binds the abstraction and the simulation in Th. 5 is the proof $\widehat{\psi}$. In practice, synthesis of αP and $\langle \sigma, \rho_\alpha \rangle$ is benefitted from the guidance by $\widehat{\psi}$. Furthermore, when discovered, $\langle \sigma, \rho_\alpha \rangle$ is directly used to migrate $\widehat{\psi}$ from P to Q. In the rest of the section, we elaborate on these routines in more detail.

4.3.2 Basic Proof Lifting Algorithm

The main practical importance of PDE, introduced in Sect. 4.3, is that it allows adapting a proof $\widehat{\psi}$ of program P to a proof $\widehat{\varphi}$ of program Q if there exists a $\widehat{\psi}$-safe abstraction of P that simulates Q (recall Th. 5). In such a case, no additional

Algorithm 15: PROOFADAPT (P, Q, ψ)

```
Input: Programs $P, Q$ proof $\widehat{\psi}$ of $P$
Output: Verification result $res \in \{$SAFE, BUGGY$\}$, proof $\widehat{\varphi}$ of $Q$
Data: Candidate invariant $\exists\langle\sigma,\rho_\alpha\rangle\widehat{\psi}$ of $Q$
1 $\langle\sigma,\rho_\alpha\rangle, \alpha P \leftarrow$ SIMABS$(Q, P, \widehat{\psi})$;   ▷ Find an abstraction $\alpha P$ of $P$, s.t. $Q \preceq \alpha P$
2 if (ISPSISAFE$(\alpha P, \psi)$) then
3     return SAFE, $\exists\langle\sigma,\rho_\alpha\rangle\widehat{\psi}$;   ▷ Apply Th. 5
4 else
5     $|\psi| \leftarrow$ WEAKEN$(\widehat{\psi})$;   ▷ Weaken $\widehat{\psi}$ such that it is inductive for $\alpha P$
6     $|\varphi| \leftarrow \exists\langle\sigma,\rho_\alpha\rangle|\psi|$;   ▷ Lift invariant to $\alpha P$
7     return VERIFY$(Q, |\varphi|)$;   ▷ Strengthen $|\varphi|$ until it is SAFE or a *Cex* is found
```

analysis of Q is required unless there is a need to eliminate existential quantifiers from the adapted proof (we elaborate on such a scenario in Sect. 3.1.2). However, if the conditions of Th. 5 are not met, we are still interested in accelerating the verification process for Q. In particular, if the detected abstraction αP of P is not $\widehat{\psi}$-safe, we still may be able to lift some (not safe, but) inductive invariant to be further strengthened by a Horn-based model checker.

In the rest of this section, we address the problem of verifying Q using P and $\widehat{\psi}$. Our solution is outlined in Alg. 15. PROOFADAPT proceeds as follows. First (line 1) it invokes a 2-steps procedure of SIMABS (Alg. 9): (1) obtaining a relation σ between cutpoints of P and Q via iterative unrolling of P and checking validity of the implication (4.6); (2) discovering an abstraction αP of P and a relation ρ_α such that simulates $Q \preceq_{\sigma,\rho_\alpha} P$.

The discovered abstraction αP is then checked for being $\widehat{\psi}$-safe (line 2). This is done by deciding validity of a set of implications (4.11) for each edge of the CPG of P. If this check succeeds then the simulation relation ρ_α discovered by means of SIMABS is combined with $\widehat{\psi}$ using existential quantification to obtain a mapping $\exists\langle\sigma,\rho_\alpha\rangle\widehat{\psi}$ (defined in (4.8)).

If αP is not $\widehat{\psi}$-safe then ρ_α can not be directly used to lift invariants. But since $P \preceq \alpha P$ by the identity relation (i.e., using the same variable names), $\widehat{\psi}$ can be weakened to become an inductive invariant $|\psi|$ of αP (line 5). Method WEAKEN can implement different methods including simple generation of the strongest post-condition (as in Ex. 17), or a *counter-example guided inductive weakening* (method MKIND, to be discussed in Sect. 4.3.3).

Finally, ProofAdapt relies on Verify to perform iterative strengthening of the inductive invariant $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ provided by Weaken and existentially conjoined with ρ_α. In particular, if $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ was already safe, Verify would return immediately. However, since at this step of ProofAdapt the abstraction αP is not $\widehat{\psi}$-safe, $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ is also not safe. Another verifier that cannot work by strengthening a given invariant would be useless, as it would drop $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ completely and verify Q from scratch.

Example 17. *Consider programs P_0 and Q_0 (shown in Fig. 4.1 and Fig. 4.3(a) respectively). Suppose, P_0 is verified and has a proof $\widehat{\psi}$ (shown in Fig. 4.2). Let us show how* ProofAdapt *operates in order to derive the proof $\widehat{\varphi}$ of Q_0 (envisioned in Fig. 4.5). First,* ProofAdapt *invokes* SimAbs *to iteratively abstract P_0, e.g, to αP_0 and to βP_0 (both shown in Fig. 4.2) and check whether the abstraction simulates Q_0: the former does not, but the latter does. Second,* ProofAdapt *confirms that βP_0 is not $\widehat{\psi}$-safe and thus proceeds to the weakening-strengthening routine.*

While doing Weaken, ProofAdapt *exploits the efforts spent on checking that βP_0 is not $\widehat{\psi}$-safe. In particular, $\widehat{\psi}$ is broken for the edge (CP_0, err), i.e., the following implication is invalid:*

$$(x \geq 0) \wedge \big((x < 0) \vee (x = 13)\big) \implies \bot$$

This means that the $\bot$ is too strong to label err in $|\psi|$, and a weaker formula should be discovered. For example, the labeling $|\psi|$ of the cutpoint err can be immediately assigned to $\top$ (e.g., aggressive weakening). Alternatively, $|\psi|(err)$ can be assigned to the strongest post-condition for the cutpoint edge (CP_0, err) and its pre-condition $\psi(CP_0)$ (e.g., forward-reachability-based weakening). That is, if $|\psi|(err) = (x = 13)$, the inductive invariant $|\psi|$ of βP_0 is obtained. Finally, $|\psi|$ is lifted to become inductive invariant $|\varphi|$ of Q_0 using the already established simulation relation ρ_α ($|\varphi|$ is shown in Fig. 4.5). The last step of strengthening $|\varphi|$ to become a proof $\widehat{\varphi}$ of Q_0 is due to Verify *which will be described in more detail in Sect. 4.3.5.*

4.3.3 Counter-Example Guided Inductive Weakening

In this section, we propose a method called MkInd that performs weakening of an invariant ψ of program P to be an inductive invariant φ of program Q using an incremental SMT solver. We require that P and Q share the same CPG, and differ only in the labeling of edges. If Q is known to be an abstraction of P (i.e., as in our application in algorithm ProofAdapt), this requirement is trivially fulfilled.

```
Algorithm 16: MkInd($\psi, Q$)
Input: Candidate invariant $\psi$, program $Q$
Output: Inductive invariant $\varphi : CP \to Expr(Vars)$ of $Q$
1  $\varphi \leftarrow \psi$;
2  $WL \leftarrow E$;
3  while $WL \neq \emptyset$ do
4      $(u, v) \leftarrow$ getEdge($WL$);
5      $WL \leftarrow WL \setminus \{(u, v)\}$;
6      $old\_post \leftarrow \varphi(v)$;
7      $\varphi(v) \leftarrow$ WeakPost$\big(\varphi(u), \tau(u, v), \varphi(v)\big)$;
8      if $\big(old\_post \neq \varphi(v)\big)$ then
9          $WL \leftarrow WL \cup \{(v, x) \in E \mid x \in CP\}$;
10 return $\varphi$;
```

MkInd is shown in Alg. 16. The algorithm maintains a work-list $WL \subseteq E$ that is initialized with all the edges $(u, v) \in E$ and iteratively adjusts φ initially assigned by ψ. In each iteration of the main loop, first, an edge $(u, v) \in WL$ that is the least in the Weak Topological Ordering (WTO) [25] (in which inner loops are traversed before outer loops) is picked. Second, an SMT solver is used to check whether the formula (4.1) is valid for the current values of $\varphi(u)$, $\varphi(v)$ and $\tau(u, v)$. If this is not the case, $\varphi(v)$ is weakened until the triple becomes valid and all outgoing edges of v are added to WL. Soundness of MkInd is immediate – the work-list is empty only if every cutpoint edge is annotated with a valid invariant. Termination follows from the fact that at each iteration either the work-list is reduced, or a $\varphi(v)$ is weakened, and, our implementation of WeakPost allows only for finitely many weakening steps.

WeakPost is shown in Alg. 17. The input is a formula *pre* (also referred to as pre-condition), a formula *post* (also referred to as post-condition) and a loop-free program fragment S. The output is a weakening $post'$ of *post* such that $pre \wedge S \implies post'$ is valid. We assume that *post* is given as a conjunction of lemmas, i.e., $post = \bigwedge_i \ell_i$. The algorithm computes the (possibly empty) subset of $\{\ell_i\}$ that forms a valid post-condition.

The naive implementation of WeakPost iteratively checks whether each ℓ_i is a post-condition. Instead, we use an incremental SMT solver to do this enumeration efficiently. We assume that in addition to the SmtSolve API, an SMT solver has the method SmtAssert to add constraints to the current context.

Algorithm 17: WEAKPOST($pre, S, post$)

```
Input: $pre, S, post \in Expr$; $post = \bigwedge_{i=0}^{n} \ell_i$
Output: $post' \in Expr$, such that $pre \wedge S \implies post'$
1  let $\{x_i \mid 0 \leq i \leq n\}$ be fresh Boolean variables; $U \leftarrow \{0, \dots, n\}$;
2  SMTASSERT$\Big(pre \wedge S \wedge \neg\big(\bigwedge_{i \in U} (x_i \implies \ell'_i)\big)\Big)$;
3  while $\Big($ISSAT$\big($SMTSOLVE()$\big)\Big)$ do
4      $m \leftarrow$ SMTMODEL();
5      foreach $\{0 \leq i \leq n \mid m \models x_i\}$ do
6          SMTASSERT($\neg x_i$);
7          $U \leftarrow U \setminus \{i\}$;
8      end
9  $post' \leftarrow \bigwedge_{i \in U} \ell_i$;
10 return $post'$;
```

In each iteration of our algorithm, the SMT context contains the formula (further referred to as Φ) that determines the validity of the current candidate post-condition. Initially, the entire post-condition is asserted under assumptions (line 2), encoded by Boolean variables x_i such that lemma ℓ_i is active iff x_i is true. In the next iterations, the algorithm may disable some of the lemmas simply by negating the corresponding assumptions.

The core of the algorithm is in the incremental disabling of assumptions. If in some iteration, there exists a model m of Φ then m identifies the variable(s) x_i to be negated (line 6). The correspondent lemma(s) are to be excluded from the current candidate post-condition and the algorithm iterates. This terminates eventually (whenever the formula in the current context Φ is unsatisfiable) since there are finitely many lemmas and at least one is disabled at every iteration. The conjunction of all remained lemmas is returned as *post'*.

In the worst case scenario, *post'* is equal to $\top$ which means all lemmas were disabled. The more lemmas are contained in the input formula *post*, the more effective could be the algorithm. Thus, before passing *post* into the algorithm, it makes sense to additionally factor *post* (e.g., replace terms like $z = 0$ by $(z \geq 0) \wedge (z \leq 0)$), or enhance *post* by additional weaker lemmas (e.g., add terms like $(z \geq -10) \wedge (z \leq 10)$ to $(z \geq 0) \wedge (z \leq 0)$).

4.3.4 Eliminating Quantifiers from Lifted Invariants

In this subsection we continue discussing algorithm ProofAdapt, and in particular, focus on obtaining a quantifier-free formula φ that is equivalent to $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ (line 6) and check whether φ is an inductive invariant of Q. However, this procedure is expensive and for the sake of practical efficiency φ may be replaced by some under-approximation $\widehat{\varphi}$ of φ. However, $\widehat{\varphi}$ may be unnecessarily strong, and require inductive weakening (e.g., as proposed in Sect. 4.3.3).

As in Sect. 4.2.3, our solution for under-approximating the existential quantifiers is based on a notion of MBP. Recall that for a formula $\exists\vec{y}\,.\,\varphi(\vec{x},\vec{y})$, an $MBP_{\vec{y}}$ is a function that given a model m, returns a $\vec{y}$-free formula, such that (1) m is also a model of the image $MBP_{\vec{y}}(m,\varphi)$, and (2) $MBP_{\vec{y}}(m,\varphi)$ is an under-approximation of $\exists\vec{y}\,.\,\varphi(\vec{x},\vec{y})$. In ProofAdapt, we exploit the properties of the $MBP_{\vec{y}}$ function to obtain $\widehat{\varphi}$ as a disjunction of projections $\widehat{\varphi} = \bigvee_{i=1}^{n} MBP_{\vec{y}}(m_i,\varphi)$ based on different models $m_1,\ldots,m_n$ for some finite number n. In practice, it is often more efficient to chose a relatively small and bounded n and make the obtained $\widehat{\varphi}$ inductive by applying an algorithm for inductive weakening (as in Sect. 4.3.3).

4.3.5 Strengthening Inductive Invariants without Quantifier Elimination

Here we focus on describing the algorithm Verify for strengthening an inductive invariant $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$ for the program Q that is used in ProofAdapt (line 7). Alternatively to perform quantifier elimination (as described in Sect. 4.3.4), we can substitute the quantified formulas directly to the system of linear Horn Clauses that describes the verification condition of Q (recall Def. 19). This idea comes from a simple observation that adding invariants to the transition relation does not affect any behavior of the program. In the nutshell, the invariants are additional constraints about pre- and post-states of each cutpoint edge $(u,v)\in E$.

Given $(u,v)\in E$ and $\tau : E \to Expr(V(u)\cup V'(v))$, let $\widehat{\tau} : E \to Expr(V(u)\cup V'(v))$ denote the relation constrained by the invariants $\exists\langle\sigma,\rho_\alpha\rangle|\psi|$, i.e.:

$$\begin{aligned}\widehat{\tau}(u,v) = {} & \exists\vec{y}\,.\,\big(\rho_\alpha(u,\sigma(u))(\vec{x},\vec{y}) \wedge \psi(u)(\vec{x})\big) \wedge \tau(u,v))(\vec{x},\vec{x}') \wedge \\ & \exists\vec{y}'\,.\,\big(\rho_\alpha(v',\sigma(v'))(\vec{x}',\vec{y}') \wedge \psi(v')(\vec{x}')\big)\end{aligned} \tag{4.12}$$

It is easy to see that a program $\widehat{Q} = \langle Vars, CP, en,\ err, E, \widehat{\tau}\rangle$ is equivalent to program $Q = \langle \mathit{Vars}, \mathit{CP}, en, err, E, \tau \rangle$, and the proof $\widehat{\psi}$ of Q is sufficient for $\widehat{Q}$. However, the the opposite is not true, i.e., a proof $\widehat{\psi}$ of $\widehat{Q}$ might not be sufficient for Q.

The algorithm Verify reduces the task of obtaining $\widehat{\varphi}$ to solving a system of appropriate linear Horn Clauses. This system consists of the rules (4.1) and (4.2). The quantifier elimination is done lazily inside the solving engine. Notably, the algorithm is also applicable in cases when $\exists \langle \sigma, \rho_\alpha \rangle |\psi|$ is not only inductive, but also safe. If so, a constant mapping $\widehat{\varphi}(u,v) = \top$ for any (u,v) is a solution for the Horn system, and solving terminates immediately.

4.3.6 Calculating the Change Impact

In case when ProofAdapt (and in turn Verify) cannot prove safety (i.e., fails to strengthen the inductive invariant), it generates a so called *change impact* – an indication whether the change of the code in a particular edge of the CPG broke the proof. Change impact can be calculated cheaply as a by-product of checking whether an abstraction αP of P is $\widehat{\psi}$-safe for the proof $\widehat{\psi}$.

Definition 26. *Given P, Q, a proof $\widehat{\psi}$ of P and abstraction αP of P such that $Q \preceq_{\sigma,\rho_\alpha} \alpha P$, the change impact δ of program Q is a mapping $\delta : E_Q \to \{\top, \bot\}$ such that for each $(u,v) \in E_Q$:*

$$\delta(u,v) \equiv \begin{cases} \top & \text{if } \psi(\sigma(u)))(\vec{x}) \wedge \tau_\alpha(\sigma(u), \sigma(v)))(\vec{x}, \vec{x}') \implies \psi(\sigma(v')))(\vec{x}') \\ \bot & \text{else} \end{cases} \tag{4.13}$$

If calculated this way, the change impact is precise enough to indicate all cutpoint-edges that are responsible for a property violation. Let us denote the set of edges $\Delta = \{(u,v) \in E_Q \mid \delta(u,v) = \bot\}$. In order to fix the given bug, the encoding τ of some of the cutpoint-edges from Δ must be rewritten, but the encoding τ of the edges in $E_Q \setminus \Delta$ can remain unchanged. In other words, program Q can be used to create a *partial program* Q_Δ that preserves the encoding of the edges $E_Q \setminus \Delta$ and contains *holes* to represent the absence of the encoding of Δ. Then, such a partial program Q_Δ is given as input to a program synthesizer, such as Sketch [128] to automatically find instantiations of the holes. In our future work we plan to integrate an automatic program repairer with ProofAdapt.

4.4 Related Work to Incremental Verification

FIV aims at automated establishing the equivalence of programs with respect to some common property since it has a direct application in model checking. In this section, we give a brief overview of the related formal methods excluding simulation relation synthesis (that is shown in Sect. 3.4).

There are techniques to establish a stronger property, *absolute equivalence* (i.e., equivalence of programs with respect to any possible property) [43; 85; 67]. The first automatic solutions to equivalence checking date back to hardware verification. Based on BDDs and SAT solving, the methods [16; 27; 91; 89; 31; 110] aim at searching for a counter-example witnessing inequivalence of the two circuits. Most of them exploit structural similarities between the circuits that make them able to scale well with the circuit size.

A step in incremental verification towards software was made in [43] that proposed to check equivalence of a Verilog circuit and a C program through encoding and solving a quantifier-free SAT formula. A more recent solution [67] employs BMC to establish absolute equivalence between C programs. The method traverses the call graph bottom-up and separately checks whether identity of inputs implies identity of outputs for each pair of matched (e.g., by name) functions, while all the nested calls are abstracted using the same uninterpreted function. A similar but language-agnostic approach is implemented in the SYMDIFF tool [85].

The problem of checking non-absolute equivalence between programs was addressed in a number of works, e.g., [20; 14; 10; 137]. The main motivating idea behind this line of research is the ability of reusing efforts between verification runs, thus achieving performance speedup compared to verification of programs separately. The closest approach to PDE is function-summarization-based Incremental BMC (EVOLCHECK), discussed in details in Chap. 2. Recall that EVOLCHECK extracts the over-approximating function summaries from the one program satisfying the given property, and then re-checks if the summaries still over-approximate function behavior in another program. PDE and EVOLCHECK are the complementing approaches in a sense that the former is designed to handle programs with complicated loop structures, but ignoring function calls, and the latter requires loop-free programs, but exploits their call trees. Furthermore, the approaches rely on conceptually different computation engines: SAT solving and interpolation versus SMT and Horn solving.

The earlier attempts to reuse information learned during analysis of the previous program version were employed in [35; 75; 46]. The approaches in [75; 46] store the entire abstract reachable state space and revalidate the affected parts after a change. In [35], the authors study substitutability of updated components of a system. Their algorithm is based on inclusion of behaviors and uses a CEGAR loop combining over- and under-approximations of the component behaviors. First, a containment check is performed, ensuring behaviors of the old component occur also in the new one. Second, learning-based assume-guarantee reasoning algorithm is used to check compatibility, i.e., that the new component satisfies a given property when the old component does. When compared, our approach focuses on real low-level properties of code expressed as assertions rather than abstract inclusion of behaviors.

The idea of "precision reuse" between software versions for CEGAR-based verification also appeared later in [20]. However, this technique does not relies on automatically derived relational specifications (like the mapping between variables or function names) so it requires revalidating the artefacts migrated from the verified programs. In contrast, the technique presented in this chapter benefits from using certified simulation relations between programs, thus confirming that the migrated invariants are always sound.

Alternatively, there are approaches [94; 137] to reason not only about differences between behaviors, but also to analyze differences between properties in different programs. The technique called Verification Modulo Versions (VMV) [94] transforms assertions from one program into assumptions for another program. VMV then tries to find (or prove absence of) bugs that are present only in the latter program. The technique called Directed Incremental Symbolic Execution (DISE) [137] based on Differential Symbolic Execution [112] is driven by the change impact which in fact is the program slice obtained by symbolic execution of the syntactic delta between the programs. The change impact is, however, property-independent, so DISE still requires further analysis whether the requested properties hold or do not in both programs. In contrast, the change impact calculated by ProofAdapt is always property-dependent that makes it useful to identify program locations that are likely responsible for the property violations.

4.5 Summary of Contributions

In this chapter, we contributed a FIV technique for Unbounded Model Checking (right branch in Fig. 1.1) that searches for safe inductive invariants in programs with (possibly infinite) loops. We formalized the concept of the Property-Directed Equivalence between programs that allows migrating safe inductive invariants across program transformations. We proposed the algorithm that performs an iterative abstract-refinement reasoning to automatically derive simulations relations between programs with different loop structures. If one program does not simulate another one, we proposed to automatically detect an abstraction of the former that simulates the latter. In contrast to the use of simple existential abstraction as in Chap. 3, we proposed to derive abstractions directly from invariants. Finally, we presented the new algorithm PROOFADAPT to establish PDE through combining synthesis of safe abstractions, simulation relations and safe inductive invariants. Our algorithm tries to adapt as much information from the invariant as possible, and then (if needed) strengthens the adapted invariant using an induction-based model unbounded model checker.

We implemented SIMABS and PROOFADAPT within an LLVM-based framework NIAGARA that extends the state-of-the-art unbounded model checker UFO [3; 86]. We evaluated SIMABS by discovering simulation relations between programs from Software Verification Competition and their LLVM optimizations. Our results show that SIMABS is able to efficiently synthesize abstractions and simulations between original and optimal programs in both directions. The results of SIMABS were further used to evaluate PROOFADAPT confirming that establishing PDE can be made more efficient than verification of both programs from scratch. More details on the evaluation can be found in Sect. 5.3.

Chapter 5

Tool Support

This chapter discusses the evaluation of the FIV techniques proposed in the previous chapters. We implemented the interpolation-based function summarization, refinement, automatic recursion depth detection and automatic assertion implication detection (all described in Sect. 2.2) in a tool called FUNFROG extending the CBMC [40] model checker. The incremental BMC algorithm (described in Sect. 2.3) is implemented in a tool called EVOLCHECK extending the FUNFROG model checker. Both tools, FUNFROG and EVOLCHECK, rely on the SAT solver PERIPLO [119], extending the OPENSMT solver [29] used both for satisfiability checks and interpolation. Note that PERIPLO is using only propositional (QF-BOOL) logic of OPENSMT which allows bit-precise reasoning. The tools are implemented using the CPROVER framework which takes care of preprocessing (by means of the tool GOTO-CC), parsing, intermediate representation of the program, and simple program analyses such as pointer aliasing and constant propagation. CPROVER is a mature code base and a part of CPROVER is available in the standard Debian distribution. The tools were evaluated on the range of academic and industrial benchmarks provided by the PINCETTE EU project[1].

We implemented the algorithms for simulation relation synthesis SIMABS and establishing a Property Directed Equivalence PROOFADAPT (described in detail in Chap. 3 and Chap. 4, respectively) on the top of the LLVM-based unbounded model checker UFO [3; 86] (which takes care of preprocessing) and the SMT solver Z3 [106]. We evaluated SIMABS by discovering total simulation relations between programs and their LLVM optimizations. Our results show that SIMABS is able to efficiently synthesize abstractions and simulations between original and

[1]http://www.pincette-project.eu

optimal programs in both directions. The application of SIMABS, however, is not limited to optimizations. It is able to deal with any program transformations preserving the program loop structures. In addition to checking optimizations, we also applied it to mutation testing. The results of SIMABS were further used to evaluate PROOFADAPT. PROOFADAPT confirmed that establishing PDE between two programs from Software Verification Competition can be made more efficient than verification of both programs from scratch.

The results reported in this chapter have been published in the following papers: [60] (co-authored with Natasha Sharygina), [56] (co-authored with Andrea Callia D'Iddio, Antti Eero Johannes Hyvärinen and Natasha Sharygina), [59] (co-authored with Ondrej Sery and Natasha Sharygina), [55] (co-authored with Arie Gurfinkel and Natasha Sharygina) and [57] (co-authored with Arie Gurfinkel and Natasha Sharygina).

5.1 FunFrog Bounded Model Checker

The basic architecture of FUNFROG is depicted in Fig. 5.1. The tool takes a C program with inlined assertions representing assertions and uses the **parser** for pre-processing. The parser produces an intermediate code representation, which is then used for encoding into a PBMC formula by **PBMC encoder**. Encoding is achieved using **symbolic execution**, which unwinds the program and prepares its static single assignment (SSA) form. Then FUNFROG choses the order, in which all assertions will be checked: by assertion implication detection, randomly or manually specified by the user and checks one assertion at a time. Then, for each assertion, the **SSA slicing** removes the SSA steps irrelevant to the assertion, and **SAT flattening** produces the final formula by encoding it into propositional logic. FUNFROG loads function summaries from a persistent storage and attempts to use them during encoding as over-approximations of the corresponding program functions. The tool passes the resulting formula to a **solver**. If the formula is unsatisfiable, the program is safe and FUNFROG uses interpolation to generate new function summaries and stores them for use in later runs. In order to control the strength and size of summaries, FUNFROG specifies the algorithm for interpolation for PERIPLO (e.g., *McM*, *Pud*, or *McP*) before the actual interpolation. In case of a satisfiable formula, FUNFROG asks **refiner** whether a refinement is necessary and if so, it continues by precisely encoding the functions identified by the refiner. If a refinement is impossible (there is no function to be refined) then

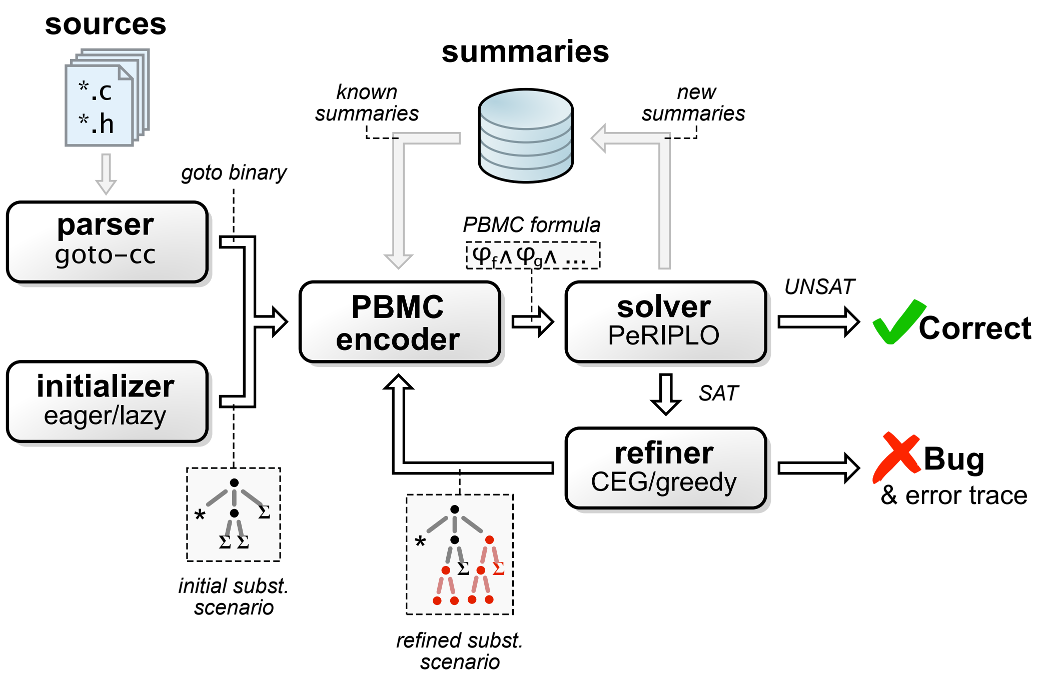


Figure 5.1. FUNFROG architecture overview

the counter-example is real, and the program is proven unsafe. In the following, we describe each step of FUNFROG in more details.

Parsing. As the first step, the source code is parsed and transformed into a *goto-program*, where the complicated conditional statements and loops are simplified using only guards and goto statements. For this purpose, FUNFROG uses GOTO-CC , i.e., a parser specifically designed to produce intermediate representation suitable for formal verification. Other tools from the CPROVER framework can be used to alter this representation. For example, GOTO-INSTRUMENT injects additional assertions (e.g., array bounds, division by zero, arithmetic overflow and underflow tests) to be checked during analysis.

Symbolic execution. In order to unwind the program, the intermediate representation is symbolically executed tracking the number of iterations of loops. The result of this step is the SSA form of the unwound program, i.e., a form where every variable is assigned at most once. This is achieved by adding version numbers to the variables. In FUNFROG, this step is also influenced by the choice of an *initial substitution scenario*. Intuitively, it defines how different functions should be encoded (e.g., using precise encoding, using a summary or treated nondeterministically).

Slicing. After the symbolic execution step, slicing is performed on the resulting SSA form. It uses dependency analysis in order to figure out which variables and instructions are relevant for the assertion being analyzed. The dependency analysis also takes summaries into account. Whenever an output variable of a function is not constrained by a function summary, its dependencies need not be propagated and a more aggressive slicing is achieved.

SAT flattening. When the SSA form is pruned, the PBMC formula is created by flattening into propositional logic. The choice of using SAT allows for bit-precise reasoning. However, in principle, the SAT flattening step could be substituted by encoding into a suitable SMT theory that supports interpolation.

Solving. The PBMC formula is passed to a SAT solver to decide its satisfiability. FunFrog uses PeRIPLO for both satisfiability checks and as an interpolating engine. Certain performance penalties follow from the additional bookkeeping in order to produce a proof of unsatisfiability used for interpolation.

Summaries extraction. For an unsatisfiable PBMC formula, FunFrog asks PeRIPLO to extract function summaries using interpolation over the proof of unsatisfiability. For this task, PeRIPLO uses the algorithm chosen prior (i.e., *McM*, *Pud*, or *McP*). The extracted summaries are serialized in a persistent storage so that they are available for other FunFrog runs. In this step, FunFrog also compares the new summaries with any existing summaries for the same function and the same bound, and keeps the more precise one.

Refiner. The refiner is used to identify summaries and nondeterministically treated function calls directly involved in the error trace. In the next iteration, the corresponded function calls will be encoded precisely. We call this strategy CEG (Counter-Example-Guided). This strategy is also used to iteratively detect a recursion depth. Alternatively, in case there is no recursive function calls, the refiner can avoid identification of too weak abstractions in the error trace (greedy strategy). Greedy strategy falls back to the standard BMC, encoding precisely all the function calls of the program in order to prove the assertion.

Eclipse plug-in. In order to make the tool as user-friendly as possible, we integrated FunFrog in the Eclipse development environment in the form of a plug-in. For a user, developing a program using the Eclipse IDE, the FunFrog plug-in makes it possible to verify different assertions of the single version of the code. Graphical capabilities of Eclipse contain a variety of helpers, allowing configuration of the verification environment.

The plug-in is developed using Plug-in Development Environment, a tool-set to create, develop, test, debug, build and deploy ECLIPSE plug-ins. It is built as an external jar-file, which is loaded together with ECLIPSE. The plug-in follows the paradigm of Debugging components, and provides the separate perspective, containing a view of the source code and visualization of the error traces computed for each violated assertion of the program. At the low level, the plug-in delegates the verification tasks to the corresponding command line tool FUNFROG. It maintains a database and external file storage to keep goto-binaries, summaries and other meta-data.

The binaries of FUNFROG were compiled using GCC version 4.8.2. The implementation of FUNFROG is totally deterministic and not parallelized. All the experiments were run on the Ubuntu 14.04 LTS machine, Intel(R) Xeon(R) CPU E5620 2.40GHz, 16 GB RAM. The verification time in all tables corresponds to a single verification run of the correspondent program.

In the rest of the section we present the evaluation of FUNFROG for different experimental settings. In Sect. 5.1.1, we evaluate Alg. 1 to verify a single assertion in each benchmark and detects the recursion depth. In Sect. 5.1.2, we evaluate Alg. 2 to verify a set of assertions and reuse function summaries between the algorithm runs. Finally, in Sect. 5.1.3, we evaluate Alg. 3 to detect the assertion implication relation and exploit this for Alg. 2.

5.1.1 Evaluating Recursion Depth Detection

We evaluated Alg. 1 on a set of various recursive C programs, adapted from the `SVCOMP'14`[2] set (`Ackermann X` `McCarthy`, `GCD`, `EvenOdd`), obtained from industry[3] (`P2P_Joints X`), and crafted by USI students for evaluation of interpolation-based abstractions.

Table 5.1 summarizes the verification statistics for a set of benchmarks. Each row corresponds to a program with one of the following type (**T**) of recursion: **a** - single recursion, **b** - multiple recursion, **c** - indirect recursion. The number of recursive functions present in each program is shown in the column #**R**. Each program was verified using CBMC, FUNFROG[4] without recursion depth detection and 3 different versions of FUNFROG+RDD. The first configuration of FUNFROG+RDD

[2]`http://sv-comp.sosy-lab.org/2014/`

[3]in scope of FP7-ICT-2009-5 — project PINCETTE 257647

[4]CBMC (standard distribution, version 4.3) was run with default parameters.

benchmark				FunFrog+RDD											FunFrog	CBMC
				In≡1					$1 < \text{In} < \nu$			In≡ ν				
name	#R	T	Result	In	Time	#It	ν	#Calls	In	Time	#It	In	Time	#It	Time	Time
`Array A`	5	a	SAFE	1	664.02	15	15	75	10	513.986	6	15	121.381	1	3600+	3600+
`Array B`	12	a	SAFE	1	777.432	24	24	71	2	1781.92	23	24	3600+	—	3600+	3600+
`Array C`	3	a	SAFE	1	1113.68	27	16	106	14	991.724	3	16	557.281	1	3600+	3600+
`Ackermann A`	2	b	SAFE	1	55.758	34	20	2169	7	3493.64	10	20	3600+	—	3600+	3600+
`Ackermann B`	2	b	BUG	1	56.772	30	17	1942	7	3547.29	10	17	3600+	—	3600+	3600+
`Alternate A`	2	c	SAFE	1	35.068	50	50	100	30	22.206	20	50	0.902	1	3600+	3600+
`Alternate B`	2	c	BUG	1	92.314	77	77	154	50	53.315	28	77	1.681	1	3600+	3600+
`Multiply`	10	a	SAFE	1	710.517	110	10	110	7	569.559	4	10	226.659	1	3600+	3600+
`InterleaveBitsRec`	1	a	SAFE	1	150.053	33	33	33	15	125.241	19	33	8.188	1	3600+	3600+
`BitShiftRec A`	1	a	SAFE	1	128.074	64	64	64	20	13.416	45	64	2.413	1	3600+	3600+
`BitShiftRec B`	2	b	SAFE	1	65.537	12	12	4285	3	65.399	10	12	3600+	–	3600+	3600+
`P2P_Joints A`	1	a	SAFE	1	1234.71	4	4	4	2	1195.31	3	4	1092.26	1	3600+	3600+
`P2P_Joints B`	1	a	BUG	1	1266.38	4	4	4	2	1222.11	3	4	1120.03	1	3600+	3600+

Table 5.1. Verification statistics for various BMC tools with and without automated detection of recursion depth.

performs the algorithm with the initial recursion depth set to 1 (denoted as **In ≡ 1** in the table), detects recursion depth (ν) and also reports the number of unwound recursive calls as #**Calls**. Then, in purpose of comparison, the second and the third configurations perform the same algorithm with the another values of the initial recursion depths (**1 < In <** ν and **In** ≡ ν respectively). For each experiment, we report total verification time (**Time** in seconds) and a number of iterations of FunFrog+RDD (#**It**). The verification results (SAFE/BUG) were identical for experiments with all configurations and we placed them in the table in the section describing the benchmarks.

Note that for all different types of recursion, the experiments with CBMC and pure FunFrog failed as they reached the timeout (**3600+**) of 1 hour without producing the result. This in general was not a problem for any of the experiments when FunFrog+RDD was used. We compare different configurations of FunFrog+RDD in order to demonstrate possible behaviors of FunFrog+RDD depending on the structure of benchmarks. The benchmarks `Multiply`, `Alternate A/B`, `Array A/C`, `InterleaveBitsRec` and `BitShiftRec A` witness the overhead of the procedure. In `InterleaveBitsRec` and `BitShiftRec A` there is a single recursive function called one time; in `Multiply` and `Alternate A/B` there are several recursive calls requiring the same recursion depth; in `Array A` and `Array C` there are several recursive calls requiring different, but relatively close recursion depths. That is, if we compare the first configuration with the third one, we can see that such overhead exists. The first configuration takes more time to complete verification than the second one, and the second configuration takes more time to complete verification than the third one. This is because FunFrog+RDD executes more iterations in the first configuration than in the second one and more iterations in the second configuration than in the third one. Again, the difference and the advantage is in the fact that the first and the second configurations do not know the recursion depth needed for verification and the third one gets it provided (as an initial recursion depth for FUNFROG+RDD). Therefore, for the third configuration it is always enough to execute one iteration.

The benchmarks `Array B`, `Ackerman A/B` and `BitShiftRec B` show the opposite behavior, where the first configuration takes less time to complete than the second and the third ones. These cases demonstrate the benefits of using *minimality* feature of the FUNFROG+RDD, since they require different recursion depths for each recursive function call appearing in the code. In all configurations we specify **In** by a fixed number which may fit well some of the recursive calls, but for other ones it may be bigger than needed. In this case, FUNFROG+RDD creates unnecessary PBMC partitions, blows up the formula and consequently slows down the verification process. While using **In** $= 1$, incremental unwinding automatically finds depths for each recursive function call. It means that for such cases the new approach for BMC not only detects the recursion depth sufficient for verification but that it also performs it efficiently and allows to slice out parts of the system which are redundant for verification purpose.

Interesting results are demonstrated by experimentation with the industrial benchmark `P2P_Joints A/B`. It contains expensive nonlinear computations, a complex call tree structure with relatively trivial recursion requiring unrolling 4 times. The experiments show that the difference in timings between different FUNFROG+RDD configurations is minor.

5.1.2 Evaluating Summarization-Based Recursion Depth Detection

Another set of experiments of verifying recursive programs by applying FUNFROG +SRDD is summarized in Table 5.2. There are two configurations of `FunFrog` compared in the table. The first one, **FunFrog+RDD**, is similar to the first configuration in Table 5.1. The second one, **FunFrog+SRDD**, is SRDD driven by *assertion decomposition*.

We explain the idea of assertion decomposition on the example from Fig. 2.1. The assertion `assert(y >= 0)` ($A1$) can be used to derive a set the following *helper*-assertions `assert(x < 5 || y >= 0)` ($A2$), `assert(x < 7 || y >= 0)` ($A3$) and so on. It is clear that if $A1$ holds, then both $A2$ and $A3$ hold as well; and

benchmark					FunFrog+RDD			FunFrog+SRDD				
name	#R	T	Result	v	In	TotalTime	#It	In	#A	TotalTime	ItpTime	#It
`Arithm`	1	a	SAFE	100	1	128.47	100	1	20	9.676	2.036	119
`McCarthy`	2	b	SAFE	11	1	3600+	—	1	5	10.495	4.859	24
`GCD`	3	b	SAFE	11	1	145.381	64	1	4	54.185	0.409	37
`EvenOdd`	2	c	SAFE	25	1	38.621	50	1	8	27.99	4.49	82
`P2P_Joints C`	1	a	SAFE	4	1	1531.38	4	1	4	1151.72	68.10	4
`P2P_Joints D`	1	a	SAFE	4	1	1192.28	4	1	4	1089.04	87.08	4

Table 5.2. Verification statistics of FunFrog+RDD and FunFrog+SRDD

if $A2$ holds then $A3$ holds as well. We will say that $A3$ is *weaker* than $A2$, and $A2$ is *weaker* than $A1$.

In this experiment, we derive helper-assertions (number of them is denoted **#A** in the table) by guessing values of the input parameters of recursive functions, then order assertions by strength and begin verification from the weakest one. If the check succeeds, the summaries of all (even recursive) functions are extracted. They will be reused in verification of stronger assertions. This procedure is repeated until the original assertion is proven valid. We summarize total timings (**TotalTime**) for verification of each weaker assertion, which includes the timings for interpolation (**ItpTime**).

For all benchmarks in the table, FunFrog+SRDD outperforms FunFrog+RDD. Technically, it means that checking a single assertion may be slower than checking itself and also several other assertions. The strongest result, we obtained, is verifying a well-known `McCarthy` function. Running FunFrog+SRDD for it takes around 10 seconds, while FunFrog+RDD, pure FunFrog and CBMC exceed timeout. Note that the interpolation may take up to a half of whole verification time. In some cases, summarization increases the number of iterations. But in total, FunFrog+SRDD remains more efficient that FunFrog+RDD.

5.1.3 Evaluating Assertion Implication Checking

Dynamic analysis tools such as Daikon [54] are often used for producing assertions. Such tools observe program behavior to form a set of the expressions over values of the program variables, which is then turned into a set of assertions V. Since the assertions are obtained by monitoring the execution of the program over a limited set of input parameters, there is no guarantee that such assertions hold for every execution of the program. BMC is used in [111] to check which of those assertions hold. While precise, a model checking run might consume a significant amount of time and require high amounts of memory. Therefore any optimization in the process immediately renders the technique more applicable to a wider set of benchmarks.

The assertion implication relation (AIR) can be used for various BMC applications that deal with large sets of assertions. In this section, we present two of those applications, namely *Optimizing Assertion Checking Order* and *Assertion Implication Checking in Function Summarization*.

In the first application, FUNFROG checks the validity of a set of assertions. We determine whether the number of verification runs can be reduced by skipping assertions whose validity is implied by already performed checks and AIR.

In the second application, FUNFROG constructs *function summaries* based on a set of assertions. We study whether excluding weak assertions using AIR reduces the size of function summaries.

Optimizing Assertion Checking Order

We propose two complementing strategies for the efficient detection of assertions which hold in the program. Given the SSA form U of a program that contains a set of assertions $V \subseteq U$, let $G_U = (V, E)$ be its assertion implication graph. We denote the nodes of G_U that do not have incoming edges as $\{A_s\}$. These correspond to the *strongest* assertions in the program. Similarly, we denote the edges with no outgoing edges as $\{A_w\}$, and these correspond to the *weakest* assertions in the program.

In the first (*forward*) strategy, FUNFROG traverses G_U starting from $\{A_s\}$ in the depth-first order. For each assertion node A_i, if there exists a holding predecessor A_j, the BMC tool concludes that A_i also holds. Otherwise, it verifies the program with respect to A_i. This strategy is efficient in cases when there are many holding assertions in the program.

In the second (*backward*) strategy, FUNFROG traverses G_U in reverse, starting from $\{A_w\}$. For each assertion node A_k, if there exists a failing successor A_j, the BMC tool concludes that A_k also fails. Otherwise, it verifies the program with respect to A_k. This strategy is efficient in cases when there are many assertions which fail in the program.

We evaluated the performance of FUNFROG+AIR in summarization-based BMC on a range of academic and industrial benchmarks widely used in model checking experiments. The assertions for the benchmarks were obtained from the user,

Benchmark name	#SSA Steps	# Asserts	# Checks	Strategy	#AIR Impl	AIR Time	BMC +AIR	Pure BMC
`token_ring`	11769	108	34	F	90	36.5	312.4	498.0
`mem_slave`	2843	146	116	F	61	24.6	70.9	108.9
`ddv`	537	152	103	F	93	14.9	162.1	240.2
`diskperf`	1730	192	34	B	172	75.8	65.5	332.5
`s3`	1733	131	47	B	265	4.4	20.6	55.5
`cafe`	2686	146	101	B	97	42.2	216.3	301.8

Table 5.3. Verification of a set of assertions by FunFrog and FunFrog+AIR. The timing values are given in seconds.

the BCT dynamic assertion generator [97] that internally uses Daikon [54], and static invariant synthesizers implemented in FunFrog.

We report the effect of exploiting AIR on the assertion checking in Table 5.3. In the experiment we are given a benchmark (represented as a SSA form with the corresponding **#SSA Steps**) and a set of assertions (**#Asserts**). First, FunFrog+AIR constructs the AIR (that reveals **AIR Impl** implications and takes **AIR Time** (*excluded from* **FunFrog+AIR**)). Then, FunFrog+AIR proceeds to assertion verification following one of the two strategies (**Strategy** = **F** (forward) or **B** (backward)), in which **#Checks** was actually performed. Finally, we compare the time spent on verification by **FunFrog+AIR** with the time needed to verify each assertion by the pure **FunFrog**. The assertions for these benchmarks come from the BCT tool.

In all our benchmarks FunFrog+AIR is able to reduce the total number of checks needed to perform the verification. In the best case scenario we observe run times that are more than two times faster than the pure FunFrog (see, `diskperf`). Note that for benchmarks containing many redundant assertions it is possible to detected more implications than the number of existing assertions in the code. For example, the benchmark instance `s3` has 131 assertions but over 200 implications. We illustrate the redundancy of assertions in Fig. 5.2 showing the assertion implication relation computed for the benchmark instance `mem_slave`.

Assertion Implication Checking in Function Summarization

The assertion implication graph $G_U = (V, E)$ can be used to reduce the size of function summaries. We propose to construct each subset $\{\mathcal{A}\}_0^k$ of V while traversing G_U. The method is based on the following observation. If each asser-

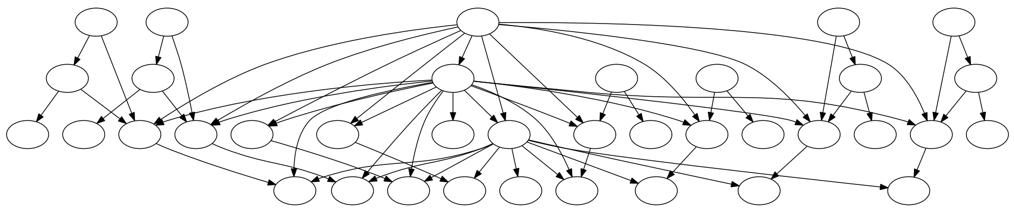


Figure 5.2. The assertion implication relation for benchmark instance `mem_slave`. Note that the figure only contains assertions that imply another assertion.

					AIR Summary		Pure Summary	
Benchmark name	#SSA Steps	# Asserts	#AIR Impl	AIR Time	# Vars	# Clauses	# Vars	# Clauses
`diskperf`	6000	47	7	0.083	150413	49362	162902	83625
`gd_simp`	673	21	5	0.138	6091	15420	12119	33504
`two_expands`	183	4	1	0.033	735	1221	1087	2277
`p2p_joints`	759	146	24	1.71	158034	452427	307897	902016
`goldbach`	7502	1344	65	25.82	6159	13455	13237	34689
`floppy`	15076	721	134	26.38	228357	11973	228659	12879

Table 5.4. Creation of function summaries by FUNFROG and FUNFROG+AIR. The timing values are given in seconds.

tion $A \in \mathcal{A}_i$ is implied by some assertion $A' \in \mathcal{A}_j$ then the summaries constructed from PBMC formula ϕ_j will be sufficient to prove both $\mathcal{A}_j$ and $\mathcal{A}_i$. On the other hand, if no implication is found between assertions $A \in \mathcal{A}_j$ and $A' \in \mathcal{A}_i$ then there is no guarantee that the summaries constructed from ϕ_j will be sufficient to prove $\mathcal{A}_i$. We propose to use the AIR to identify the set of strongest assertions and perform the verification only on this set. As a result we expect to obtain a strong summary that due to the simplicity of the resulting formula will be more compact (as our following experimental results confirm).

Table 5.4 reports statistics on constructing function summaries with FUNFROG when AIR was used as a preprocessor. Similarly to the experiment from Sec. 5.1.3, we are given a benchmark (with the size of **#SSA Steps**) and a set of assertions (**#Asserts**). First, FUNFROG+AIR constructs the AIR (with **AIR Impl** relations). Then FUNFROG+AIR obtains the set of strongest assertions to be encoded to the BMC formula, solved and used to create function summaries. Finally, we calculate the total number of variables and clauses in the resulting summary formula (**#Vars** and **#Clauses** respectively). We compare these values with the ones collected after the pure FunFrog run (**FunFrog** time, **#Vars'** and

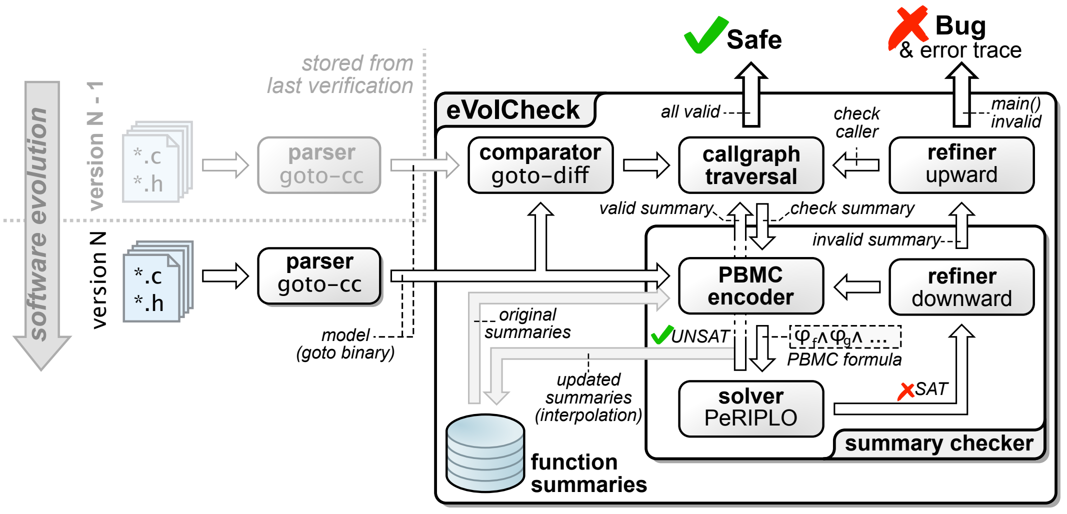


Figure 5.3. EVOLCHECK architecture overview

#Clauses' respectively). For these benchmarks we obtained the assertions using the GOTO-INSTRUMENT library inherited by FUNFROG from CPROVER.

The experimentation demonstrates that on our benchmark set the proposed approach improves the performance and the effect of BMC in the context of interpolation-based function summarization. Using particular optimization techniques (i.e., threshold for assertion locations and timeout for implication checks), in many cases it was possible to reduce the overhead of performing the implication checks. Note that at least in these benchmarks the construction of AIR requires a considerably smaller amount of time than needed for the actual assertions checking in the classic BMC approach.

5.2 eVolCheck Incremental Bounded Model Checker

This section presents the architecture of the EVOLCHECK tool as depicted in Fig. 5.3. Each version of the analyzed software is compiled using GOTO-CC separately. The resulting models are stored for future checks. The EVOLCHECK tool itself consists of a **comparator**, a **call graph traversal**, an **upward refiner** and a **summary checker**. The comparator identifies the changed functions calls. Note that if a function call was newly introduced or removed (i.e., the structure of the call graph is changed), it is considered as change in the parent function call. The call graph traversal attempts to check summaries of all the modified function calls bottom up. The upward refiner identifies the parent function call to be rechecked when a summary check fails. The summary checker performs the actual check of a function call against its old summary. In turn, it consists of a PBMC encoder that takes care of unwinding loops, generation of the SSA form and bit-blasting, a solver wrapper that takes care of communication with the solver/interpolator (PERIPLO), and a downward refiner that identifies ancestor functions to be refined when a summary check fails possibly due to imprecise representation of the ancestor function calls. Additionally, there are two optional optimizations in EVOLCHECK, namely **slicing** and **summary optimization**. The meaning of these optimizations exactly the same as their counterparts for the FUNFROG model checker.

Goto-diff. For comparing the two models, of the previous and the modified versions, we implemented a tool called GOTO-DIFF. The tool accepts two goto-binary models and analyzes them function by function. The longest common subsequence algorithm is used to match the preserved instructions and to identify the changed ones.

It is crucial that GOTO-DIFF works on the level of the models rather then on the level of the source files, i.e., performing not only Syntactic but also Semantic Diff. This way, it is able to distinguish some of the inconsequential changes in the code, for example, changes in the order of function declarations and definitions, text changes in comments and white spaces, and simpler cases of refactoring. These changes are usually reported as semantic changes by the purely syntactic comparators (e.g., the standard GNU diff tool). Moreover, as GOTO-DIFF works on the goto-binary models (i.e., after the C pre-processors) it correctly interprets also changes in the pre-processor macros.

Solver and interpolation engine. EVOLCHECK requires a solver that is able to generate multiple interpolants with the tree interpolant property from a single satisfiability query. For this reason, we use the interpolating solver, PERIPLO, which creates multiple interpolants from the same unsatisfiability proof and provides API for convenient specification of the partitions corresponding to the functions in the call tree.

Eclipse plug-in. We developed an EVOLCHECK-plug-in for the ECLIPSE IDE. It provides a capability to verify changes as part of the development flow for each version of the code. If the version history of the program is empty, the bootstrapping (initial verification) is performed first. Otherwise, EVOLCHECK verifies the program with respect to the last safe version.

Table 5.5. eVolCheck experimental evaluation.

Benchmark		Bootstrapping		Incremental check						
Name	Inst [#]	Itp [s]	Total [s]	Diff [#]	Diff [s]	Itp [s]	Total [s]	Inv [#]	Total [#]	Speedup [X]
floppy 1	2434	1.455	556.231	21	1.188	0	1.304	0	187	223.21
p2p 1	276	0.633	76.884	0	0.018	id	0.018	0	8	4271.33
p2p 3	358	0.498	40.618	1	0.02	0.277	10.453	0	20	3.88
arith 36	60	20.047	40.53	2	0.001	5.997	7.663	2	5	5.29
arith 31	51	12.134	33.043	1	0.001	1.119	1.509	1	4	21.88
kbfiltr 1	1024	0.369	31.828	1	0.072	0.004	0.113	0	56	172.04
kbfiltr 1	1024	0.371	31.813	2	0.071	0.004	0.24	1	56	102.29
life 1	118	3.137	30.9	2	0.004	err	18.757	4	30	1.65
arith 2	64	8.123	26.121	2	0.002	0.52	0.927	2	5	28.12
arith 6	78	9.914	22.287	3	0.001	2.791	4.227	3	6	5.27
arith 20	70	9.83	22.125	3	0.002	3.478	4.607	3	6	4.80
arith 24	61	8.844	21.234	2	0.001	17.898	33.008	3	5	0.64
arith 19	61	5.561	21.159	1	0.002	0.434	0.571	2	5	36.93
euler 1	85	0.742	19.439	1	0.001	0.147	0.678	1	11	28.63
diskperf 1	538	0.449	19.301	1	0.027	0.014	0.183	0	19	91.91
diskperf 2	535	0.447	19.134	1	0.026	0.259	11.326	2	19	1.69
diskperf 3	523	0.4	19.017	2	0.025	0.285	11.298	2	19	1.68
arith 7	100	1.518	12.784	3	0.001	2.847	14.881	7	9	0.86
p2p 2	355	0.493	6.595	0	0.02	id	0.02	0	9	329.75
floppy 3	323	0.161	3.677	1	0.029	0.003	0.07	0	18	37.14
floppy 2	320	0.16	3.675	1	0.028	0.003	0.07	0	18	37.50
diskperf 1	1880	0.124	3.149	1	0.114	0.001	0.151	1	5	11.88
floppy 4	322	0.088	2.127	2	0.028	0	0.101	0	7	16.49
floppy 5	330	0.089	1.895	5	0.028	0.082	2.041	0	7	0.92

5.2.1 Evaluating eVolCheck

Similarly to FunFrog, the binaries of eVolCheck were compiled using gcc version 4.8.2. The implementation of eVolCheck is totally deterministic and not parallelized. All the experiments were run on the Ubuntu 14.04 LTS machine, Intel(R) Xeon(R) CPU E5620 2.40GHz, 16 GB RAM. The verification time in the table corresponds to a single verification run of the correspondent pair of programs.

To demonstrate the applicability and advantages of eVolCheck, we provide evaluation details of several test cases. The benchmarks `p2p_`*n* were provided by our industrial partners for which the changes were extracted from the project repositories. `diskperf_`*n*, `floppy_`*n*, `kbfiltr_`*n* were derived from Windows device driver library. The changes (with different level of impact, from adding an irrelevant line of code to moving a part of functionality between functions) were introduced manually there. The rest of the benchmarks are crafted programs with arithmetic computations. Pre-processing the code with the goto-cc tool generated a collection of goto-binaries that were then processed with eVolCheck focusing the validation to particular functional sub-projects.

Table 5.5 represents results of the experiments. Each benchmark is shown in a separate row, which summarizes statistics about the initial verification and the incremental verification. **Inst [#]** measures the original source code size as a number of instructions in the goto-binary (this is the more accurate parameter than the usual "lines of code" since it does not contain declarations of the variables, empty lines of code and so on). **Total [#]** represents the number of function calls in the program, and **Diff [#]** – the number of changed function calls, identified by GOTO-DIFF.

Time (in seconds) for running GOTO-DIFF (**Diff [s]**) and for generation of the interpolants (**Itp [s]**) represents the computational overhead of the incremental checking procedure, and included to the total running time (**Total [s]**) of EVOLCHECK. Note that interpolation can not be performed for the buggy program transformations (marked as "err"), for which the corresponded PBMC formula is satisfiable; or for the identical program versions (marked as "id"), for which GOTO-DIFF returned empty set of changed function calls.

To show advantages of EVOLCHECK, for each change we calculated the speedup (**Speedup**) of the incremental check versus standalone verification of the changed code from scratch, performed only for the sake of comparison and thus not shown in the table. Finally, the number of invalidated summaries (due to the change) is listed in **Inv [#]** column.

Discussion. Our evaluation demonstrates good performance of EVOLCHECK. In particular, the experiments show high effect of incremental checking for safe program transformations since they result in a small number of refinements (both, upward and downward). Moreover, if the changed function are located deeper in the call tree, this generally leads to a small number of invalidated summaries, as witnessed by the ratio **Inv [#] / Total [#]**. For example, consider the case, where summaries of all changed functions were proven valid.

EVOLCHECK is less efficient in case of program transformations, which affect a large amount of function calls located on the different levels of the call tree. When the transformation introduced a bug, it will cause an increasing amount of upward refinements. However, sometimes even a single buggy change introduced in a higher level of the call tree, might be verified efficiently.

In classical model checking, confirming the absence of bugs is usually more expensive (since it requires the full state-space search) then detecting the bugs (where the search can be terminated once a counter-example is detected). On the contrary, EVOLCHECK is less time- and resource-demanding in proving the absence of bugs. Thus, it might make sense to run eVOLCHECK and a classical model checker (e.g., FUNFROG) in parallel, and terminate both processes whenever one of them returned a result.

The use of GOTO-DIFF has been particularly useful since it managed to detected test cases with small syntactic changes which did not require running the main eVOLCHECK procedures. For example, in `p2p 1/2`, the comparator proved that the models are identical (however, the source code might still be different), so no further checking was needed.

As expected, in the majority of the experiments, the localized incremental check provided by eVOLCHECK outperforms the verification from scratch, which is indicated by **speedup** > 1. Moreover, in many instances (usually on large industrial cases) the speedup is large, which demonstrates good efficiency and usefulness of the tool.

In future we plan to conduct a case study where the incremental BMC is applied to a realistic line of revisions (e.g., several dozens of successive program versions from a repository) for a given project.

5.3 Niagara Framework for Simulation Discovery and Proof Lifting

We present NIAGARA, an LLVM-based framework that contains an implementation of SIMABS, a novel iterative abstraction-refinement algorithm to find an abstraction of the target T that simulates the source S, and an implementation of PROOFADAPT, a novel algorithm to use a proof of T and a simulation relation between S and T to obtain a proof of S.

The architecture of SIMABS is illustrated in Fig. 5.4(a). Initially, in the **Synthesize** step, SIMABS guesses a relation ρ between S and T. Then, in the **Solve** step, SIMABS checks whether T simulates S via ρ. If the check fails, SIMABS iteratively performs the **Abstract** step to find an abstraction αT of T and a simulation relation ρ_α between S and αT. Finally, in the **Refine** step, SIMABS refines both αT and ρ_α. The algorithm terminates when either no refinement or no abstraction is possible. The search space of the algorithm is shown in Fig. 5.4(b): SIMABS explores the space of abstractions of T, starting with the most general abstraction αT that simulates S via ρ_α, and iteratively refines it to $\alpha^{(n)}T$ that simulates S via $\rho_{\alpha^{(n)}}$.

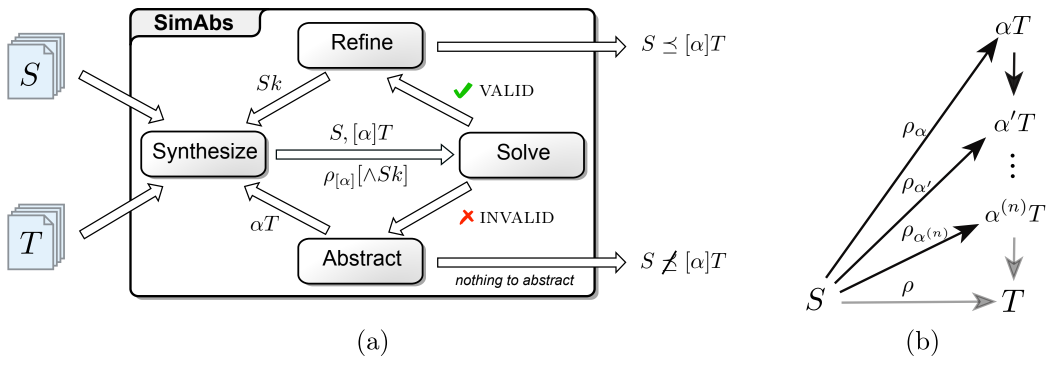


Figure 5.4. (a) SIMABS and (b) its search space.

PROOFADAPT takes as input the programs S and T, a proof φ of T and an abstraction $\alpha^{(n)}T$ of T that simulates S via $\rho_{\alpha^{(n)}}$. First, it weakens φ to become an inductive invariant $\lfloor\varphi\rfloor$ for $\alpha^{(n)}T$. Second, it adapts $\lfloor\varphi\rfloor$ to become an invariant of S using $\rho_{\alpha^{(n)}}$. Finally, it strengthens the adapted invariant to become safe using the underlying model checker.

We implemented SIMABS, PROOFADAPT and AE-VAL on the top of the state-of-the-art unbounded model checker UFO [3; 86] . For solving Horn clauses, AE-VAL uses PDR engine [80] implemented in Z3 [106]. Our tool works with LLVM-preprocessed abstractions of programs in which procedures are inlined, memory is lowered to registers, and other memory and procedures are treated as non-deterministic. While aggressive, this abstraction is precise enough to preserve properties of Linux Device Drivers in Software Verification Competition.

UFO takes care of the preprocessing the input programs, extracting a CPG-representation, synthesizing a safe inductive invariant and serializing all these data into an external storage. NIAGARA restores the invariants and the verification conditions and uses them as inputs to SIMABS and PROOFADAPT. As output, SIMABS and PROOFADAPT provide (whenever possible) a simulation relation and a new safe inductive invariant, that are also serialized for possible future runs.

For the experiments, we used the version of UFO based on LLVM-2.9, and compiled UFO and NIAGARA using GCC version 4.7.3. The implementation of NIAGARA is totally deterministic and not parallelized. All the experiments were run on the Ubuntu 12.04 LTS machine, Intel(R) Core(TM) i7-3740QM CPU 2.70GHz, 12 GB RAM.

5.3.1 Evaluating SimAbs

We evaluated SimAbs on the SVCOMP benchmarks and `constprop`, `globalopt`, `instcombine`, `simplifycfg`, `adce`, and `mem2reg` optimizations of LLVM. The `constprop` performs constant propagation, the `globalopt` transforms global variables, the `instcombine` simplifies local arithmetic operations, the `simplifycfg` performs dead code elimination and basic block merging, the `adce` performs aggressive dead code elimination, and `mem2reg` promotes memory references to be register references. Notably, combinations of the optimizations provide more aggressive optimizations than each individual optimization, thus increasing a *semantic gap* between the original and the optimized programs. In our evaluation, we aim at synthesizing concrete or abstract simulation relations for programs with a bigger semantic gap and empirically demonstrate the power of AE-VAL (that is expected to have a higher number of AE-VAL iterations in such cases).

For each of the 228 considered source programs S (300 - 5000 lines of source code), we created an optimized program[5] T, and applied SimAbs to discover abstractions and simulations in two directions: $S \preceq [\alpha]T$ and $T \preceq [\alpha]S$. We present the results in two diagrams in Fig. 5.5. Each diagram is a pie chart and a collection of SimAbs execution times for each benchmark in the spherical coordinate system. The pie chart in Fig. 5.5(a) represents a *proportion* of four main classes of SimAbs results:

◷: T simulates S via identity relation;

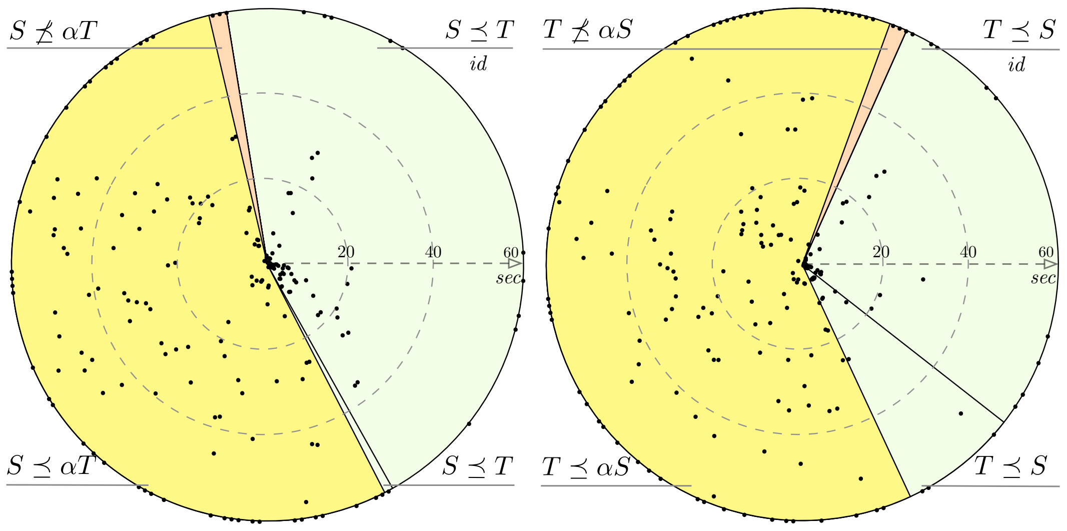
◶: T simulates S via some Skolem-relation-based ρ;

◵: some abstraction αT simulates S;

◴: we did not find an abstraction αT that simulates S.

Each dot represents a runtime of SimAbs on a single benchmark. It is placed in one of the circular sectors, ◷, ◶, ◵ or ◴, with respect to the outcome, and is assigned the radial distance to represent time in seconds. For example, a benchmark on which $S \preceq_{id} T$ solved in 20 seconds is placed in the sector ◷ in a distance 20 from the center. Being closer to the center means being faster. Runs that took longer than 60 seconds are placed on the boundary. Fig. 5.5(b) is structured similarly, but with inverse order of S and T.

[5]We combined the optimizations in the following order to create each T: `-constprop -globalopt -instcombine -simplifycfg -mem2reg -adce -instcombine -simplifycfg`.



(a) Simulation of S by T (b) Simulation of T by S

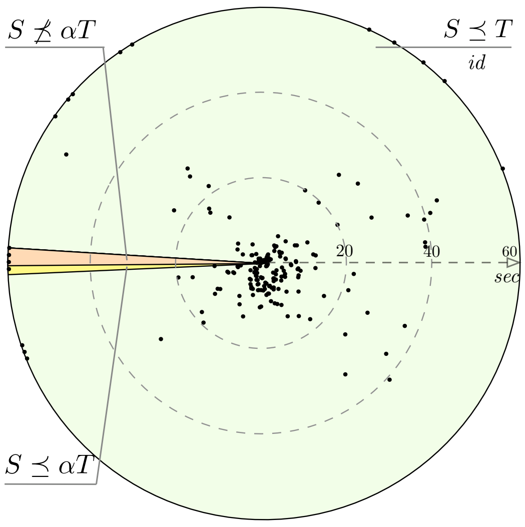
Figure 5.5. Pie chart and running times in the spherical coordinate system.

The experiment shows that SIMABS is able to effectively discover abstractions and simulations between S and T in both directions. While in many cases (101 in Fig. 5.5(a), and 65 in Fig. 5.5(b)) it proved simulation by identity, in the remaining cases SIMABS goes deeper into the abstraction-refinement loop and delivers either a concrete or abstract simulation in 124 and 160 cases respectively. SIMABS terminates with a positive result in all, but 3 pairs of programs. The 3 negative cases can be explained by the fact that T happened to have some CPG-edge (u, v) with the inconsistent labeling $\tau_T(u, v)$.

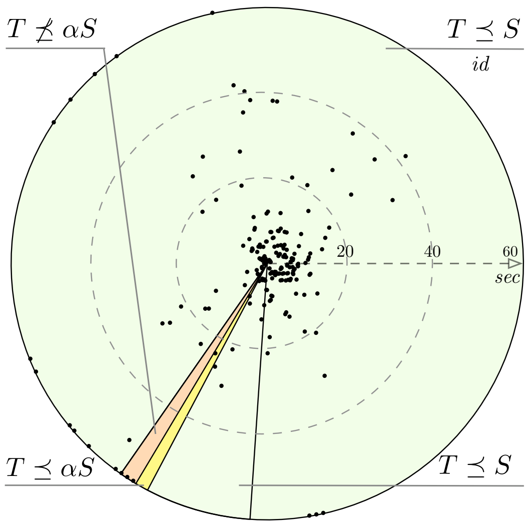
The core solving engine, AE-VAL, invoked on the low level of SIMABS was shown to be effective while eliminating quantifiers. Overall, it solved 84587 formulas (each formula contains up to 1055 existentially quantified variables, and requires up to 617 iterations to terminate), and extracted 3503 Skolem relations.

Unfortunately, during the experiment, we were not able to compare SIMABS with other techniques. There were several practical difficulties. First, SIMABS works at the level of LLVM, and we are not aware of other incremental-verification-like tools at LLVM level except of [64]. They, however, require the programs to contain assertions and furthermore, they require all the assertions to be safe. As stated in Sect. 3.1.3, SIMABS does not have such restrictions.

There are some tools at the C level [92; 67; 61]. [92] is not available even in binary form, [67] is based on BMC and cannot handle infinite loops, [61] can



(a) Simulation of S by T



(b) Simulation of T by S

Figure 5.6. `constprop, globalopt, adce`

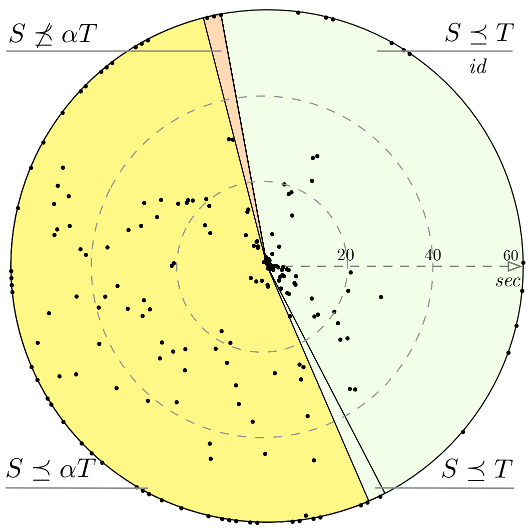
only handle very small subset of C. Furthermore, we focus on the case when programs are not equivalent (may not even simulate each other). So these tools are not applicable. For example, neither one allows for programs to be non-deterministic, which they are in our case due to modelling abstraction, i.e., bit-vector operators, external function calls, etc.

Evaluating other optimizations. In addition to simulation discovery for the combination of optimizations (depicted in Fig. 5.5), we applied SIMABS to discover simulation relations between the source program and the target program resulted from individual (and therefore, less aggressive) optimizations.

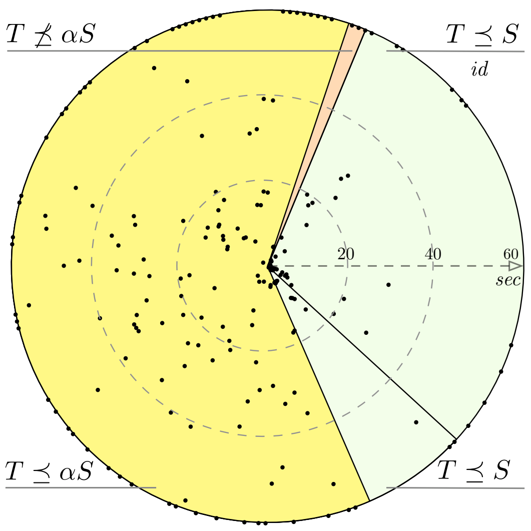
Fig. 5.6 depicts the effect of optimizations `constprop, globalopt, adce`. SIMABS was able to discover mutual simulation relation between S and T in 223 cases out of 228 ones. This can be explained by our intuition that the considered optimizations are relatively lightweight.

Fig. 5.7 depicts the effect of optimizations `instcombine, simplifycfg`. SIMABS performs similarly to the experiment in Fig. 5.5, but it was able to discover 3 more total simulation relations in the direction $S \preceq T$ and 2 more total simulation relations in the direction $T \preceq S$.

Our experiments for discovering simulation relations individually for `instcombine`, were identical to the case of `instcombine, simplifycfg`. We therefore omit

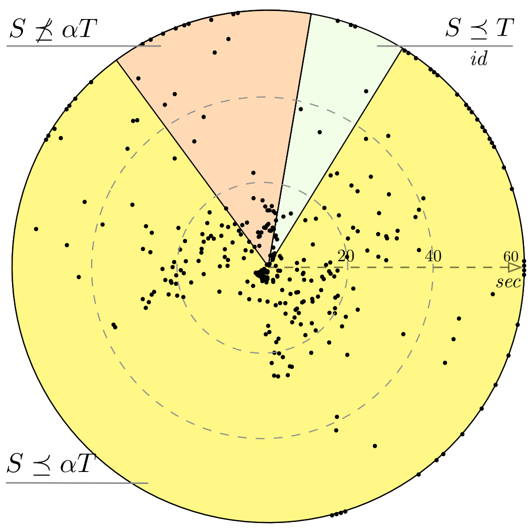


(a) Simulation of S by T

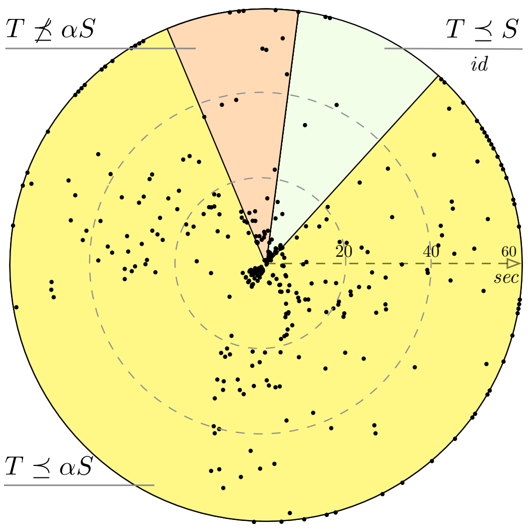


(b) Simulation of T by S

Figure 5.7. `instcombine, simplifycfg`



(a) Simulation of S by T



(b) Simulation of T by S

Figure 5.8. Discovering simulation for program mutants.

presenting the corresponding figures.

Application to mutation testing. We demonstrate the applicability of SIMABS to a rather different scenario from proving correctness of compiler optimizations.

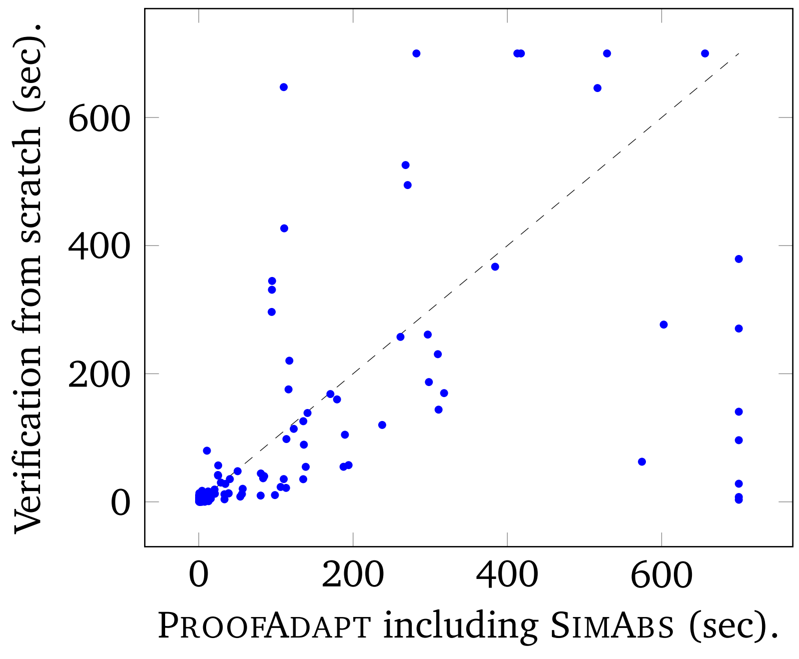
Mutation testing is used to evaluate the quality of program specification. The idea is to generate a so called *program mutant* using a small random change. It could change arithmetic operations (e.g., "+" to "-", as in Fig. 3.1(a)-3.1(b)). Finally, the goal of testing is to *kill* such mutant (i.e., to prove that it is inconsistent with the given specification), and therefore to demonstrate that the change brakes the specification.

We expand this idea and move it into the verification domain. More specifically, for a given program S, we generate a mutant program T and then proceed with synthesizing a simulation relation between S and T. The results of the experiment are shown in Fig. 5.8. SIMABS was applied to 352 pairs of S and T, but succeeded in synthesis of total simulation relation (i.e., $S \preceq T$) less than in 10% of cases (21 cases for $S \preceq T$ and 34 cases for $T \preceq S$). On the the other hand, SIMABS was unable to find any abstraction αT, such that $S \preceq \alpha T$ in more than in 8% cases (45 cases for $S \not\preceq \alpha T$ and 31 cases for $T \not\preceq \alpha S$). Both numbers significantly differ from the correspondent ones in the experiments with optimizations: number of $\star \preceq \star$ results was much larger, and the number of $\star \not\preceq \alpha\star$ was much lower. Intuitively, this means that our mutation was successful, and the correspondent program mutants were killed.

5.3.2 Evaluating ProofAdapt

For evaluation of PROOFADAPT, we used benchmarks from Software Verification Competition and the following LLVM-optimizations: `-indvars -loop-rotate -licm -loop-simplify -instcombine -simplifycfg -lowerswitch`. The `indvars` transforms the loops to have a single canonical induction variable initially assigned to zero and being incremented by one. The `loop-rotate` performs loop rotations. The `licm` detects loop invariants and moves them outside of the loop body. The `loop-simplify` transforms loops into a simpler form. The `simplifycfg` simplifies the control-flow structure, and is particularly effective after the arithmetic simplifications by the `instcombine`. Finally, the `lowerswitch` rewrites switch instructions. Notably, this combination of optimizations does not necessarily preserve the program's loop structure.

We focus our attention only on safe programs, i.e., those for which UFO is able to find a proof $\widehat{\psi}$. We further use $\widehat{\psi}$ to create a $\widehat{\psi}$-safe abstraction αT of each original program T and discover a simulation relation ρ_α between αT and the optimized program S. Both, proof $\widehat{\psi}$ and simulation relation ρ_α, are further



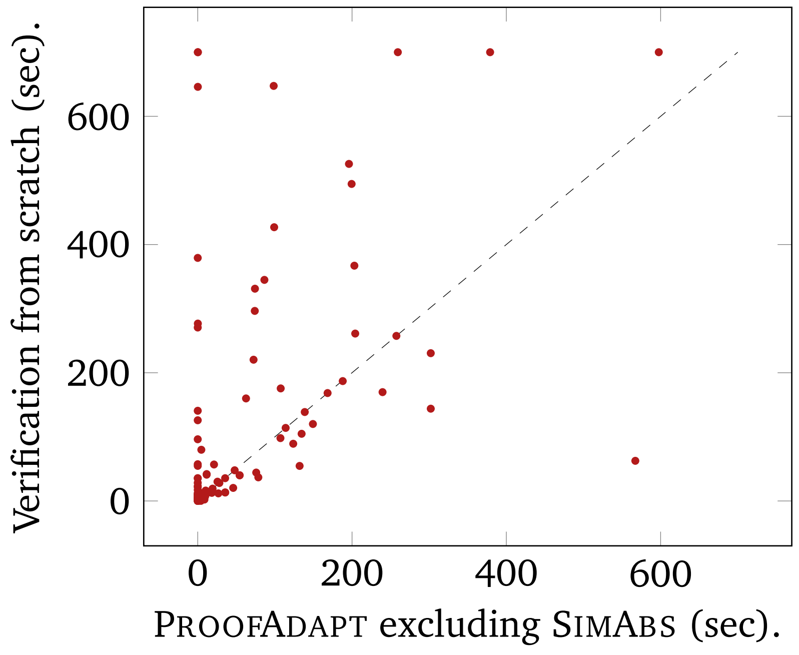


Figure 5.9. Verification by ProofAdapt (including and excluding SimAbs) compare to Verification from scratch.

used to produce a proof $\widehat{\varphi}$ of S.

The need to apply PROOFADAPT to establish equivalence between a program and its optimization comes from an unfortunate fact that an implementation of the optimization pass might contain bugs (as shown, e.g., in [96]). Thus, such program transformations (especially if they contain combinations of optimizations) could end up with the programs that are not semantically equivalent to the original programs. Establishing the Property Directed Equivalence is served to ensure that optimizations do not break the particular safety properties of the particular programs.

We compared the performance of PROOFADAPT with the performance of the stand-alone model checker UFO while verifying the optimized program from scratch (Fig. 5.9(a)). Of course, it would be too naive to expect that PROOFADAPT outperforms UFO in majority of cases, since PROOFADAPT requires an expensive run of SIMABS (which in turn proceeds by deciding validity of a sequence of $\forall\exists$-formulas). Thus, we additionally compared the performance of a lightweight version of PROOFADAPT (i.e., PROOFADAPT excluding SIMABS) with UFO (Fig. 5.9(b)). Having in mind that both abstraction and simulation might already be known before the run of PROOFADAPT, such application scenario is still practical.

Among 128 pairs of considered programs, and given a timeout of 700 sec, SIMABS was able to discover an abstraction and a simulation for all but 7 pairs. Among these 121 pairs, 14 abstractions were $\widehat{\psi}$-safe that allowed migrating the proof directly; the others required a further weakening-strengthening procedure.

PROOFADAPT outperformed UFO in 43 cases (including 5, in which UFO exceeded timeout). Notably, this includes the time for SIMABS that took in average 38% of the PROOFADAPT time. Another important observation is that PROOFADAPT tends to be better than UFO whenever the verification takes more time. That is, the average time for verifying the benchmarks won by PROOFADAPT is 137 sec, while the average time for verifying the benchmarks won by UFO is 100 sec.

The overall picture of the comparing PROOFADAPT (excluding SIMABS) versus UFO is even more impressive: PROOFADAPT outperformed UFO in 99 cases. In the remaining 29 cases, the performed optimizations dramatically simplified the program so it became easier to verify it from scratch.

To conclude, we must mention that being an SMT-based framework, NIAGARA currently supports only LRA that makes it difficult to evaluate programs handling arrays, floating point arithmetic, bit-vectors and so on. We look forward to enhance the main computation engine (mainly, MBPs) with more recent solutions; however, it slightly moves out of the scope of this dissertation. NIAGARA shown its potential to be the first framework that is able to connect reusable and relational specifications of the versioned software, and we envision multiple improvements of its workflow in future.

Chapter 6

Conclusions

In this dissertation, we investigated the new techniques to FIV that aims to automatically verify different versions of software. The main message, we would like the reader to take home, is that FIV techniques should not treat the underlying model checking engine as a black box, but instead should exploit all its subroutines in a certain duly justified manner. Only in this case FIV helps earning performance benefits without sacrificing soundness of the analysis. Our contributions extend three complementary approaches to automated formal verification, thus confirming that FIV capabilities can be implemented on the top of widely-used model checking tools.

In Chap. 2, we presented our FIV contributions for SAT-based BMC designed to verify programs with (possibly recursive) functions. We invented the function-summarization framework based on Craig interpolation that allows generating reusable specifications between verification runs. We introduced the algorithm to revalidate the summaries existing from the verification of one program version locally in order to prevent re-verification of another program version from scratch. In order to create function summaries of a better quality, we proposed to exploit the assertion implication relation during the BMC preprocessing. Additionally, we contributed in speeding-up BMC for an individual program version and finding a proper loop and recursion bound for BMC.

In Chap. 3, we presented our FIV contributions for simulation relation synthesis for loop-free programs, but does not require any assertions to be supplied. We introduced an SMT-based abstraction-refinement algorithm that proceeds by guessing a relation and checking whether it is a simulation relation for the given programs (called the source and the target, respectively). The problem of checking simulation is therefore reduced to deciding validity of a sequence of ∀∃-formulas. Our algorithm manipulates implicit abstractions of the target program by introducing existential quantifiers to the right-hand-side of the ∀∃-formulas. We presented the algorithm AE-VAL for deciding validity of ∀∃-formulas in LRA that extracts a Skolem relation to witness the existential quantifiers. This Skolem relation is the key to refine the considered abstractions of the target, and therefore requires to be minimized and factored. The results of AE-VAL are then used to strengthen both, the discovered simulation relation and the current abstraction of the target.

In Chap. 4, we presented our FIV contributions for Unbounded Model Checking designed to verify programs with (possibly, nested) loops. We formulated the challenge as establishing a *Property Directed Equivalence* (PDE) between programs and proposed a solution based on migrating safe inductive invariants across program versions. Our algorithm performs an iterative abstract-refinement reasoning to automatically derive simulations between programs with different loop structures. The key insight of the algorithm is it automatically derives an abstraction of the already verified program that is still precise enough to satisfy the existing proof. Finally, our algorithm uses the automatically synthesized simulation relation between the new (not verified) program version and the safe abstraction of the old (verified) program version to lift the existing proof, thus preventing generation of a new proof from scratch.

In Chap. 5, we presented the evaluation of all the algorithms we contributed. We implemented the interpolation-based function summarization, refinement, automatic recursion depth detection and automatic assertion implication detection in the tool FunFrog extending the CBMC model checker and the SAT solver PeRIPLO. We further extended FunFrog with the incremental BMC-based Upgrade Checking algorithm resulting in the tool eVolCheck. Both tools, FunFrog and eVolCheck, were evaluated on the range of academic and industrial benchmarks provided by the EU project PINCETTE.

We implemented the algorithm SimAbs for simulation relation synthesis and the algorithm ProofAdapt for establishing PDE in the framework Niagara extending the LLVM-based model checker UFO and the SMT solver Z3. We evaluated SimAbs by discovering total simulation relations between programs and their LLVM optimizations. The results of SimAbs were further used to evaluate ProofAdapt. ProofAdapt confirmed that establishing PDE between two programs from Software Verification Competition can be made more efficient than verification of both programs from scratch.

6.1 Future Work

We consider two major research directions that would enhance the contributions outlined in the dissertation. First, the function-summarization framework would benefit if its main computation engine turns to SMT. Second, our solution to PDE would benefit of the automated program repair capabilities.

6.1.1 SMT-Based Function Summarization

Function summarization (contributed in Sect. 2.2) is a generic technique for abstracting programs which allows both propositional and first-order instantiations. In this dissertation, we have focused on the propositional encoding of the BMC formulas to achieve bit-precise results. However, there are many cases where using a theory solver might result in a more efficient summary, without sacrificing the precision.

Our current implementation considers the bit-precise encoding of the program, encoding the sums and inequalities as addition circuits. This choice sometimes causes major problems due to the big sizes of the corresponding formulas. The big formulas often are expensive to solve making the overall verification procedure impractical in such cases. We believe that the function summaries resulting from theory interpolation would likely be significantly more compact.

In the potential future work, one can investigate constructing function summaries over theories including uninterpreted functions, difference logic, and linear arithmetics. We believe that the use of SMT will speed up the calculation and allow scaling the model checking applications to process much larger projects.

Many of the widely used theories in SMT have interesting properties with respect to over-approximating program behavior and in particular when computing function summaries. The most precise encoding in case of model checking is to express the algorithm as a bit-precise propositional satisfiability formula. However, as discussed above, there are several reasons to try to model a verification problem with less expensive theories.

In the potential future work, one can do fundamental research on algorithms that identify iteratively the function calls that violate an assertion, and substitute these functions with increasingly strong theory representation. We believe

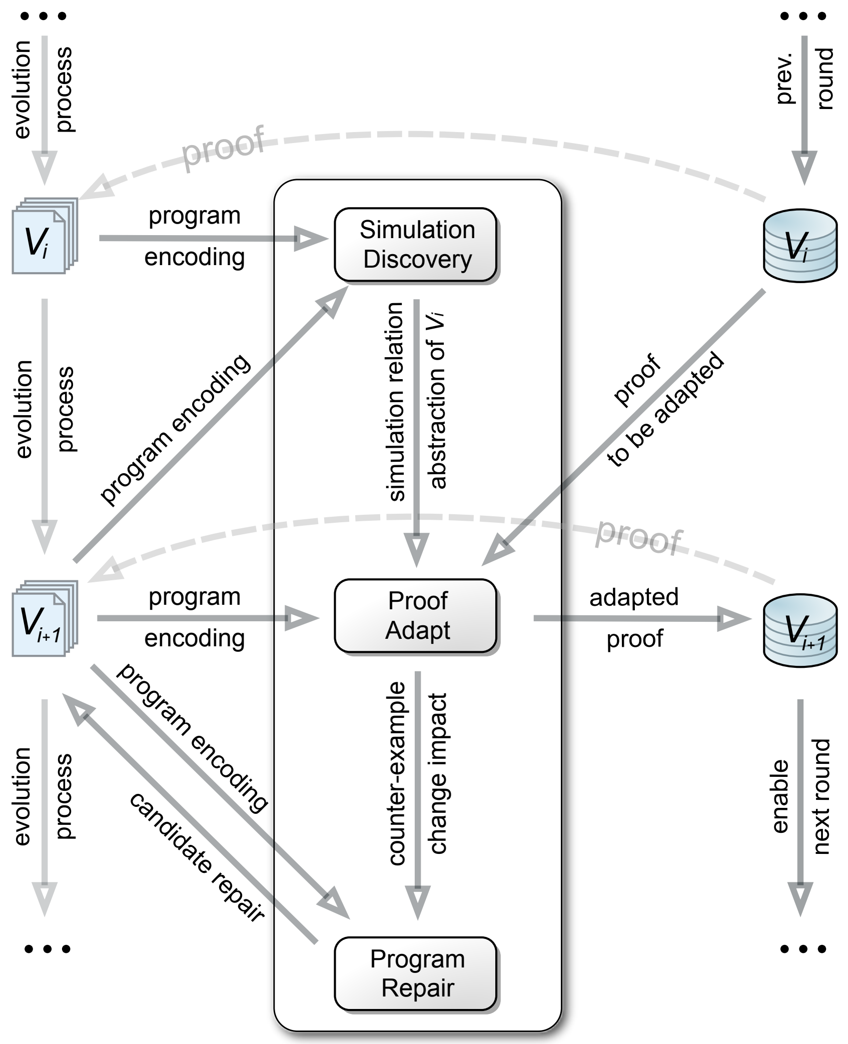


Figure 6.1. A round of NIAGARA for V_i and V_{i+1}.

that the use of this *theory-sensitive refinement* (as opposed to the straightforward propositional refinement described in Sect. 2.2.6) will allow us to automatically adapt the model so that the parts critical to correctness will be encoded in the coarsest precision required by the proof.

6.1.2 Automated Program Repair

We envision our FIV framework NIAGARA to combine efforts on unbounded proof-based upgrade checking and program repair. Its workflow outlined in Fig. 6.1 considers a transition within the evolutionary chain of a given program between versions V_i and V_{i+1}. NIAGARA obtains a simulation relation between versions, uses it to verify V_{i+1} by adapting the proof of V_i and (if disproven safe) attempts to incrementally fix the detected bug in V_{i+1}. Thus, NIAGARA iterates between the components **Simulation Discovery**, **Proof Adapt**, and **Program Repair**.

Simulation Discovery (already contributed and described in Sect. 4.2) receives the symbolic encoding of V_i and V_{i+1}; and discovers an abstraction αV_i of V_i that simulates V_{i+1} via some total relation ρ.

Proof Adapt (already contributed and described in Sect. 4.3) takes V_i, V_{i+1} simulated by some αV_i via ρ and the safety proof $\widehat{\pi}_{V_i}$ of V_i; and either adapts $\widehat{\pi}_{V_i}$ to become a proof $\widehat{\pi}_{V_{i+1}}$ of V_{i+1}, or provides a counter-example. In cases if V_{i+1} is buggy, NIAGARA also generate a *change impact* certificate (described in Sect. 4.3.6), a function labeling the CPG of V_{i+1} by propositional constants indicating whether the lifted inductive invariant in the correspondent cutpoint did (or did not) require strengthening. In other words, it is an indication whether the code in a particular program location is affected by the change or not.

Program Repair receives V_{i+1}, a counter-example and the change impact between V_i and V_{i+1}; and outputs a *candidate repair*, a new program version V'_{i+1} which does not contain bad behaviors. Our method aims to find an abstraction of V_{i+1} (where the code, not affected by the change, should be kept) such that it contains at least one safe behavior. This abstraction is further refined using the witnessing Skolem function to finally deliver the candidate repair. The candidate repair is model-checked, and, if another counter-example is found, NIAGARA iterates until it discovers the repair fixing each counter-example, or a user intervention is made.

We believe that the extension of NIAGARA to do **Program Repair** will have a broad impact and numerous applications including improving scalability of automated verification, simplifying software certification, and helping software engineers to produce new program versions of a higher quality. Furthermore, since the solutions are often applicable across domains, we hope that all our delivered and planned contributions will stimulate a new research and give rise to new thoughts and ideas helping to solve other problems involving verification and synthesis.

Bibliography

[1] ACM: Press Release on the 2007 A.M. Turing Award recipients (2007), `http://www.acm.org/press-room/news-releases-2008/turing-award-07`

[2] Albarghouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. In: ESOP. LNCS, vol. 9032, pp. 634–660. Springer (2015)

[3] Albarghouthi, A., Gurfinkel, A., Chechik, M.: UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In: CAV. LNCS, vol. 7358, pp. 672–678. Springer (2012)

[4] Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In: VMCAI. LNCS, vol. 7148, pp. 39–55. Springer (2012)

[5] Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: CAV. LNCS, vol. 8044, pp. 313–329. Springer (2013)

[6] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: LPAR. LNCS, vol. 7180, pp. 46–61. Springer (2012)

[7] Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A Proof-Sensitive Approach for Small Propositional Interpolants. In: VSTTE. LNCS, vol. 9593. Springer (2015)

[8] Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: CAV. vol. 3114, pp. 484–487. Springer (2004)

[9] Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: ICSE. pp. 211–220. ACM (2008)

[10] Backes, J.D., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: SPIN. LNCS, vol. 7976, pp. 99–116. Springer (2013)

[11] Balabanov, V., Jiang, J.R.: Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In: CAV. LNCS, vol. 6806, pp. 149–164 (2011)

[12] Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN. LNCS, vol. 1885, pp. 113–130. Springer (2000)

[13] Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. LNCS, vol. 2102, pp. 260–264. Springer (2001)

[14] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. LNCS, vol. 6664, pp. 200–214. Springer (2011)

[15] Basler, G., Kroening, D., Weissenbacher, G.: SAT-based summarization for Boolean programs. In: SPIN. LNCS, vol. 4595, pp. 131–148. Springer (2007)

[16] Berman, C.L., Trevillyan, L.H.: Functional comparison of logic designs for VLSI circuits. In: ICCAD. pp. 456–459. IEEE (1989)

[17] Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL. pp. 221–234. ACM (2014)

[18] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software Model Checking via Large-Block Encoding. In: FMCAD. pp. 25–32. IEEE (2009)

[19] Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011)

[20] Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Meyer, B., Baresi, L., Mezini, M. (eds.) ESEC/FSE. pp. 389–399. ACM (2013)

[21] Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58, 118–149 (2003)

[22] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)

[23] Bjørner, N., Gurfinkel, A.: Property Directed Polyhedral Abstraction. In: VMCAI. LNCS, vol. 8931, pp. 263–281. Springer (2015)

[24] Bodík, R., Jobstmann, B.: Algorithmic program synthesis: introduction. STTT 15(5-6), 397–411 (2013)

[25] Bourdoncle, F.A.: Efficient Chaotic Iteration Strategies with Widenings. In: FMPA. LNCS, vol. 735, pp. 128–141. Springer (1993)

[26] Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011)

[27] Brand, D.: Verification of large synthesized designs. In: ICCAD. pp. 534–537. IEEE (1993)

[28] Brugh, N.H.M.A.D., Nguyen, V.Y., Ruys, T.C.: MoonWalker: Verification of .NET Programs. In: TACAS. LNCS, vol. 5505, pp. 170–173. Springer (2009)

[29] Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: TACAS. LNCS, vol. 6015, pp. 150–153. Springer (2010)

[30] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers pp. 677–691 (1986)

[31] Burch, J.R., Singhal, V.: Tight integration of combinational verification methods. In: ICCAD. pp. 570–576. IEEE (1998)

[32] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: LICS. pp. 428–439. IEEE (1990)

[33] Cabodi, G., Murciano, M., Nocco, S., Quer, S.: Stepping forward with interpolants in unbounded model checking. In: ICCAD. pp. 772–778 (2006)

[34] Cabodi, G., Palena, M., Pasini, P.: Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking. In: FMCAD. pp. 43–50. IEEE (2014)

[35] Chaki, S., Clarke, E., Sharygina, N., Sinha, N.: Dynamic Component Substitutability Analysis. In: FM. LNCS, vol. 3582, pp. 512–528. Springer (2005)

[36] Christakis, M., Godefroid, P.: IC-Cut: A Compositional Search Strategy for Dynamic Test Generation. In: SPIN. LNCS, vol. 9232, pp. 300–318. Springer (2015)

[37] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD. pp. 165–168. IEEE (2013)

[38] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: TACAS. LNCS, vol. 8413, pp. 46–61. Springer (2014)

[39] Ştefan Ciobâcă, Lucanu, D., Rusu, V., Rosu, G.: A language-independent proof system for mutual program equivalence. In: ICFEM. LNCS, vol. 8829, pp. 75–90. Springer (2014)

[40] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)

[41] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000)

[42] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)

[43] Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC. pp. 368–371. ACM (2003)

[44] Clarke, E., Emerson, A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logics of Programs, Workshop. LNCS, vol. 131, pp. 52–71. Springer (1981)

[45] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

[46] Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In: CAV. LNCS, vol. 3576, pp. 449–461. Springer (2005)

[47] Cordeiro, L.C., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ANSI-C software. In: ASE. pp. 137–148. ACM (2009)

[48] Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. In: J. of Symbolic Logic. pp. 269–285 (1957)

[49] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: An efficient method of computing static single assignment form. In: POPL. pp. 25–35. ACM (1989)

[50] Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In: CAV. LNCS, vol. 575, pp. 255–265. Springer (1991)

[51] D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI. LNCS, vol. 5944, pp. 129–145. Springer (2010)

[52] Dutertre, B.: Solving Exists/Forall Problems With Yices. In: SMT Workshop (2015), extended abstract

[53] Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134. IEEE (2011)

[54] Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27(2), 99–123 (2001)

[55] Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Incremental verification of compiler optimizations. In: NFM. LNCS, vol. 8430, pp. 300–306. Springer (2014)

[56] Fedyukovich, G., Callia D'Iddio, A., Hyvärinen, A.E.J., Sharygina, N.: Symbolic Detection of Assertion Dependencies for Bounded Model Checking. In: FASE. LNCS, vol. 9033, pp. 186–201. Springer (2015)

[57] Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: LPAR. vol. 9450, pp. 606–621. Springer (2015)

[58] Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental Upgrade Checker for C. In: TACAS. LNCS, vol. 7795, pp. 292–307. Springer (2013)

[59] Fedyukovich, G., Sery, O., Sharygina, N.: Flexible SAT-based Framework for Incremental Bounded Upgrade Checking. STTT 17, 1–18 (2015)

[60] Fedyukovich, G., Sharygina, N.: Towards Completeness in Bounded Model Checking through Automatic Recursion Depth Detection. In: SBMF. LNCS, vol. 8941, pp. 96–112. Springer (2014)

[61] Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ASE. pp. 349–360. ACM (2014)

[62] Flanagan, C., Leino, K.R.M.: Houdini: an Annotation Assistant for ESC/Java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)

[63] Gascón, A., Tiwari, A.: A Synthesized Algorithm for Interactive Consistency. In: NFM. LNCS, vol. 8430, pp. 270–284 (2014)

[64] Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V.N., Zuck, L.D.: From verification to optimizations. In: VMCAI. LNCS, vol. 8931, pp. 300–317. Springer (2015)

[65] Godefroid, P.: Compositional dynamic test generation. In: POPL. pp. 47–54. ACM (2007)

[66] Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL. pp. 43–56. ACM (2010)

[67] Godlin, B., Strichman, O.: Regression verification. In: DAC. pp. 466–471. ACM (2009)

[68] Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: CAV. LNCS, vol. 1254, pp. 72–83. Springer (1997)

[69] Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. pp. 405–416. ACM (2012)

[70] Gurfinkel, A., Rollini, S., N.Sharygina: Interpolation Properties and SAT-Based Model Checking. In: ATVA. LNCS, vol. 8172, pp. 255–271. Springer (2013)

[71] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)

[72] Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL. pp. 471–482. ACM (2010)

[73] Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: FOCS. pp. 453–462 (1995)

[74] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. pp. 232–244. ACM (2004)

[75] Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme Model Checking. In: Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer (2003)

[76] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70. ACM (2002)

[77] Heule, M., Seidl, M., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: FMCAD. pp. 107–114. IEEE (2014)

[78] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)

[79] Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. Symposium on Semantics of Algorithmic Languages pp. 102–116 (1971)

[80] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT. vol. 7317, pp. 157–171. Springer (2012)

[81] Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. pp. 279–295 (1997)

[82] Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404(3), 256–274 (2008)

[83] Jancík, P., Kofron, J., Rollini, S.F., Sharygina, N.: On interpolants and variable assignments. In: FMCAD. pp. 123–130. IEEE (2014)

[84] John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem Functions for Factored Formulas. In: FMCAD. pp. 73–80. IEEE (2015)

[85] Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Tech. Rep. MSR-TR-2010-119, Microsoft Research (2010)

[86] Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: CAV. LNCS, vol. 8559, pp. 17–34 (2014)

[87] Kozen, D., Patron, M.: Certification of Compiler Optimizations Using Kleene Algebra with Tests. In: CL. LNCS, vol. 1861, pp. 568–582. Springer (2000)

[88] Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with Wolverine. In: CAV. LNCS, vol. 6806, pp. 573–578. Springer (2011)

[89] Kuehlmann, A., Krohm, F.: Equivalence checking using cuts and heaps. In: DAC. pp. 263–268. IEEE (1997)

[90] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013)

[91] Kunz, W.: Hannibal: An efficient tool for logic verification based on recursive learning. In: ICCAD. pp. 538–543. IEEE (1993)

[92] Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: FSE. pp. 345–355. ACM (2013)

[93] Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV. LNCS, vol. 7358, pp. 427–443. Springer (2012)

[94] Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: towards usable verification. In: PLDI. p. 32. ACM (2014)

[95] Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. 36(5), 450–462 (1993)

[96] Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably Correct Peephole Optimizations with Alive. In: PLDI. pp. 22–32. ACM (2015)

[97] Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. IEEE Transactions on Software Engineering 37(4), 486–508 (2011)

[98] McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV. LNCS, vol. 2725, pp. 1–13. Springer (2003)

[99] McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. LNCS, vol. 4144, pp. 123–136. Springer (2006)

[100] McMillan, K.L.: Lazy annotation for program testing and verification. In: CAV. LNCS, vol. 6174, pp. 104–118. Springer (2010)

[101] McMillan, K.L.: Lazy annotation revisited. In: CAV. LNCS, vol. 8559, pp. 243–259. Springer (2014)

[102] McMillan, K.L., Rybalchenko, A.: Solving constrained Horn Clauses using interpolation. Tech. Rep. MSR-TR-2013-6, Microsoft Research (2013)

[103] Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: VSTTE. LNCS, vol. 7152, pp. 146–161. Springer (2012)

[104] Milner, R.: An algebraic definition of simulation between programs. In: IJCAI. pp. 481–489 (1971)

[105] Monniaux, D.: A quantifier elimination algorithm for linear real arithmetic. In: LPAR. LNCS, vol. 5330, pp. 243–257. Springer (2008)

[106] de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)

[107] Namjoshi, K.S.: Lifting Temporal Proofs through Abstractions. In: VMCAI. LNCS, vol. 2575, pp. 174–188. Springer (2003)

[108] Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: SAS. LNCS, vol. 7935, pp. 304–323. Springer (2013)

[109] Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI. pp. 83–94. ACM (2000)

[110] Paruthi, V., Kuehlmann, A.: Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation. In: ICCD. pp. 459–464. IEEE (2000)

[111] Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehestedt, S., Muhammad, A.: Verification-aided regression testing. In: ISSTA. pp. 37–48. ACM (2014)

[112] Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: FSE. pp. 226–237. ACM (2008)

[113] Phan, A.D., Bjørner, N., Monniaux, D.: Anatomy of alternating quantifier satisfiability (work in progress). In: SMT. EPiC Series, vol. 20, pp. 120–130. EasyChair (2012)

[114] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)

[115] Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)

[116] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: ISP. pp. 337–351 (1982)

[117] Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. pp. 49–61. ACM (1995)

[118] Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: CAV. LNCS, vol. 9206, pp. 198–216. Springer (2015)

[119] Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: LPAR. LNCS, vol. 8312, pp. 683–693. Springer (2013)

[120] Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: CAV. LNCS, vol. 7358, pp. 193–209. Springer (2012)

[121] Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and Solving Horn Clauses for Verification. In: VSTTE. LNCS, vol. 8164, pp. 1–21. Springer (2013)

[122] Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-Clause verification. In: CAV. LNCS, vol. 8044, pp. 347–363. Springer (2013)

[123] Schwartz-Narbonne, D., Oh, C., Schäf, M., Wies, T.: VERMEER: A tool for tracing and explaining faulty C programs. In: ICSE. pp. 737–740. IEEE (2015)

[124] Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded model checking with interpolation-based function summarization. In: ATVA. LNCS, vol. 7561, pp. 203–207. Springer (2012)

[125] Sery, O., Fedyukovich, G., Sharygina, N.: Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In: FMCAD. pp. 114–121. IEEE (2012)

[126] Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC. LNCS, vol. 7261, pp. 160–175. Springer (2012)

[127] Skolem, T.: Über die mathematische logik. In: Norsk Matematisk Tidsskrift 10. pp. 125–142 (1928)

[128] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415. ACM (2006)

[129] Unno, H., Terauchi, T.: Inferring simple solutions to recursion-free Horn Clauses via sampling. In: TACAS. LNCS, vol. 9035, pp. 149–163. Springer (2015)

[130] Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: FMCAD. pp. 1–8. IEEE (2009)

[131] Vizel, Y., Gurfinkel, A.: Interpolating Property Directed Reachability. In: CAV. LNCS, vol. 8559, pp. 260–276. Springer (2014)

[132] Vizel, Y., Gurfinkel, A., Malik, S.: Fast interpolating BMC. In: CAV. LNCS, vol. 9206, pp. 641–657. Springer (2015)

[133] Vujosevic-Janicic, M., Kuncak, V.: Development and evaluation of LAV: an SMT-based error finding platform - system description. In: VSTTE. LNCS, vol. 7152, pp. 98–113. Springer (2012)

[134] Weiser, M.: Program slicing. In: ICSE. pp. 439–449. IEEE (1981)

[135] Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL. pp. 351–363. ACM (2005)

[136] Xie, Y., Aiken, A.: Saturn: A SAT-Based Tool for Bug Detection. In: CAV. LNCS, vol. 3576, pp. 139–143. Springer (2005)

[137] Yang, G., Khurshid, S., Person, S., Rungta, N.: Property differencing for incremental checking. In: ICSE. pp. 1059–1070. ACM (2014)