Finding inductive invariants using satisfiability modulo theories and convex optimization

Egor George Karpenkov

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

March 30, 2017

Présentée par

George Egor Karpenkov

Thèse dirigée par **Dr. David Monniaux**

préparée au sein du laboratoire **Verimag**
et de l'école doctorale **MSTII**

Finding Inductive Invariants Using Satisfiability Modulo Theories and Convex Optimization

Thèse soutenue publiquement le **29 mars 2017**,
devant le jury composé de :

Pr. Sylvie Putot
Professor, École Polytechnique, Rapporteur
Dr. Pierre-Loïc Garoche
Research Scientist, Onera, Rapporteur
Pr. Dirk Beyer
Professor, Ludwig-Maximilian University of Munich, Examinateur
Dr. Nikolaj Bjørner
Principal Researcher, Microsoft Research, Examinateur
Pr. Lydie du Bousquet
Professor, Université Grenoble Alpes, Présidente
Pr. Helmut Seidl
Professor, Technical University of Munich, Examinateur
Dr. David Monniaux
Directeur de Recherche, CNRS, Directeur de thèse

Remerciements

During my work on this thesis, I was lucky to benefit from help and advice from many wonderful people. Without them, this work would have never been finished.

For the detailed and meticulous proof-reading (many times over!) and providing many useful comments I would like to thank Grigory Fedyukovich and Alexey Bakhirkin.

The engineering contributions of this thesis greatly benefited from the guidance of Philipp Wendler, who has dispensed a large amount of invaluable advice on software systems architecture. This collaboration was possible due to the help of Dirk Beyer, who has kindly let me spend three months at his research group

Additionally, I would like to thank Tim King and Arie Gurfinkel for their time, and many insightful conversations explaining the nature of the scientific inquiry.

The material in the chapter on generating summaries with policy iteration was influenced by productive discussions with Peter Schrammel.

Additionally, I am grateful to my supervisor, David Monniaux, for many productive collaborations.

Finally, I am very indebted to love, support and patience I got from Kate and Andreas during these years, which I could not do with out. Thanks!

Abstract

Static analysis concerns itself with deriving program properties which hold universally for *all* program executions. Such properties are used for *proving* program properties (e.g. there never occurs an overflow or other runtime error regardless of a particular execution) and are almost invariably established using *inductive invariants*: properties which hold for the initial state and imply themselves under the program transition, and thus hold universally due to induction.

A traditional approach for finding numerical invariants is using *abstract interpretation*, which can be seen as interpreting the program in the *abstract domain* of choice, only tracking properties of interest. Yet even in the *intervals* abstract domain (upper and lower bounds for each variable) such computation does not necessarily converge, and the analysis has to resort to the use of *widenings* to enforce convergence at the cost of precision.

An alternative game-theoretic approach called *policy iteration*, *guarantees* to find the least inductive invariant in the chosen abstract domain under the finite number of iterations. Yet the original description of the algorithm includes a number of drawbacks: it requires converting the entire program to an equation system, does not easily integrate with other approaches, and does not directly benefit from known results for Kleene iteration (e.g. iteration order).

Our new algorithm for running *local* policy iteration (LPI) instead formulates policy iteration as traditional Kleene iteration, with a widening operator that *guarantees* to return the least inductive invariant in the domain after finitely many applications. Local policy iteration runs in *template linear constraint domains* which requires setting *in advance* the "shape" of the derived invariant (e.g. $x + 2y$ for deriving $x + 2y \leq 10$). Our second theoretical contribution involves development and comparison of a number of different template *synthesis* strategies, and their evaluation when used with LPI. Additionally, we present an approach for generating abstract reachability trees using abstract interpretation, enabling the construction of counterexample traces, which in turns lets us to generate new templates using Craig interpolants.

In our third contribution we bring our attention to interprocedural and potentially recursive programs. We develop an algorithm parameterizable with any abstract interpretation for *summary* generation, and we study it's parameterization with LPI. The resulting approach is able to generate least inductive invariants in the domain for a *fixed number of summaries* for recursive programs.

Our final theoretical contribution is a "formula slicing" method for finding potentially disjunctive inductive invariants from program fragments obtained by symbolic execution.

We implement all of these techniques in the open-source state-of-the-art CPAchecker program analysis framework, enabling collaboration between different analyses.

The techniques mentioned above rely on *satisfiability modulo theories* solvers, which are capable of giving solutions to first-order formulas over certain *theories* or showing that none exists. In order to simplify communication with such tools we present the JavaSMT library, which provides a generic interface for such communication. The library has shown itself to be a valuable tool, and is already used by many researchers.

Résumé

L'analyse statique correcte d'un programme consiste à obtenir des propriétés vraies de *toute* exécution de ce programme. Celles-ci sont utiles pour démontrer des caractéristiques appréciables du logiciel, telles que l'absence de dépassement de capacité ou autre erreur à l'exécution quelle que soient les entrées du programme. Elles sont presque toujours établies à l'aide d'*invariants inductifs* : des propriétés vraies de l'état initial et telles que si elles sont vraies à une étape de calcul, alors elles restent vraies à l'étape suivante de la transition de calcul, donc sont toujours vraies par récurrence.

L'*interprétation abstraite* est une approche traditionnelle de la recherche d'invariants numériques, que l'on peut exprimer comme une interprétation non-standard du programme dans un *domaine abstrait* choisi et ne tenant compte que de certaines propriétés intéressantes. Même dans un domaine aussi simple que les *intervalles* (un minorant et un majorant pour chaque variable), ce calcul ne converge pas nécessairement, et l'analyse doit recourir à des *opérateurs d'élargissement* pour forcer la convergence au détriment de la précision.

Une autre approche, appelée *itération de politique* et inspirée par la théorie des jeux, garantit de trouver le plus fort invariant inductif dans le domaine abstrait choisi après un nombre fini d'itérations. Cependant, la description originale de cet algorithme souffrait de quelques faiblesses: elle se basait sur une conversion totale du programme en un système d'équations, sans intégration ni synergie avec les autres méthodes d'analyse.

Notre nouvel algorithme est une forme *locale* de l'itération de politique, qui la replace dans l'itération de Kleene mais avec un opérateur d'élargissement spécial qui garantit d'obtenir le plus petit invariant inductif dans le domaine abstrait après un nombre fini de ses applications. L'itération de politique locale opère dans les domaines de contraintes linéaires données par patron, qui demandent de fixer *d'avance* la « forme » de l'invariant (p.ex. $x + 2y$ pour obtenir $x + 2y \leq 10$). Notre seconde contribution théorique est le développement et la comparaison de plusieurs stratégies de synthèse de patrons, utilisées en conjonction avec l'itération locale de politiques. De plus, nous présentons une méthode pour générer des arbres d'accessibilité abstraite par interprétation abstraite, permettant la génération de traces de contre-exemples, et ensuite la génération de nouveaux patrons à partir d'interpolants de Craig.

Notre troisième contribution concerne l'analyse interprocédurale de programmes, éventuellement récursifs. Nous proposons un algorithme qui génère pour chaque procédure un *résumé*, applicable à toute interprétation abstraite et notamment à l'itération de politique locale. Nous pouvons ainsi générer les invariants inductifs les plus forts dans le domaine pour un nombre fixé de résumés pour un programme récursif.

Notre dernière contribution théorique est une méthode d'affaiblissement permettant de trouver des invariants inductifs, éventuellement disjonctifs, à partir de formules obtenues par exécution symbolique.

Nous avons mis en œuvre toutes ces approches dans le système d'analyse statique CPAchecker, un logiciel libre, ce qui permet des communications et collaborations entre analyses.

Nos techniques utilisent des résolveurs de satisfiabilité modulo théorie, capables, étant donné une formule de logique du premier ordre sur certaines théories, d'en donner un modèle ou de démontrer qu'aucun n'existe. Afin de simplifier les communications avec ces outils, nous présentons la bibliothèque JavaSMT, fournissant une interface générique. Cette bibliothèque a déjà démontré son utilité pour de nombreux chercheurs.

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Preliminaries

Chapter 1

Introduction

1.1 Motivation: Software Systems Complexity

Every day computerized systems are becoming more ubiquitous in human life. While just two decades ago computers were thought of as stationary standalone devices dedicated to a particular task, today it is difficult to find a device which does *not* have a computer inside. Computerized devices exist on all scales and in all areas of the industry, starting from phones and Raspberry Pi chips, to rockets and nuclear power stations.

The complexity and the scope of those systems has experienced a dramatic growth, which magnifies the possibility of damage from errors in the software. Traditionally, quality assurance was performed by extensive testing and having a large amount of redundancy, yet these approaches can not give any *guarantees* of conformance to the specification, which becomes a necessity for large-scale systems.

Yet despite the fact that the formal methods research dates back to 1960's, until very recently the adoption of the formal tools in the industry remained weak. However, currently this trend starts to change. Many large companies have started using formal methods during the development of large software solutions. For example, Amazon has adopted the usage of TLA+ specification language for designing systems, which has "added significant value" by "finding subtle bugs" or proving correctness after "aggressive performance optimizations" [New14; New+15]. In the embedded system domain, the B Method was used to design correct by construction safety critical controller for automated screen doors for subway, with no bugs found after twenty years of deployment [Lec08]. Acknowledging the growing importance of formal methods, the recently published DO-178C [RTC11] document, which is used a basis for certification authorities on commercial software-based aerospace systems, allows the application of formal methods to replace certain forms of testing.

Chapter Outline We consider the question that given a *specification* S and a *program* P, how to *automatically* check whether P *conforms* to S. We start by outlining traditional approaches for checking correctness in Section 1.2. In Section 1.3 we give an informal definition of a specification. We outline the approaches for conformance checking Section 1.4, and in Section 1.5 we describe the outputs software analysis tools provide, and the practical implications of the possible verdicts. Finally, in Section 1.6 we list the contributions of this thesis which shape the following chapters.

1.2 Traditional Approaches for Ensuring Reliability

Extensive testing, either manual or automated through *unit-tests* is often the most preferred approach for ensuring the reliability of software. Clearly, if a unit-testing suite exposes a bug, an underlying program is buggy, yet if all tests pass we do not know whether there exists an input causing a problematic behavior. Traditionally, test engineers have been using the measure of *coverage* (a percentage of lines of code executed by the test) to assess the completeness of a test data. However, coverage is only a good metric while it is below 100%, as it gives a direction for improvement. Once the full coverage is achieved, there are no guarantees on whether the program contains problematic behavior, as not the entire input state space is covered; in words of Dijkstra [Dij69]: "testing is good for finding bugs, yet is woefully inadequate for showing their absence".

Both manual integration testing are costly and time consuming, even though they do not provide any guarantees on the resulting product. Research on *test generation* aims to generate the test input automatically instead. A notable success story, especially in the security domain, is a simple approach called *fuzzing* [MFS90]. The idea of fuzzing is to randomly change (*mutate*) user-provided *seed* data, and to feed it to the input program, until a crash (or a specification violation) is encountered. More sophisticated fuzzers such as AFL [Zal] additionally rely on genetic programming to generate new inputs, where the fitness function is determined by the coverage the generated input achieves for the program under test. Despite the technique simplicity, (or perhaps, due to) fuzzing has achieved enormous success in the security community, where a large fraction of vulnerabilities discovered today are found using fuzzing, including the *heartbleed* [CVE13] bug found in the OpenSSL stack.

Automated test generation can be especially successful when coupled with *runtime verification* [Bar+01; HG08] approaches which dynamically analyze the program during the execution, and report if the specification is violated. Examples of popular and influential runtime verification tools include ThreadSanitizer [SI09] and AddressSanitizer [Ser+12].

1.3 What is a Specification?

Informally, the specification defines *what* the software should do (e.g. store and retrieve data, drive a car, often referred to as a *functional* requirement), as well as *how* it should do it (e.g. do not crash, have a certain uptime, often referred to as a *nonfunctional* requirement).

Research into defining a formal language for expressing formal (and formally checkable) specification dates back to 1960's and includes languages such as E-ACSL [DKS13] and TLA+ [Cha+08] which allow the verification engineer to formally state checkable requirements the system has to fulfill. Yet today most software systems do not have such formally defined specifications, and their behavior is defined using a combination of prose and UML diagrams.

Generating a formal specification is a challenging task, additionally complicated by the fact that specification is written in a separate language software engineers are not familiar with. Thus, many approaches have been proposed to lessen the fundamental alienation between the code and the specification. For example, the Dafny [Lei10] programming language proposed by Leino et al., allows software engineers to embed the specification inside the code using usual language constructs, making the system more approachable for software engineers. Many *correct-by-construction* systems were designed using Dafny [Haw+14]. This follows the idea that specification is, fundamentally, not different from the code, and one should *refine* the other.

1.3.1 Safety and Liveness

Temporal language allows exactly stating how system should evolve with time, supporting a very rich set of properties. These properties are divided into two distinct groups: *safety* and *liveness* [MP91]. Fundamentally, a *safety* property establishes that no *bad* behavior happens, while *liveness* is used to express the property that something *good* eventually happens[1]. For example, safety deals with properties such as "no integer overflow", or "no assertion violation", or "no undefined behavior according to C11 standard", while the most famous liveness property is "the program has to be terminating", followed by more properties specific to concurrent systems, such as "parent thread eventually gains control".

In this thesis we only concern ourselves with safety properties, and we assume that a specification is already given in the form of a set of a program states which are considered to be erroneous. Examples of such specification include `assert` statement violations, as well as violations of *implicit* specification imposed by the language standard. In case of C that means lack of undefined behavior, which includes properties such as signed integer overflow, buffer overflow, null pointer dereference, use-after-free, and many others. Even though such a specification might seem simplistic, security exploits are regularly found in programs resulting from undefined behavior.

1.4 The Halting Problem and the Program Analysis Landscape

A decision procedure is *sound* if and only if it never returns false answers. Similarly, a decision procedure is *complete*, if it is able to return a verdict for all inputs.

The Halting problem [Tur36] states that it is *impossible* to construct an algorithm which would would state whether a given input program terminates for all possible inputs: that is, the underlying decision problem is *undecidable*. Rice's theorem [Ric53] generalizes this result further to the undecidability of yielding any sound and complete non-trivial statements about the computation result of any program written in a Turing-complete [Tur36] programming language.

These results shape the entire field of program analysis, stating that it is *impossible* to derive a sound and complete algorithm finding even the simplest property. However it can be also seen as the theorem which pushes the field towards being more applied: as applied mathematicians can not analytically solve most of the differential equations they deal with, it does not stop them from computing very good numerical approximations.

Similarly, the absence of soundness and completeness in no way precludes the possibility of software engineering tools which provide *useful* statements about the program. In this section we briefly outline the landscape of the software analysis field, and how different approaches get around the halting problem in order to reason about the software.

1.4.1 Finite Space Exploration

Technically, the halting problem does not apply to software running on physical hardware, as it is executed on machines with finite memory, making the entire system finite state. In practice though, total enumeration of all possible inputs is impractical, even for the simplest programs. For example, just iterating over all possible values for a *single* 64-bit integer requires $2^{64} = 2^{10^6} \approx 10^{3^6} = 10^{18}$ CPU cycles, taking decades on modern computers. Yet security

[1] Somewhat confusingly, liveness properties are not directly related to live variables [ASU86], though a variable may be considered alive if some operation depending on the variable value *eventually* happens.

researchers are often capable to achieve *total* coverage just by testing on systems with smaller bit width.

Additionally, many useful systems can be modelled as *automatons* which *do* have finite state-space, which can be exhaustively explored. Notably, Clarke and Emerson [CE82] (and independently Sifakis and Queille [QS82]) have published a influential idea that for a system represented as a Kripke structure [Kri63] and a specification given as an CTL formula, it *is* possible to construct an algorithm checking the conformance to the specification. Intuitively, the construction is performed by reducing both the system and the specification to a Büchi automaton, and then constructing the combination of the program automaton, and the negation of the property, and checking it for emptiness. From a high level it can be seen as an exhaustive state space exploration, which attempts to perform many reductions. The approach was successfully adapted for efficiently analyzing many finite state systems and protocols.

1.4.2 Correct By Construction Software

Instead of considering a *given* software artifact and attempting to construct a proof or finding a counter-example, such approaches aim to build a correct software in the first place, often embedding the proof in the produced software artifact.

A variable *type* can be considered an arbitrary predicate over the contents of the variable. From that perspective most type systems found in mainstream programming languages are very simple, and let the programmer express only the most basic predicates (e.g. in Java if a variable is declared as an `Integer`, a compiler will only compile the source code if it can prove that a variable indeed either points to an integer or to `null`). However, very complicated type systems including dependent types also exist, with Coq [CH85] being a notable example. In Coq, a proof of a theorem is constructed using types, and due to the Curry-Howard correspondence [How80], executable, correct-by-construction OCaml code can be lifted once the proof is finished. The Coq theorem prover is very successful, with verified C compiler [Bol+13] being one of the most prominent success stories.

1.4.3 Under-Approximating Approaches

Using under-approximation one can explore some subset of the reachable state space, stopping whenever some bound (time limit, size, input complexity) is reached. Such verification techniques verification include symbolic execution [Kin76], concolic execution [SMA05], bounded model checking [Bie+03] and others. An advantage of such techniques include the ability to generate the unit test triggering the fault on the discovery of the error state.

1.4.4 Over-Approximating Approaches

Approaches based on over-approximation sacrifice *completeness* in favor of *soundness*. For instance, an over-approximating checker for assertion violations can prove absence of errors for some input programs, while reporting an `UNKNOWN` verdict for others. Though many software engineers might see this as an unacceptable compromise, such a tool can still be very useful in practice: for example, terminating programs usually terminate for a simple intuitive reason which could be then found by the approach.

Over-approximating approaches usually rely on *abstraction* to shrink the program state space, making the full exploration feasible. Use of abstraction may result in *spurious* behaviors which the real system does not exhibit, causing the incompleteness. Such approaches include

dataflow [Kil73] analysis which predates the formal methods field, and later generalization into abstract interpretation [CC77a]. One of the biggest success stories of abstract interpretation is using Astrée [Bla+03] static analyzer for verifying lack of implicit errors in Airbus fly-by-wire controller.

In the rest of this thesis we mainly concern ourselves with over-approximating approaches, which are sound, but incomplete.

1.5 What is a Verifier Output?

Tools used for verification normally have three possible outputs for a given program and a specification:

- `TRUE`, signifying that the program conforms to the specification.
- `FALSE`, meaning that a counterexample was found.
- `UNKNOWN`, meaning that the analysis results are insufficient to draw a conclusion.

The `UNKNOWN` verdict is imposed by the halting problem (Section 1.4), and in practice often manifests itself as a timeout.

The `FALSE` verdict indicate that the tool has found a program *trace* violating the specification. Such a trace can be given e.g. as a deterministic *refinement* of the original program which replaces all user inputs and non-deterministic choices by given values, effectively generating a test vector [Bey+04] that can be compiled and executed with the original program. A reliable `FALSE` verdict is valued by software engineers, as it provides a reproducible bug which can be recorded in the bug tracker and eventually fixed, thus increasing the software quality.

The `TRUE` verdict means that the program does not contain any specification violations, and it was successfully *verified*. Despite its usefulness, *trusting* such a verification verdict in a business setting is often quite challenging. Firstly, the specification is often incomplete and potentially incorrect, and many bugs in safety critical software during the last decade were present in the specification. Secondly, software engineers often raise the classical point "who verifies the verifier", doubting the verdict due to the possibility of bugs in the software performing the analysis. Finally, while specification violation can be presented as a set of inputs to a given program, it is a lot more difficult to give *verification certificate* back to the end user, especially in a human-readable form. We believe that all three of these problems are fundamentally solvable. While specification does not have to be complete, verifying partial, or implicit (no crashes) correctness already increases a level of safety of the program. Secondly, the verifier can be either built in a correct-by-construction system such as Coq [CH85], or it can produce checkable *certificates* which can be independently verified by a separate tool.

1.6 Contributions and Thesis Outline

Generally, over-approximating approaches prove properties using *inductive strengthening*: finding an invariant which makes the desired property inductive under all program transitions, effectively giving a proof by induction. Such a strengthening is called an *inductive invariant*, as it establishes a universally valid fact for all program executions. Inductive invariants are very useful in all branches of program analysis, including compiler optimizations, verification and automated bug-hunting.

In this thesis we address the task of generating inductive invariants for the purpose of *program verification*: that is, a construction of verification algorithms which can present the TRUE verdict for a given input program and a safety specification. We concern ourselves with finding *small* inductive invariants, which can be used to prove the desired property of interest. Our work follows the tradition of static analysis and abstract interpretation [CC77a], yet we heavily use logic and solvers for boolean formulas over an arithmetic theory.

Computing even simplest numeric properties (e.g. *intervals*, $x \in [1, 5]$) with abstract interpretation is not guaranteed to converge in a finite time. As a result, a *widening* operator is used, which enforces convergence at a cost of precision. There exist many heuristics to recover this precision, yet they are inherently very brittle. A different approach called *policy iteration*, derived from game theory, provides a *guarantee* that the resulting inductive invariant is smallest possible in the given domain. Yet, policy iteration is rarely used in a program analysis field due to the complexity of the algorithm, high running cost, limitation to certain domains, and inability to cooperate with other analyses.

In this thesis we tackle these problems by extending the original algorithm, providing new strategies for generating abstract domains, and adapting policy iteration to summary computation. Additionally, we provide a new algorithm for computing potentially disjunctive inductive invariants, which is aimed to complement the analysis by policy iteration.

1.6.1 Theoretical Contributions

- In Chapter 3 we present the *local policy iteration* algorithm, which combines the precision of policy iteration with versatility of the traditional Kleene iteration approach. We study the properties of the new algorithm, we describe various extensions, and we provide extensive empirical evaluation. This contribution is a significantly improved version of the previously published result [KMW16], which also includes extended background on max-policy iteration.

- Local policy iteration finds the least inductive invariant (Section 2.5.1) in the *template constraints domain* (Section 2.8.3), which requires already present template annotations defining the *shape* of the possible inductive invariant. We present various approaches for generating *templates* with respect to the property we wish to prove in Chapter 4. This contribution was not published.

- In Chapter 5 we present a new framework for using intraprocedural analysis for *summary* computation, which potentially drastically improves the performance of the analysis, and makes it applicable to recursive programs. We study the parameterization of this framework with local policy iteration, its properties, and we provide an empirical evaluation. This contribution was not published.

- Local policy iteration technique excels at finding *convex* inductive invariants for the given program, yet many safety properties require non-convex inductive strengthening. We complement our LPI algorithm by the formula slicing approach for finding disjunctive inductive invariants. This approach is presented in Chapter 6 and was published in "Haifa Verification Conference" [KM16].

1.6.2 Engineering Contributions

A major goal of our work is to present *working*, *practical* tools which can be used both by end-users for program verification, and by the verification community for comparing different invariant synthesis approaches. Thus we present the following engineering contributions:

- LPI and Slicer modules for automated software analysis as presented for Software Verification Competition, described in Chapter 7 and published in "Tools for Algorithm Construction And Synthesis" [Kar16].
- JavaSMT library for interacting with SMT solvers, heavily used by our tools, described in Chapter 8 and published in "Verified Software: Tools, Techniques and Evaluations" [KBF16].

Chapter 2

Background

2.1 Introduction

In this chapter we introduce the concept of an inductive invariant, and we state its importance for proving program safety. We provide an overview for existing techniques for finding inductive invariants along with necessary prerequisites.

2.1.1 Chapter Outline

We start by fixing the notation in Section 2.1.2, and giving a formal definition of a program in Section 2.2. We then proceed to describe how both the program and the specification can be encoded as boolean formulas in Section 2.3. In Section 2.5 we show how safety properties are proven using inductive invariants. We give extensive background for abstract interpretation in Section 2.7, which can be seen as a framework for deriving inductive invariants, and we give an overview of a number of different abstract domains in Section 2.8. We outline the large block encoding technique which we use extensively in the rest of this thesis in Section 2.9. Finally, in Section 2.10 we show how different approaches can be unified (on the theoretical and on the implementation level) in the *configurable program analysis* (CPA) framework.

2.1.2 Notation and Definitions

We denote sets using capital Latin letters, e.g. A, B, C, D, and for elements of these sets we use lowercase letters, e.g. $a \in A$. We use **bold** letters in order to indicate vectors: $\mathbf{x}$. To distinguish between the program lines, as written by the software engineer, and mathematical expressions representing these, we use `typewriter` font for program statements, e.g. `x := x + 1`, and *math* font for mathematical expressions, e.g. $x' = x + 1$.

Usual sets of numbers will be denoted with blackboard letters, such as $\mathbb{R}$ for reals, $\mathbb{Q}$ for rationals, $\mathbb{Z}$ for integers and $\mathbb{N}$ for natural numbers. Let $\mathbb{B} = \{\top, \bot\}$ denote the set of the boolean values, where $\top$ stands for "true", and $\bot$ stands for "false". We define the set of *extended* real numbers as reals with positive and negative infinities adjoined as $\bar{\mathbb{R}} \equiv \mathbb{R} \cup \{+\infty, -\infty\}$. We use extended real numbers only for comparison and upper bound operations, and thus we do not define paradoxical operations (that is, we never evaluate e.g. $\infty - \infty$).

We denote a projection operator returning a one-indexed i^{th} element from a tuple of values v as $v|_i$ (e.g. $(3,4,5)|_2 = 4$). This notation is naturally extended to sets of indexes (e.g. $(3,4,5)|_{\{2,3\}} = (4,5)$), and variables by using quantifier elimination, e.g. $(y = 3 \wedge x = y)|_x \equiv \exists y.\, (y = 3 \wedge x = y) \equiv (x = 3)$. We shall also use basic linear algebra notation: $x^\top$ denotes the transpose of x, and thus $x^\top y$ is the inner product of x and y: e.g. $(1,2,3)^\top (x,y,z) = x + 2y + 3z$.

We denote the disjoint union operation using the coproduct symbol: that is, $A \coprod B$ evaluates to $A \cup B$, and additionally states that A and B are disjoint.

We make use of lambda calculus notation for defining functions, e.g. a function which squares its input is defined as $\lambda s.\, s^2$.

2.2 Program Formalization

We assign formal meaning to programs by mathematically modelling the program *semantics* [Flo67]. We formalize the program as a control flow automaton (CFA), which is equivalent to a program in a simple programming language with no procedures and no heap access[1]. This form is very similar to traditional control flow graph representation in compiler theory [ASU86], yet we associate statements with program edges.

Non-heap-manipulating programs in an imperative programming language with no recursion and no function pointers can be trivially converted to such a format by inlining functions and removing aliasing[2].

We define $\mathbf{x}$ to be a tuple of all program variables. For simplicity, we do not model variable types (even though they are supported by our implementation), and we assume that every variable is assigned a value from $\mathbb{R}$.

Definition 2.1 (Concrete Data State)**.** A *concrete data state* is a map $\mathbf{x} \to \mathbb{R}$ which assigns a real value to every program variable, and corresponds to a snapshot of a program memory during execution, excluding the program counter.

The set of all concrete data states is denoted by $\mathcal{C}$. A set $r \subseteq \mathcal{C}$, describing multiple concrete data states, is called a *region*. The set of all regions is denoted by $\mathcal{R} \equiv 2^{\mathcal{C}}$. In order to model the program counter, we introduce the set *nodes* of all possible program location.

Definition 2.2 (Concrete State)**.** A *concrete state* c is a tuple (m, n) where m is a concrete data state, and $n \in$ *nodes* is a program location. A concrete state corresponds to the whole memory snapshot, sufficient to reconstruct the entire program state.

A set of *edges* describes all possible transitions within a program. Each edge $e \in$ *edges* is a tuple $(a, \texttt{OP}, b)$, modelling the constraints on a transition from a to b, where $\{a, b\} \subseteq$ *nodes*, and `OP` is an operation performed on a transition, which is either a guard or an assignment. We formalize the analyzed language in Figure 2.1.

With the grammar for the language given, we define the formal semantics for the operators. As we are primarily interested in sets of concrete states, we only give the definition of the *collecting semantics* $\mathcal{R} \to \mathcal{R}$, which describes the transformation caused by an operator to an entire set of states.

The semantics of a numerical expression evaluation

$$[\![\texttt{<expr>}]\!] : \mathcal{R} \to 2^{\mathbb{R}}$$

is given by the usual evaluation rules on concrete data states contained in the input region, and the subsequent union of all resulting states. For example:

$$[\![\texttt{x + y}]\!] (\{\{x : 1, y : 1\}, \{x : 2, y : 2\}\}) = \{2, 4\}$$

[1] We extend the program model to allow for function calls in Chapter 5.

[2] In practice, our implementation supports both, as described in Chapter 7.

⟨*stmt*⟩ ::= ⟨*ident*⟩ ‘`:=`’ ⟨*expr*⟩ // Assignment to a single variable
| ⟨*ident*⟩ ‘`:=`’ ‘`input()`’ // Non-deterministic assignment
| ‘`assume`’ ‘`(`’ ⟨*bool_expr*⟩ ‘`)`’ // Guard
| ⟨*empty*⟩ // No-op

⟨*bool_expr*⟩ ::= ⟨*expr*⟩ ⟨*cmp_op*⟩ ⟨*const*⟩
| ⟨*bool_expr*⟩ ‘`or`’ ⟨*bool_expr*⟩
| ⟨*bool_expr*⟩ ‘`and`’ ⟨*bool_expr*⟩
| ‘`not`’ ⟨*bool_expr*⟩

⟨*expr*⟩ ::= ⟨*ident*⟩
| ⟨*const*⟩ // Numerical Constant
| ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| ‘`-`’ ⟨*expr*⟩

⟨*op*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’

⟨*cmp_op*⟩ ::= ‘`<=`’ | ‘`<`’ | ‘`>`’ | ‘`>=`’ | ‘`!=`’ | ‘`==`’

FIGURE 2.1: BNF Grammar of the Analyzed Language

If the evaluation process results in an invalid operation, such as division by zero, the output is $2^{\mathbb{R}}$, corresponding to all possible values (also referred to as $\top$). For example:

$$[\![\texttt{x / y}]\!]\left(\{\{x:1, y:0\}\}\right) = \top$$

Thus, the semantics for an assignment statement is given by the union of the assignment application on all concrete states contained inside the region:

$$[\![\texttt{<ident> := <expr>}]\!] \equiv \lambda r. \bigcup \{x : c' \text{ if } x = \texttt{<ident>} \text{ else } s[x] \mid x \in \mathbf{x} \wedge s \in r \wedge c' \in [\![\texttt{<expr>}]\!](s)\}$$

For example:

$$[\![x := x + y]\!]\left(\{\{x:1, y:0\}, \{x:0, y:1\}\}\right) = \{\{x:1, y:0\}, \{x:1, y:1\}\}$$

Similarly, the semantics for a non-deterministic input is given by:

$$[\![\texttt{<ident> := input()}]\!] \equiv \lambda r. \bigcup \{x : c' \text{ if } x = \texttt{<ident>} \text{ else } s[x] \mid x \in \mathbf{x} \wedge s \in r \wedge c' \in \mathbb{R}\}$$

We define the helper semantics for Boolean expressions as a function $\mathcal{C} \to \mathbb{B}$, which performs evaluation under the usual rules, returning “true” if and only if there exists a concrete state in a region making the value of the expression “true”. Assume statements filter the input region,

letting only the conforming concrete states through:

$$[\![\texttt{assume(<bool\_expr>)}]\!] \equiv \lambda r. \bigcup \{s \mid s \in r \wedge [\![\texttt{<bool\_expr>}]\!](s)\} \tag{2.1}$$

For example, $[\![\texttt{x <= 5}]\!](\{\{x:1\},\{x:6\}\}) = \{\{x:5\}\}$.

We have defined the collecting semantics naturally assuming that the program performs execution *forward*, which corresponds to the actual execution process (*strongest postcondition*). Dually, the semantics can be also defined using the *weakest precondition*: for an output region r' and an operator OP it gives the largest region which maps into r' under $[\![\texttt{OP}]\!]$. In this thesis we stick to the strongest postcondition semantics.

Definition 2.3 (Control Flow Automaton)**.** A CFA is a tuple $(\mathit{nodes}, \mathit{edges}, n_0, \mathbf{x})$, where *nodes* is a set of program control states modelling the program counter, $n_0 \in \mathit{nodes}$ is a program starting point, $\mathbf{x}$ is a set of program variables, and $\mathit{edges} \subseteq \mathit{nodes} \times \texttt{OPS} \times \mathit{nodes}$, where OPS is a set of all possible program operators.

Definition 2.4 (Program Path)**.** A *program path* is a sequence of concrete states $\langle c_0, \ldots, c_n \rangle$ where $c_0 = (m_0, n_0)$ and for any two consecutive states $c_i = (m_i, n_i)$ and $c_{i+1} = (m_{i+1}, n_{i+1})$ there exists an edge $(n_i, \texttt{OP}, n_{i+1}) \in \mathit{edges}$ such that $n_{i+1} \in [\![\texttt{OP}]\!](n_i)$.

A concrete state $s_i = (m, n)$, and the associated node n, are both called *reachable* iff there exists a program path which contains s_i.

2.3 Logic in Program Analysis

So far, we have introduced *regions* and *operators*: sets of states and functions from regions to regions. We shall now use *formulas* to succinctly represent both, as well as the *specification* (Section 1.3) the program is expected to conform to.

We operate over first-order logic formulas within a theory $\mathcal{T}$ such that the problem of deciding the satisfiability of a quantifier-free formula is NP-hard. Suitable theories include propositional reasoning, linear real arithmetic, and linear integer arithmetic (Presburger arithmetic). A set of all such formulas over a set of free variables is denoted by $\mathcal{F}$.

A formula is said to be an *atom* if it does not contain any logical connectives (e.g. it is a comparison $x \leq y$ between integer variables), a *literal* if it is an atom or its negation, a *clause* if it is a disjunction of literals, and a *cube* if it is a conjunction of literals. A formula is in *negation normal form* (NNF) if negations are applied only to atoms, and it is in *conjunctive normal form* (CNF) if it is a conjunction of clauses.

We abuse the notation by conflating the set of program variables $\mathbf{x}$ defined in Section 2.2, and the set of free variables $\mathbf{x}$ appearing inside the formula. For a set of variables $\mathbf{x}$, we denote by $\mathbf{x}'$ a set of primed variables where the prime symbol was added to all the elements of $\mathbf{x}$. With $\phi[a_1/a_2]$ we denote the formula ϕ after all free occurrences of the variable a_1 have been replaced by a_2. This notation is extended to sets of variables: $\phi[\mathbf{x}/\mathbf{x}']$ denotes the formula ϕ after all occurrences of the free variables from $\mathbf{x}$ were replaced with corresponding free variables from $\mathbf{x}'$. For brevity, a formula $\phi[\mathbf{x}/\mathbf{x}']$ may be denoted by ϕ'. We use the brackets notation to indicate what free variables can occur in a formula: e.g. $\phi(\mathbf{x})$ can only contain free variables in $\mathbf{x}$. The brackets can be dropped if the context is obvious.

A formula $\phi \in \mathcal{F}$ is called *satisfiable* if there exists a *variable assignment* $\mathcal{M}$ (referred to as *model*) such that $\phi[\mathbf{x}/\mathcal{M}]$ is a tautology (written as $\mathcal{M} \models \phi$). For example, $\{a:\top, b:\top\} \models a \wedge b$.

Note that models associated with formulas over $\mathbf{x}$ are isomorphic to concrete data states, and we abuse the notation by treating them interchangeably.

Checking a formula for satisfiability (finding a model or proving that none exists) is a classical NP-complete problem [Coo71] even in the absence of extra theories (all variables are boolean). However, modern SMT (*satisfiability modulo theories*) solvers (notably Z3 [MB08], CVC4 [Bar+11], Yices [DM06]) often perform these checks very efficiently in practice, e.g. Z3 is routinely capable of dealing with formulas which are too large to fit into the machine RAM. This paradoxical property is very similar to the halting problem stated in the introduction (Section 1.4): the computational complexity is defined for the worst case, which is not necessarily relevant for the actual queries posed to the solver.

2.3.1 Conversion to Formulas

Semantics of a formula $\phi(\mathbf{x})$ defines a region of all concrete data states which it models ($[\![\phi]\!] \equiv \{c \mid c \models \phi\}$). This allows us to treat formulas over $\mathbf{x}$ as regions.

We represent an operator $\texttt{OP} \in \texttt{OPS}$ as a formula $\tau(\mathbf{x} \cup \mathbf{x}')$ over the initial variables $\mathbf{x}$ (representing a valid from-state) and primed variables $\mathbf{x}'$ (representing a valid to-state) such that for every pair of models $(\mathcal{M}_1, \mathcal{M}_2)$ we have $\mathcal{M}_2 \in [\![\texttt{OP}]\!](\mathcal{M}_1)$ if and only if $(\mathcal{M}_1 \cup \mathcal{M}_2[\mathbf{x}/\mathbf{x}']) \models \tau$. Intuitively, a formula is satisfiable over $(\mathcal{M}_1 \cup \mathcal{M}_2)$ if an operator applied to $\mathcal{M}_1$ generates $\mathcal{M}_2$ modulo priming renamings.

E.g. for a program over variables `x, y` a guard `x <= 9` is represented by a formula $x \leq 9 \wedge x' = x \wedge y' = y$, and an assignment `x := x + 2` is represented by $x' = x + 2 \wedge y' = y$. Note that as the number of variables increases, so does the number of *frame* assignments which state that all unmodified variables remain the same. The problem of dealing with a large number of such spurious assignments is remarkably similar to the *frame problem* [Hay71] in the artificial intelligence field. Consequently, in practice instead of having a large number of spurious assignments a single static assignment form (SSA) [Cyt+91] is used, which avoids the problem by renaming variables in such a way that every variable is assigned exactly once.

For example, the program `x=0; y=1; z=1; x=x+y; x=x+z;` is converted into the program `x_0=0; y_0=1; z_0=1; x_1=x_0+y_0; x_2=x_1+z_0;` which is represented by the formula $x_0 = 0 \wedge y_0 = 1 \wedge z_0 = 1 \wedge x_1 = x_0 + y_0 \wedge x_2 = x_1 + z_0$.

Finally, by converting regions and operators to formulas we can also encode the strongest postcondition. For a region represented by $\phi(\mathbf{x})$ and a transition given by $\tau(\mathbf{x} \cup \mathbf{x}')$ a region corresponding to the strongest postcondition is $(\exists \mathbf{x}.\, \phi(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}'))[\mathbf{x}'/\mathbf{x}]$. For example, the postcondition of a region $x > 5$ under a transition $x' = x + 1$ is given by $(\exists x.\, x > 5 \wedge x' = x + 1)[\mathbf{x}'/\mathbf{x}]$ which simplifies to $x > 6$.

2.4 Finding Bugs with Formula Encoding

The logic based encoding gives rise to several approaches for automatically findings error properties. *Symbolic execution* [Kin76] runs the program while keeping the variables symbolic: this is equivalent to dynamically encoding the formula after each step. Symbolic execution has been often used in practice to find many bugs in real world software [CDE08]. A symbolic and concrete execution hybrid *concolic execution* [GKS05], which uses concrete program values where the constraint solving is intractable, has been successfully used at Microsoft to find many bugs in released products [GLM08].

Bounded model checking [Bie+99] takes a different encoding approach, and proves that the program satisfies the property for all traces of length $\leq n$, where n is increased from 1 up to the user-supplied bound (or until the timeout is reached). CBMC [CKL04] is one of the most successful tools based on this approach.

2.5 Proving Safety

For programs without loops, applying the formula encoding stated in Section 2.3.1 gives a straightforward way for checking safety: the entire program is converted to a formula $\phi(\mathbf{x})$ by iteratively applying the strongest postcondition encoding, the desired property is converted to a formula $P(\mathbf{x})$, and an SMT solver is queried for the satisfiability of $\phi(\mathbf{x}) \wedge \neg P(\mathbf{x})$. If the formula is unsatisfiable, the program is safe (assuming our program encoding is sound), and otherwise the model $\mathcal{M} \models \phi(\mathbf{x}) \wedge \neg P(\mathbf{x})$ gives us the *counterexample* which can be used to automatically generate a failing testcase [Bey+04] (assuming our encoding is complete).

This procedure is not directly applicable for programs where the maximum execution length is unbounded. For certain structures of loops, it is possible to find a *transitive closure* representing the loop effect in a sound and complete way: then the effect of the program can be still represented by a single formula. Such a summarization is performed by approaches based on *acceleration* [Boi98].

Yet in general it is not possible to *summarize* a loop with a first order formula *within a decidable theory*. Thus safety is generally proven for infinite systems using *inductive invariants* [Tur49; Flo67].

2.5.1 Inductive Invariants

Consider a general safety property: prove that some region is unreachable at some set of program locations. By encoding the program counter as a regular variable we can encode the program using a single location n, and two transitions $I(\mathbf{x}')$ and $\tau(\mathbf{x} \cup \mathbf{x}')$ representing the initial state and the transition relation respectively.

In general, τ represents a complex, non-deterministic recurrent relation for which due to the halting problem it is impossible to obtain a computable analytic solution. Properties of such discrete systems are almost invariably proven by *induction*. That is, in order to establish a property P universally we first show that it holds at the initial state, and then that it is preserved under the transition relation.

The negation of the error $\neg E(\mathbf{x})$ is a natural property to check for inductiveness. However, even if $\neg E$ holds universally, it is rarely inductive with respect to the transition relation. Thus it is important to distinguish between *invariants* and *inductive invariants*.

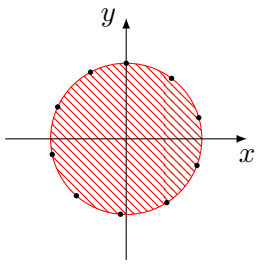
Definition 2.5 (Invariant)**.** A property $I(\mathbf{x})$ is an *invariant* for a CFA P if and only if for all program paths (Definition 2.4) for P, for all elements (m, n) of a program path, $m \models I$ holds.

Definition 2.6 (Inductive Invariant)**.** A property $E(\mathbf{x})$ is an *inductive invariant* for the CFA P represented by the initial state I and the transition relation τ if and only if E satisfies the initial condition I and is inductive under the single-state encoding of P (that is, $\forall \mathbf{x}, \mathbf{x}'.\, E(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}') \implies E(\mathbf{x}')$).

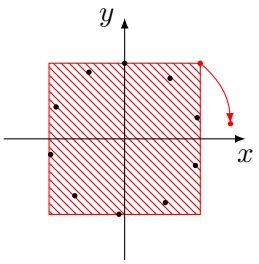
By definition, a set R of all reachable CFA states is an inductive invariant. Such a set is a least invariant, and a least inductive invariant at the same time. However, as we have previously mentioned, R is not computable in general due to the halting problem. All over-approximations

```
float x = 0;
float y = 1;
while (input()) {
    x=0.8*x+0.6*y;
    y=-0.6*x+0.8*y;
}
```

(A) program listing: each iteration rotates (x, y) clockwise



(B) the circle is an inductive invariant, fitting first ten concrete states



(C) the box is a non-inductive invariant: a corner is mapped outside

FIGURE 2.2: Example of a non-inductive invariant. A program shown in Figure 2.2a at each iteration multiplies the vector (x, y) by a rotation matrix given by an angle $\theta \equiv \sin^{-1}(-0.6) \approx -37°$. That effectively rotates (x, y) by ≈ 37 degrees clockwise for a non-deterministic number of iterations starting from the point $(0, 1)$. An inductive invariant for such a program is a circle of radius 1 centered at the origin, as shown in Figure 2.2b. The box of size 2 centered at the origin is a *non-inductive* invariant, as shown in Figure 2.2c: despite the fact that the box contains all the reachable points, it is not inductive under the transition relation, due the existence of points which are mapped outside by the rotation.

$O \equiv R \coprod E$ of R are invariants, but not all of them are inductive, as states in E can give rise to spurious transitions not contained in O. An example of such a non-inductive invariant is shown in Figure 2.2.

Both definitions are not *constructive*: they tell us nothing about how such a property can be found. Moreover, the first definition is not even *certifiable*: while *counterexamples* (program paths which contain a property violation) can be used to rule out non-invariant properties, we have no way to check whether the given property is actually an invariant. This is expected, since a sound and complete procedure for testing whether a given property is an invariant would violate the Rice's theorem. Such a check can be however easily performed for an *inductive invariant* by testing it for inductiveness, as we show in the next section. Furthermore, a standard procedure to prove that a given property P is an invariant, is to find a *strengthening* S such that $S \wedge P$ is an inductive invariant. In 1969, Manna [Man69] has shown that for intraprocedural programs it is is always possible to find such an inductive strengthening, a result which was later extended to interprocedural programs with recursive procedures by Bakker and Meertens [BM75].

In the rest of this thesis we shall deal exclusively with inductive invariants: furthermore, the theoretical contributions of this manuscript are new methods for inductive invariant synthesis.

2.5.2 Showing Inductiveness

Inductiveness can be shown for a property with the help of the formula encoding for properties and SMT solvers (Section 2.3). Given a formula $\phi(\mathbf{x})$ representing the desired property, transition relation $\tau(\mathbf{x} \cup \mathbf{x}')$ and a set of initial states $I(\mathbf{x})$, ϕ is inductive if and only if the following is valid for all $\mathbf{x}, \mathbf{x}'$:

$$\begin{aligned} \text{Initiation: } & I(\mathbf{x}) \implies \phi(\mathbf{x}) \\ \text{Consecution: } & \phi(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}') \implies \phi'(\mathbf{x}') \end{aligned} \tag{2.2}$$

Universal properties are traditionally proven using negation. That is, in order to prove the consecution condition from Equation 2.2 the query in Equation 2.3 is posed to the solver, which

is unsatisfiable if and only if the consecution is valid for all possible values of $\mathbf{x}, \mathbf{x}'$.

$$\phi(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}') \wedge \neg\phi'(\mathbf{x}') \tag{2.3}$$

For a quantifier-free formula ϕ inductiveness checking is co-NP-complete. However, if ϕ is existentially quantified, the problem is Π_2^p-complete due to the fact that outer existential quantifiers (which can be normally removed using Skolemization [KS08]) become universal under negation. Thus in the rest of this thesis we shall assume that the property ϕ representing the *candidate* inductive invariant is quantifier-free.

2.5.3 Inductive Assertion Map

Definition 2.6 and the check in Equation 2.2 give us a way for checking a given property for inductiveness. However, such a direct check is often very inconvenient, as it requires re-encoding the entire program as a single-loop transition system. Instead we use the *inductive assertion map* formalism [SSM05], which represents the inductive invariant as a map, and associates a separate property to each program location. We shall also refer to such maps as inductive invariants.

Definition 2.7 (Inductive Assertion Map)**.** A mapping $I : \mathit{nodes} \to \mathcal{F}(\mathbf{x})$ is an *inductive assertion map* (also referred to as an inductive invariant) for a CFA $(\mathit{nodes}, \mathit{edges}, n_0, \mathbf{x})$ if and only if it satisfies the following conditions for *initiation* and *consecution*:

$$\begin{aligned} \text{Initiation: } & I(n_0) = \top \\ \text{Consecution: } & \text{for all edges } (a, \tau, b) \in \mathit{edges}, \text{ for all } \mathbf{x}, \mathbf{x}' \\ & I(a)(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}') \implies (I(b))'(\mathbf{x}') \end{aligned} \tag{2.4}$$

The Equation 2.4 is referred to as a system of *semantic equations* for a CFA. Intuitively, the initiation condition dictates that the initial program state at n_0 is covered by I, and the consecution condition dictates that under all transitions I should map into itself. Similarly to Equation 2.3, the consecution condition in Equation 2.4 can be verified by checking the negation for unsatisfiability.

2.5.4 k-Induction

In many cases, a property of interest is not inductive under the transition relation, but is inductive under multiple *compositions* of it: $f \circ f$ on the operator level, or $\tau[\mathbf{x}'/\hat{\mathbf{x}}] \wedge \tau[\mathbf{x}/\hat{\mathbf{x}}]$ on the formula level. For example consider the program in Figure 2.3. The assertion $0 \leq x \leq 5$ is not inductive under the loop transition relation, due to the possibility of the transition from $\{x : 5, y : 0\}$ to $\{x : 6, y : 1\}$. Yet it *is* inductive for the modified program where the loop transition is unrolled five times. Of course, a stronger invariant $x = y \wedge 0 \leq x \leq 5$ is simply inductive, but it is not readily available as an error property. In general, from a k-inductive invariant it is always possible to extract an inductive invariant, yet sometimes at the cost of the exponential explosion.

This observation has led to the technique called *k-induction* [Don+11] where the given invariant candidate (usually, negation of an error property, potentially strengthened with an inductive invariant [KT11]) is repeatedly tested for inductiveness where the value of k

```
int x = 0;
int y = 0;
while (input()) {
    x++;
    y++;
    if (y == 5) {
        x = 0;
        y = 0;
    }
}
assert(0 <= x && x <= 5);
```

FIGURE 2.3: k-Induction Motivation

(corresponding to the number of compositions of the transition relation) is incremented at each step. At the extreme, for finite-state systems every invariant is k-inductive for a sufficiently large k, making the framework especially suitable for state machine encodings.

2.5.5 Back to Safety

In order to prove that a property P universally holds, we need to find a *strengthening* S such that $P \wedge S$ is an inductive invariant. It is important to note that once an inductive invariant is found it is irrelevant in which way it was generated: due to intuition, because a "little bird" has told us, or simply because of an intelligent guess, once we establish the inductiveness, we have a formal checkable *proof* of the fact that the invariant universally holds. In later chapters of this thesis we develop techniques for generating inductive invariants.

2.6 Inductive Invariants from Counterexamples to Induction

Equation 2.4 lets us test a desired property P for inductiveness, and the output is either an `UNSAT` verdict corresponding to the case where the property is inductive, or a *counterexample to induction*, represented by the model $\mathcal{M}(X \cup X')$. Such a model $\mathcal{M}$ not only states that P is not inductive, but it also gives us a reason *why*: it specifies a precise state inside P from which the "jump" to $\neg P'$ is possible. Many approaches for invariant synthesis rely on using such counterexamples-to-induction in order to generate an inductive invariant.

Aaron Bradley in his seminal work on *property directed reachability* [Bra07] presents a way for generating new lemmas which can be used to strengthen the initial candidate invariant from the counterexample to induction. Such method leads to an efficient IC3 algorithm for SAT checking without unrolling [Bra11].

2.7 Inductive Invariants by Abstract Interpretation

The line of research concerned with proving program properties by finding inductive invariants goes back to compiler research and classical dataflow [Kil73] analysis techniques, including live variables [ASU86] calculation, location reachability, constant propagation, and others[3].

However, it took a fundamental work of Cousot and Cousot [CC77a] to generalize the underlying notion of *abstract domain*, and consequently *abstract interpretation* to unify and

[3]Indeed, despite the fact that the term "inductive invariant" was not traditionally used in dataflow analysis, and a different CFG formalism is used, the analysis runs in a loop until a *fixed point* is reached: that is, further propagation results in no updates — which is an inductive invariant by definition.

extend the previously used approaches. As usual, we start by describing abstract interpretation intuitively, and we give the formal notions in Section 2.7.1.

The aim of the abstract interpretation is to compute an inductive invariant in the *abstract domain*. Intuitively, an *abstract domain* is a set equipped with a partial order, where every element groups concrete states by the property of interest. E.g. for a program with a single variable x an *interval abstract state* $[0, 10]$ *concretizes* to a set of concrete data states where the value of x is between 0 and 10: $[\![[0, 10]]\!] = \{\{x : v\} \mid v \in [0, 10]\}$. The set of such properties (the set of intervals in our example) is called an *abstract domain*, and a function mapping an *abstract element* to a region is called *concretization* and is traditionally denoted by γ. Dually, the function from a region $r \in \mathcal{R}$ to the smallest element of the abstract domain a, such that $r \preceq \gamma(a)$ is called an *abstraction* α[4]. Intuitively, an element of the abstract domain succinctly represents a region of variable values (Section 2.2). In order for the abstraction to exist the abstract domain needs to satisfy certain criteria which we cover in this section: e.g. in the intervals example if we define the abstract domain as the set $\mathbb{R} \times \mathbb{R}$ it would be impossible to construct the abstraction function, as there would be no valid abstraction for a region where the value of x is not bounded.

Given an abstract domain, an inductive invariant can be constructed using Kleene fixpoint iterations [Kle52]. Kleene iterations can be seen as running the program in the abstract *interpreter*, while recording intermediate values (*invariant candidates*) associated with different program locations (CFA nodes). The iteration process starts by assigning to each CFA node the smallest invariant candidate $\bot$ corresponding to an unreachable value, and assigning the invariant candidate $\top$ to program entry (corresponding to the largest possible region, as the memory is not initialized at the program start). At each step, an abstract value is propagated through the CFA edges, with *abstract transformer* being applied (running the program with abstract values instead of the concrete ones, using *abstract semantics*). E.g. the interval $x \in [0, 10]$ under the operation `x++` is transformed into $x \in [1, 11]$. If after the propagation two abstract values exist at the same location, they are *joined* — that is, replaced with the least element in the abstract domain which is greater than both of the joined elements. E.g. intervals $x \in [0, 3]$ and $x \in [4, 5]$ can be joined to a new interval $x \in [0, 5]$. The process repeats until the iterations *converge*: that is, propagation and joining steps do not change the previous invariant candidate. The obtained result, provided that the iteration has converged in finite time, is an *inductive invariant*.

2.7.1 Formal Definitions

Definition 2.8 (Lattice)**.** A lattice L is a set equipped with a reflexive, antisymmetric and transitive partial order relation $\preceq_L$.

Definition 2.9 (Complete Lattice)**.** A complete lattice L is a lattice where every subset in L has a supremum and infimum under $\preceq_L$ in L. That is, there exists a unique join operator $\sqcup_L : 2^L \to L$, and the meet operator $\sqcap_L : 2^L \to L$, which compute supremum and infimum respectively for any subset of L: $\forall a \in L. a \preceq_L \sqcup L$ and $\forall a \in L. \sqcap_L L \preceq a$.

The set of regions, representing groups of concrete program states defined in Section 2.2 is a complete lattice using the inclusion relation as a partial order, and set union and intersection as join and meet operators respectively.

[4]Though later we show that it is possible to have a useful abstract domain for which a function α does not exist.

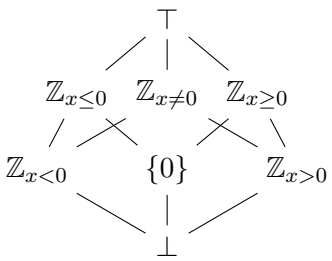


FIGURE 2.4: Hasse Diagram for Sign Abstract Domain over Integers

Definition 2.10 (Abstract Domain)**.** An abstract domain is a tuple $(\mathcal{D}, \alpha, \gamma)$ where $\mathcal{D}$ is a complete lattice equipped with a partial order $\preceq_{\mathcal{D}}$, $\alpha : \mathcal{R} \to \mathcal{D}$ is an abstraction function, converting a region r to an element of $\mathcal{D}$, and $\gamma : \mathcal{D} \to \mathcal{R}$ is a concretization function, converting an element of the abstract domain a to a region. The tuple (α, γ) has to form a Galois connection [CC92], that is the following has to hold:

$$\forall a \in \mathcal{D}. \forall r \in \mathcal{R}. \alpha(r) \preceq_{\mathcal{D}} a \iff r \subseteq \gamma(a) \tag{2.5}$$

Intuitively, an abstract domain is a grouping of concrete states by the property of interest. Thus we can *abstract* from the region description as a set of concrete states into the chosen domain of properties we choose to care about. Having such a definition allows us to reason about regions fulfilling certain properties: e.g. a region where the value of a certain variable is never *zero*.

Example 2.1 (Sign Abstract Domain)**.** Consider the sign abstract domain over a single integer variable x. The domain element states whether the variable is zero, strictly less than zero, strictly greater than zero, or a join or a meet over these. The lattice associated with this domain is shown in Figure 2.4. Note the values $\top$ and $\bot$ denoting "all are values possible" and "empty region" respectively. Concretization function for this domain corresponds to the element label in Figure 2.4, and abstraction maps a region r to the *smallest* abstract element a for which $r \subseteq \gamma(a)$. E.g. $\alpha(\{1, 2, 3, 4, 5\}) = \mathbb{Z}_{x>0}$, and $\gamma(\{0\}) = \{0\}$.

Example 2.2 (Intervals Abstract Domain)**.** The interval abstract domain is a mapping $\mathbf{x} \to (\bar{\mathbb{R}} \times \bar{\mathbb{R}})$, which unlike $\mathbb{R}^2$ forms a complete lattice. An element of the abstract domain $\{x : (a, b)\}$ concretizes to a set $\{c \mid c[x] \leq b \wedge c[x] \geq -a\}$, and the comparison is given using the usual component-wise comparison on tuples, applied component-wise to maps.

We are interested in self-maps $f : L \to L$ where L is a complete lattice which is both a domain and a codomain. Such a function f is called *monotone* if it is order preserving: $a \preceq b \implies f(a) \preceq b$. A point $a \in L$ is called a *fixed point* (or a *fixpoint*) if and only if $f(a) = a$. Tarski's fixed point theorem states that the set of fixpoints of a monotone function f on a complete lattice L is a complete lattice L_f itself [Tar55], which consequently has least and largest element. By μf we denote the least fixed point of f on L, and $\mu|_{\geq a} f$ denotes the least fixed point which is larger or equal to a. We use the power notation f^i to indicate the continuous application of f multiple times ($f(f(f(\ldots))))$.

Recall the definition of collecting semantics given in Section 2.2. A collecting semantics function $[\![\texttt{OP}]\!] : \mathcal{R} \to \mathcal{R}$ is a monotone self-map, and its fixed points correspond to inductive invariants. Yet as even storing these fixed points is often infeasible (all concrete data states

reachable at a given CFA node), we are interested in least fixed point of *abstract semantics* instead.

Definition 2.11 (Abstract Semantics)**.** Abstract semantics function $[\![.]\!]^\sharp$ for an abstract domain $(\mathcal{D}, \alpha, \gamma)$ and a program operator `OP` yields a function $\mathcal{D} \to \mathcal{D}$ which satisfies the following equation for all regions r:

$$[\![\texttt{OP}]\!](r) \subseteq \gamma([\![\texttt{OP}]\!]^\sharp(\alpha(r))) \tag{2.6}$$

That is, the abstract transformer has to be *sound*: it has to include all states given by the collecting semantics. The *best abstract transformer* which returns the smallest abstract state for a given input and a given operator `OP` while satisfying the soundness condition in Equation 2.6 for an abstract domain defined by (α, γ) is:

$$f \equiv \lambda a.\, \alpha([\![\texttt{OP}]\!](\gamma(a))) \tag{2.7}$$

2.7.2 Abstract Value Transformer

Abstract semantics defines how the value in the abstract domain is transformed by the operator. E.g. if we operate in the abstract domain of the intervals over a single program variable x, showing only the bound on the variable x for clarity, we have $[\![\texttt{x <= 9}]\!]^\sharp([0, 10]) = [0, 9]$, and $[\![\texttt{x += 1}]\!]^\sharp([0, 1]) = [1, 2]$. Yet we are interested in the abstract domain element resulting from the join over *all* possible abstract domain elements at all CFA nodes. We start by generalizing the abstract semantics to a function *next* which acts on the entire CFA at the same time, by combining the effect of the update of all the incoming edges for every node:

$$\begin{aligned} &next : (nodes \to \mathcal{D}) \to (nodes \to \mathcal{D}) \\ &next \equiv \lambda S. \left\{ n : \sqcup \left\{ [\![e]\!]^\sharp(S[n_0]) \mid (n_0, e, n) \in edges \right\} \mid n \in nodes \right\} \end{aligned} \tag{2.8}$$

Furthermore, we interested in all values possible at the given point, and we generalize *next* further to a function *update* with the same signature which *combines* the previous result with new values at all CFA nodes.

$$update \equiv \lambda S. \left\{ n : S[n] \sqcup \left\{ [\![e]\!]^\sharp(S[n_0]) \mid (n_0, e, n) \in edges \right\} \mid n \in nodes \right\} \tag{2.9}$$

Unlike abstract semantics, the output of a function *update* is always greater or equal to its input: for all input elements d it holds that $update(d) \succeq d$. Kleene's theorem [Kle52] states that for a monotone increasing self-map on a complete lattice continuous applications starting from the least element can converge only at the least fixed point. Hence least inductive invariant in the abstract domain can be found as iterative application of the function *update* $(update(\bot), update(update(\bot)), \ldots)$. Such iterative application is referred to as *Kleene iteration*.

The least fixed point under the abstract semantics which covers the input state is referred to as the *least inductive invariant in the abstract domain*, or just *least inductive invariant* when the domain is apparent from the context. The Kleene iteration process for an input CFA is formalized with the worklist algorithm in Algorithm 2.1. Results on chaotic iterations [Bou93] state that the resulting invariant does not depend on the precise iteration order.

Example 2.3 (Abstract Interpretation Run)**.** Consider running abstract interpretation with

Algorithm 2.1 Kleene Worklist Iteration Algorithm.

Input: CFA $(nodes, edges, n_0, \mathbf{x})$, abstract domain $(\mathcal{D}, \alpha, \gamma)$, abstract transformer $[\![.]\!]^\sharp$
▷ Inductive assertion map $nodes \to \mathcal{D}$
$map \leftarrow \{\}$
▷ Initial state associated with starting location.
$map(n_0) \leftarrow \top$
for all $n \in nodes \setminus n_0$ **do**
    ▷ All other nodes are initially considered unreachable.
    $map(n) \leftarrow \bot$
end for
▷ Worklist for nodes which should be expanded.
$q \leftarrow \{n_0\}$
while $q \neq \emptyset$ **do**
    Pop n from q
    for all $(n, \texttt{OP}, n') \in edges$ **do**
        ▷ Previously held value.
        $prev \leftarrow map(n')$
        $map(n') \leftarrow [\![\texttt{OP}]\!]^\sharp (map(n)) \sqcup prev$
        if $map(n') \not\sqsubseteq prev$ **then**
            ▷ Add n' to worklist if the value is not covered.
            $q \leftarrow q \cup \{n'\}$
        end if
    end for
end while
return map

intervals on a trivial counter program shown in Figure 2.5. As we have only one variable and only one CFA location an invariant candidate can be represented by a single interval $[a, b]$. The abstract transformer associated with the assignment `x := 0` sets the output interval to $[0, 0]$ regardless of the input. Similarly, the transformer associated with `x < 10; x++` increments both bounds as long as they are smaller than 10. Thus we get the following run of abstract interpretation:

- $\bot$ initially.
- $[0, 0] \sqcup \bot = [0, 0]$ after the first update.
- $[1, 1] \sqcup [0, 0] = [0, 1]$ after the first increment.
- $[0, 1] \sqcup [1, 2] = [0, 2]$ after the second increment.
- ...
- $[0, 10] \sqcup [10, 10] = [0, 10]$ the iteration reaches the *fixed point.*

2.7.3 Convergence and Widening

From Kleene fixed point theorem we know that iterations shown in Algorithm 2.1 can converge only to the least fixed point of the abstract domain. We introduce the notion of the lattice

```
1  int x = 0;
2  while (x < 10) {
3      x++;
4  }
```

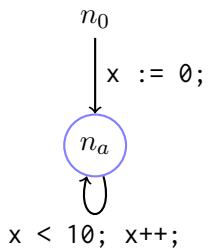


FIGURE 2.5: A simple counter program and the corresponding CFA.

height in order to give the convergence results.

Definition 2.12 (Complete Lattice Height)**.** A sequence of elements $s \equiv (s_1, s_2, s_3, \ldots)$ in a lattice L is called a *chain* if and only if every subsequent element is greater than the previous one: $\forall i.\, s_i \preceq s_{i+1}$. The size of the largest possible chain is referred to as a lattice *height*, and is denoted as $\|L\|$. The abstract domain height is defined as the height of the underlying lattice, and is similarly denoted as $\|\mathcal{D}\|$.

In the Example 2.1 the height of the sign abstract domain is 4. With the height of the domain being defined, we can state the result on Kleene iteration termination and precision:

Theorem 2.1 (Kleene Iteration Termination)**.** For an input CFA $(nodes, edges, n_0, X)$ and the analysis domain $\mathcal{D}$, the run of Kleene iteration in $\mathcal{D}$ requires at most $O(\|\mathcal{D}\|\|nodes\|)$ iterations and converges with the least inductive invariant in $\mathcal{D}$.

The proof is trivial and immediately follows from Kleene's and Tarski's theorems. However, the simplistic trace in Example 2.3 already highlights an important limitation of such an approach on *infinite height* lattice, such as the lattice of intervals. If the guard `x < 10` is removed from the transition relation, the analysis run does not terminate and continues happily incrementing the bounds forever.

In order to address this problem, Cousot and Cousot [CC77a] introduce the *widening* operator $\nabla : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, which enforces the termination after finitely many applications even for a lattice of infinite height. The widening operator has to be defined in such a way that for any input values any sequence of widening applications eventually converges at a single value. For example, widening on interval domain is defined to set the *moving* constraint to infinity, e.g. $[0, 1]\nabla[0, 2] = [0, \infty]$. The modification of Algorithm 2.1 which includes widening requires updating the right hand side of Line 17 to $prev\nabla([\![\texttt{OP}]\!]^\sharp\,(map(n)) \sqcup prev)$: that is, the widening is applied after joining. Such a change results in dramatic precision loss, as the interval $[0, 10]$ can no longer be recovered for the program in Example 2.3. In order to address this imprecision the narrowing operator $\Delta : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is introduced which repeats another round of fixed point iterations updates *after* the widening, which is often enough to restore the precision (as in the running example).

2.8 Further Examples Of Abstract Domains

A great many abstract domains were proposed to track different program properties.

2.8.1 Octagons

The intervals abstract domain introduced in the previous section is scalable, but is not *relational*: it is not capable of expressing relations between the variables, such as $x = y$. A more expressive

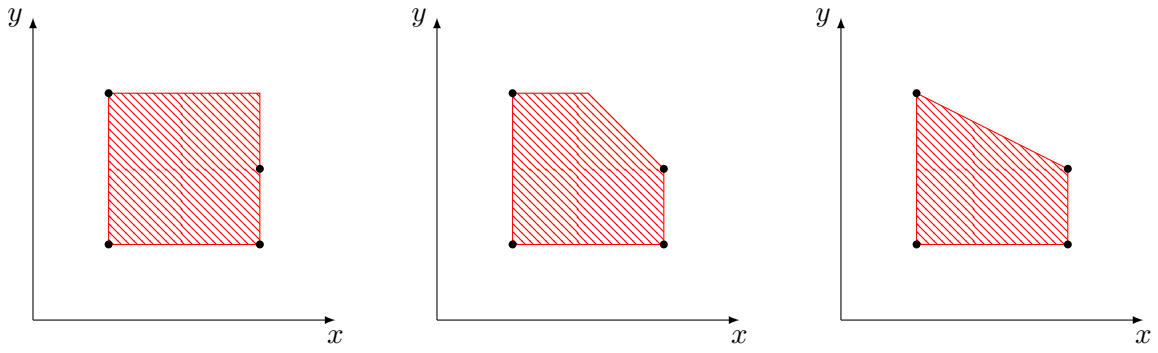
octagons [Min06] abstract domain was proposed by Miné. An element of the octagons abstract domain is the set of bounds on expressions $\pm x \pm y$ for each pair of program variables $x, y \in X$. The resulting shape of an arbitrary non-degenerate domain element for two variables is an octagon, which gives the domain its name. An abstract transformer for the octagon abstract domain can be implemented using Floyd-Warshall algorithm [Min06], and the widening operator can be defined by eliminating all moving constraints.

2.8.2 Polyhedra

The polyhedra abstract domain [CH78] generalizes the convex abstract states further, by allowing an abstract domain element to be any *convex polyhedron*[5] over the program variables. Such an abstract domain can effectively present any *linear convex* property (such as $x+y+z \leq 1 \wedge x \leq 5$), however, it requires a well constructed widening operator in order to work efficiently. A comparison of the expressive power of the intervals, octagons and polyhedra abstract domain is shown in Figure 2.6.

Note that this abstract domain is not a complete lattice and does not form a Galois connection, as the least element does not always exist for all sets of polyhedra (e.g. consider a set of all polyhedra which contain a non-empty circle). Hence it is not possible to construct least abstract transformer for polyhedra. This limitation is acceptable in practice, since due to the usage of widening operators the invariant obtained by abstract interpretation is usually not the smallest possible one.

Many highly optimized libraries such as PPL [BHZ08] are available for abstract interpreters which can perform the required transformations on polyhedra, such as convex hull or projection.



(A) intervals domain (B) octagons domain (C) polyhedra domain

FIGURE 2.6: Comparison of intervals, octagons and polyhedra abstract domains. Black points represent a set of concrete data states $\{\{x:1, y:1\}, \{x:1, y:3\}, \{x:3, y:1\}, \{x:3, y:2\}\}$ over two integral variables $\{x, y\}$, and red shaded lines represent the abstraction in the corresponding abstract domain.

2.8.3 Template Constraints Domains

The family of *template constraints domains* was proposed in a work by Sankaranarayanan [SSM05] et al. as a way to offer a configurable compromise between the scalability and precision.

Unlike the interval and octagon domains, where the shape of propagated constraints is defined by the domain itself, and the polyhedra abstract domain, where the analysis tries to

[5]Some textbooks prefer the name *polytope* for such structures in higher-dimensional space, in this thesis we stick to the name polyhedron.

track *all* the possible linear constraints, a given TCD is parameterized in advance by a vector of functions over $\mathbf{x}$ (*templates*, e.g. $x + 3y$ and $2x + y$), and an element of a domain is a vector of bounds on those functions (e.g. $(1, 5)$ represents $x + 3y \leq 1$ and $2x + y \leq 5$).

An abstract state of a TCD is a vector $(d_1, \ldots, d_m) \in \bar{\mathbb{R}}^n$. The original publication [SSM05] presents the case where all templates are linear and can be defined by a matrix $T \in \mathbb{R}^{n \times \|\mathbf{x}\|}$ (a vector $\mathbf{d}$ concretizes to a set of concrete states which satisfy the constraint $T\mathbf{x} \preceq \mathbf{d}$), and later articles generalize the domain further to non-linear templates [AGG10]. In this thesis we shall operate only over linear templates, and an abstract state $\mathbf{d}$ concretizes to $\{\mathbf{x} \mid \bigwedge_i t_i^\top \mathbf{x} \leq \mathbf{d_i}\}$, where the domain is defined by a matrix of templates T (which we shall also treat like a set of templates).

We allow the use of ∞ and $-\infty$ in the bound in order to represent unbound templates, and unreachable states respectively. A vector $(\infty, \ldots, \infty)$ corresponds to the $\top$ element of a TCD, while a vector containing at least one $-\infty$ entry represents the bottom element $\bot$. An example abstract state of a TCD is given in Figure 2.7.

The domain of *products of intervals* is one instance of TCD, where the templates are $\pm x_i \leq c_i$ for program variables x_i. The domain of *octagons* [Min06] is another, with templates $\pm x_i \pm x_j$ and $\pm x_i$. Any template constraints domain where all templates are linear is a subset of the domain of convex polyhedra [CH78].

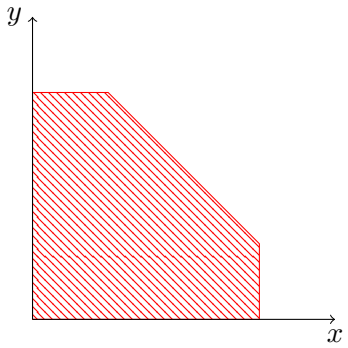


FIGURE 2.7: Example of an element of the template constraints domain defined by a vector of templates $T \equiv (-x, -y, x, y, x + y)$ with bounds $\mathbf{d} \equiv (0, 0, 3, 3, 4)$, which describes the region $0 \leq x \leq 3 \wedge 0 \leq y \leq 3 \wedge x + y \leq 4$. Observe how the lower bound is expressed as an upper bound on a negated expression.

The abstraction in a TCD is defined using a maximization operator: maximizing all templates subject to the constraints introduced by a region. Formally, $\alpha(r)_i \equiv \max t_i^\top \mathbf{x}$ s.t. $\mathbf{x} \in r$. If all templates are linear, abstraction can be performed using linear programming.

Abstract semantics can be also defined directly using maximization modulo the constraints introduced by the previous state and the transition relation. For an operator `OP` the abstract semantics is given by:

$$[\![\texttt{OP}]\!]^\sharp(\mathbf{d})|_i \equiv \max t_i^\top \mathbf{x}' \text{ s.t. } \mathbf{x}' \in [\![\texttt{OP}]\!](\gamma(\mathbf{d})) \tag{2.10}$$

For example, for the abstract state $i \leq 0 \wedge j \leq 0$ under the transition $i' = i + 1 \wedge i \leq 10$ the new abstract state is $i \leq d_i \wedge y \leq d_j$, where $d_i = \max i'$ s.t. $i \leq 0 \wedge j \leq 0 \wedge i' = i + 1 \wedge i < 10 \wedge j' = j$ and d_j is the result of maximizing j' subject to the same constraints. This gets simplified to $i \leq 1 \wedge j \leq 0$.

The templates abstract domain provides a configurable compromise in expressivity for the domains described above. The weakness and the strength of a template constraints domain is

its configurable *precision*: a small set of templates gives rise to an imprecise, yet very efficient analysis, while a larger set can get higher precision at the cost of a more expensive runtime.

As we shall show in Chapter 4, such a configurability can be used to create a refinement in the CEGAR [Cla+00] fashion, which combines both high precision and fast runtime.

2.8.4 Disjunctive Domains

The domains described above are *convex*: abstract states describe convex structures in the program variable state space. However, many programs give rise to non-convex invariants, most notably from boolean variables representing flags (e.g. $b \implies p \leq 0 \vee \neg b \implies p \geq 0$).

Many extensions (e.g. trace partitioning [MR05]) were proposed to the abstract interpretation framework in order to address this limitation. One of the possible approaches is the *boxes* [CGS09] domain, where each abstract state represents a *set* of potentially disjoint intervals. Such a domain can naturally capture disjoint sets of states, such as the disjunction of implications example in the previous paragraph, but it comes with a cost, as there are no "natural" join and widening operators, and heuristics have to be used. These heuristics often result in *non-monotone* behavior, where more *precise* candidate invariant at one location can result in *less* precise invariant at the end.

In this thesis we are primarily interested in *convex* abstract domain, however, it was shown [San+06] that it is possible to obtain a disjunctive inductive invariant in a convex abstract domain by *splitting* the analyzed states (Section 2.10).

2.8.5 Abstract Domain of Numerical Congruences

Another non-convex property which is often relevant for program is *congruence*: integer remainder after dividing a linear expression by an integer constant. Many programs rely on the modulus operator to e.g. achieve a wrap-around effect or execute a certain action on every n^{th} iteration of the loop. Consequently, the congruence [Gra91] was proposed to track such information. In addition, some polyhedra libraries [BHZ08] allow the use of congruence constraints defining the point cloud contained inside the polyhedra (e.g. abstract states representing all points where $x + y = 0 \mod 2$ and $x + y \leq 5$). Our implementation makes use of a simple congruence domain, as described in Section 7.7.1.

2.8.6 Predicate Abstract Domain

The *predicate* abstract domain was introduced in a seminal work by Graf and Saïdi [GS97] as a way to further extend the expressiveness of the abstract domain to arbitrary *predicates*. Let $\mathcal{L}$ be a finite, fixed, set of quantifier-free first order formulas.

We define the abstract domain $\mathcal{D} \equiv 2^{\mathcal{L}} \cup \{\bot\}$ to be a powerset of $\mathcal{L}$, with a partial order defined by the inclusion relation, and the fact that $\bot$ is the least element. The concretization of an element $d \in \mathcal{D}$ is a region where every concrete state models the conjunction of all constraints in d. Similarly, the abstraction of a region r is the smallest element of $\mathcal{D}$ describing every state in r.

Abstraction can be computed using an SMT solver by checking whether a given predicate p is guaranteed to hold at a state described by a formula ϕ (whether $\neg(\phi \implies p)$ is unsatisfiable), and returning a set of all implied predicated.

Observe that $\mathcal{D}$ forms a complete lattice with meet and join defined as intersection and union respectively, and using *syntactical* equality for comparing individual formulas. The syntactic

comparison is an over-approximation as it does not take the formula semantics into account, yet it generates a complete lattice of height $\|\mathcal{L}\| + 1$.

Boolean and Cartesian Abstraction The powerset domain over the set of predicates is referred to as a *cartesian* [BPR03] predicate domain. Another possible predicate abstract domain is a *boolean* one, for which the abstraction is a *disjunction* over all possible conjunctions over predicates (e.g. $p_1 \wedge p_2 \vee \neg p_1 \wedge \neg p_2$, which is not expressible using the cartesian abstraction). Such a domain is considerably more expressive: in fact Ball et al. [BPR03] prove that it is the *most* expressive abstract domain where all atoms are in the language of the given set of predicates. The abstraction for such a domain is considerably more costly, and is performed using the *ALL-SAT* algorithm: finding all models over the set of predicates which are implied by the formula ϕ (it is performed by finding one cube over predicates, blocking it with the additional clause, and iterating until the constraint set becomes unsatisfiable). The *ALL-SAT* procedure potentially requires evaluating up to $2^{\|\mathcal{L}\|}$ SMT queries.

CEGAR and Interpolation Choosing the right predicates can be a difficult trade-off between precision and performance. Many simple properties can be proven by using only a few predicates, yet a large number might be required for verifying intricate manipulations. Clarke et al. have suggested a *counterexample-guided abstraction refinement* (CEGAR [Cla+00]) approach, which combines precision and performance by starting with a most coarse abstraction, and then gradually *refining* it, if it gives rise to a *spurious* (caused by abstraction imprecision) counterexample to property.

Ken McMillan published an influential paper [McM03] advocating the use of *Craig interpolants* [Cra57] to dynamically generate predicates from the infeasible counterexamples.

Definition 2.13 (Interpolant)**.** For two satisfiable formulas a, b where $a \wedge b$ is unsatisfiable, Craig interpolant c is a new formula which has only shared symbols from a and b, and for which $a \implies c$, and $c \implies \neg b$.

Intuitively, Craig interpolant gives the *reason* for why a and b are unsatisfiable together. SMT solvers can generate Craig interpolants e.g. from proofs of unsatisfiability [Hen+04] of concrete error traces. Many approaches in program analysis [McM06] perform predicate abstraction with interpolants, as it provides a semi-decidable procedure for proving safety and finding counterexamples, and doesn't have inherent limitations of many abstract domains described in this section, such as convexity.

2.8.7 Other Domains

All domains described so far were tracking *numeric* properties of the software. However, this is not a conceptual limitation of abstract interpretation: for a example, the domain of *symbolic memory graphs* [DPV13] allows to efficiently analyze many datastructures, and to prove an absence of memory errors in the program.

2.9 Path Focusing and Large-Block Encoding

Traditionally, abstract interpreters store invariant candidates as mappings from CFA nodes to abstract states. As shown in Algorithm 2.1, this is maintained by joining (with subsequent widening) multiple states "arriving to" the same node from different locations. In convex

```
1  int x = input();
2  int out;
3  if (x < 0) {
4      out = -x;
5  } else {
6      out = x;
7  }
8  assume(out >= 1);
9  assert(x != 0);
```

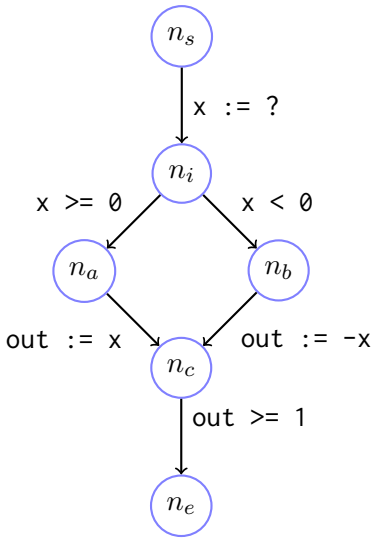


FIGURE 2.8: Motivating Example for Path Focusing

abstract domains, such join often corresponds to a convex hull operator, often resulting in large loss of precision. For example, consider the motivating program and its CFA in Figure 2.8. The program is very simple: it computes the absolute value, and asserts that if the absolute value is bigger than 1, the input must have been non-zero. This assertion represents the invariant stating that $x \neq 0$ holds universally at n_e.

Consider generating inductive invariants using abstract interpretations in the octagons domain for the motivating program. Following the path (n_s, n_i, n_a, n_c) the analysis obtains the abstract state $x \geq 0 \wedge out = x$, and following the path (n_s, n_i, n_b, n_c) the result is $x < 0 \wedge out = -x$. The merge of this two states is however $out \geq 0 \wedge out \geq x$, which becomes $out \geq 1 \wedge out \geq x$, at the node n_e, insufficient for proving $x \neq 0$.

However, the loss of precision is completely unnecessary: as we have shown in Section 2.3, programs with no looping constructs can be converted to a single formula, for which the intersection with an error state may be checked directly using a single SMT query.

Monniaux and Gonnord have shown [MG11] how abstract interpretation can avoid such a precision loss by reducing the number of "intermediate" abstract states and defining the semantics of CFA edges using existentially quantified formulas. A simplified version of the transformation can be done using the following two steps: initially, for each edge, each operator `OP` is replaced with a formula over $(\mathbf{x} \cup \mathbf{x}')$ representing its semantics. After that the operations shown in Figure 2.9 are applied until a fixed point is reached.

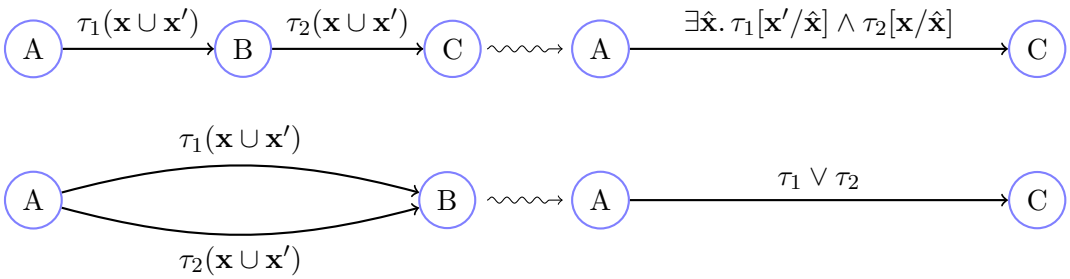


FIGURE 2.9: Transformations required for Large Block Encoding

Observe that since SMT formulas are represented as directed acyclic graphs which can share subformulas, both of these transformations do *not* copy the input formulas, and can be

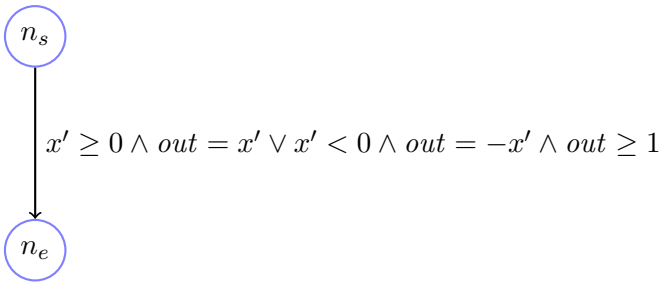


FIGURE 2.10: Motivating Example after Path Focusing Transformation

performed in $O(1)$ time and space.

For a well-structured [ASU86] CFA, repeating this transformation in a fixpoint manner (until no more edges can be merged) leads to a new CFA where the only remaining nodes are start, end, and loop heads. The original publication [MG11] defines a more complex transformation which guarantees that the only remaining nodes form the cut set [Sha79] of an arbitrary input CFA.

The program shown in motivating example after path focusing transformation has two nodes, as shown in Figure 2.10.

As shown in our motivating example, the path focusing procedure can significantly improve the analysis precision. Counterintuitively, it was also shown to often improve the analysis performance [HMM12] by avoiding the creation of intermediate states.

Independently, a similar algorithm was published by Beyer et al. [Bey+09] called *large block encoding*. Unlike path focusing, large block encoding is applied in the context of predicate abstraction with interpolants, or bounded model checking, where the edge semantics is already encoded as a formula. Similarly, the procedure was shown to result in a precision and performance gain.

2.10 Configurable Program Analysis

In this chapter we have described various approach for program analysis: bounded model checking and symbolic execution (Section 2.4), predicate abstraction (Section 2.8.6) and abstract interpretation (Section 2.7). The algorithms used for these approaches seem different, but the underlying theme remains the same: there exists an abstract domain $\mathcal{D}$, and the analysis performs some fixed point iteration for an input CFA P.

Beyer et al. have published a paper describing the Configurable Program Analysis [BHT07] (CPA) algorithm, which describes a *parametrizable* algorithm which can be used to express previously independent approaches to code analysis in the unified framework. The authors show that the primary difference in *model checking*-based approaches (BMC, symbolic execution, lazy abstraction and others) and the abstract interpretation-based approaches is the choice of the *merge* operator. In the abstract interpretation approach (and earlier, for dataflow analysis), two states corresponding to the same node have to be merged (cf. Algorithm 2.1), which often leads to over-approximation, but guarantees convergence. In contrast, model checking based approaches do not join and leave the states separate: as a result, the run of a model-checking-based tool can be presented as a *reachability* tree. By giving each analysis a choice of a merge operator, different algorithms could be presented within the unified CPA framework, running the CPA algorithm shown for completeness in Algorithm 2.2.

The algorithm input is determined by the *parametrization* chosen by the client CPA, which

Algorithm 2.2 CPA Algorithm (taken from [BHT07])

```
1: Input: a CPA (D, transfer, merge, stop)
2: Initial abstract state e_0 ∈ D
3: Output: a set of reachable abstract states
4: Variables: a set reached ⊆ D, a set waitlist ⊆ E
5: waitlist ← {e_0}
6: reached ← {e_0}
7: while waitlist ≠ ∅ do
8:     Pop e from waitlist
9:     for all e' ∈ transfer(e) do
10:        for all e'' ∈ reached do
11:            ▷ Combine with existing abstract state
12:            e_new ← merge(e', e'')
13:            if e_new ≠ e'' then
14:                waitlist ← (waitlist ∪ {e_new}) \ {e''}
15:                reached ← (reached ∪ {e_new}) \ {e''}
16:            end if
17:        end for
18:        ▷ Whether e' is already covered by existing states
19:        if ¬stop(e', reached) then
20:            waitlist ← waitlist ∪ {e'}
21:            reached ← reached ∪ {e'}
22:        end if
23:    end for
24: end while
25: return reached
```

consists of the following components:

- Transfer relation: a function $\mathcal{D} \to 2^{\mathcal{D}}$, defining the abstract semantics (for generality, every state is allowed to have zero or more successors).
- Stop operator: a function $\mathcal{D} \times 2^{\mathcal{D}} \to \mathbb{B}$, defining whether one state is *covered* by a collection of other states.
- Merge operator: a function $\mathcal{D} \times \mathcal{D} \to \mathcal{D}$ generalizing the join operator. If the merge produces the state which does not subsume the input arguments, both states are kept.

Like for Kleene iteration, the overall loop performs the fixed point iterations which iteratively expands the states contained in the *waitlist* (line 7). For each states, all successors under the *transfer relation* are found (line 9), and each successor is merged with states already in the reached set, representing the existing candidate invariant (line 12). If the result of the merge operation does not equal to the successor (line 13), which is the code for "no join", the successor element is replaced with the result of the merge. Finally, if the result of the operation is not covered by the existing elements (line 19), the result is added to *waitlist* and to the *reached* set.

The algorithm shown in Algorithm 2.2 is extremely general: it does not specify whether the analysis runs backwards or forwards (setting the transfer function to weakest precondition, the initial state to the exit node and changing the merge function to perform intersection instead is sufficient to reverse the direction), or whether the analysis states should be grouped by the CFA node, which makes it suitable for describing a large set of program analysis algorithms.

Relation to Abstract Interpretation With a merge operator set to always perform a join, an instance of a configurable program analysis can be seen as an abstract interpretation, defined using the initial state $n_0 \in \mathcal{D}$, transfer relation given by strongest postcondition, join operator $\sqcup_{\mathcal{D}}$, and a coverage check $\preceq_{\mathcal{D}}$. Observe that such definition does not require explicitly defining neither the abstraction $\alpha : \mathcal{R} \to \mathcal{D}$, nor the concretization $\gamma : \mathcal{D} \to \mathcal{R}$, nor even the concrete collecting semantics $\mathcal{R} \to \mathcal{R}$. This property makes the CPA framework suitable for describing abstract domains which do not even have a well-defined abstraction operators, such as the polyhedra domain (Section 2.8.2).

Disjunctive Abstract Domains using Splitting Moreover, the CPA framework allows one to perform analysis in the disjunctive powerset domain while using only the convex abstract transformer by strategically *splitting* the states using the merge operator. For example, using the abstract transformer associated with the interval abstract domain, and the merge operator which always splits the candidate invariant states effectively performs analysis in the disjunctive interval domain.

2.10.1 Composite Configurable Program Analysis

An analysis is usually performed using multiple CPAs, by making use of the `CompositeCPA` parameterization, which wraps a tuple of CPA objects. Such an analysis defines the abstract domain as the product of contained abstract domains, abstract semantics is applied respectively to each component. The *merge* operator joins if all the contained CPAs decide to do so, and splits otherwise, and the *stop* operator checks coverage component-wise. Thus, `CompositeCPA` allows the user to perform the analysis in multiple abstract domains which can *strengthen* each other [CCF13], resulting in the greater overall precision while preserving modularity. For example, the following sub-analyses are often used:

- `LocationCPA` binds successor computation to the outgoing edges, and only allows to merge the states corresponding to the same CFA node.
- `CallstackCPA` keeps track of callstack and performs the dynamic inlining.
- `FunctionPointerCPA` keeps an abstract over-approximation of what functions function pointers can point to, and returns all possible successors on function pointer call.
- `LoopstackCPA` performs dynamic loop unrolling.

The CPA concept is implemented in the CPAchecker [BK11] framework, which includes implementations for many program analysis approaches as CPA parameterizations. All the algorithms presented in the contributions of this thesis are implemented as separate CPA parameterizations inside the CPAchecker tool.

Part II

Theoretical Contributions

Chapter 3

Local Policy Iteration

3.1 Introduction

In this thesis we focus on the *numerical* abstract domains, equipped with a lattice of an infinite height. Such abstract domains require an application of widening operators for guaranteeing the convergence when performing the analysis in abstract interpretation. Many approaches were proposed to combat the imprecision caused by widenings necessary for analysis in such domains, e.g. using more sophisticated widening [GR06], or narrowing iterations [HH12].

However, the majority of such approaches are heuristical and do not give any optimality guarantees. In contrast, the *policy iteration* (also referred to as *strategy iteration*) approach was proposed to address the imprecision, with a promise of finding the *least* inductive invariant expressible in the given abstract domain.

The policy iteration technique dates back to an artificial intelligence research for finding the optimal *strategy* or *policy* in a game expressed by a Markov decision process [How60]. For example, this technique was used in order to program a poker-playing AI [HHS11]. In contrast to *value iteration* (Kleene fixpoint iteration described in Section 2.7), the policy iteration approach iterates on possible *policies*, and converges to the optimal solution in the given abstract domain. Such guarantees come at a cost of imposed restrictions on the abstract semantics and the abstract domain.

In this chapter we present our results on local policy iteration, which is a significantly improved version from the previously published results [KMW16]. Additionally, we present extended background of the policy iteration method for finding inductive invariants.

3.1.1 Motivation

Consider classical abstract interpretation with intervals over the program presented in Figure 3.1.

```
1  int i = 0;
2  while (i < 1000000) {
3      i++;
4  }
```

FIGURE 3.1: Motivating Example

After the first instruction, the analyzer has a *candidate invariant* $i \in [0, 0]$. Going through the loop body it gets $i \in [1, 1]$, thus by least upper bound with the previous state $[0, 0]$ the new candidate invariant is $i \in [0, 1]$. Subsequent *Kleene iterations* yield $[0, 2]$, $[0, 3]$ etc. In order to enforce convergence within a reasonable time, a *widening operator* is used, which extrapolates this sequence to $[0, +\infty)$. Then, a *narrowing iteration* returns a post-image of $[0, +\infty]$ under the constraint $i < 1000000$ yielding $[0, 99999]$.

```
1 int i = 0;
2 while (i != 1000000) {
3     i++;
4 }
```

FIGURE 3.2: Example of Narrowing Breaking Down

In this case, the invariant finally obtained is the best possible, but the same approach yields the suboptimal invariant $[0, +\infty)$ for the slight program modification in Figure 3.2, as the post-image of the interval $[0, +\infty)$ under the constraint $i \neq 1000000$ is still $[0, +\infty)$.

Of course more sophisticated narrowing heuristics can deal with the modified program from Figure 3.2. Yet in general, widenings and narrowings are brittle: a small program change may result in a different analysis behavior. Their result is *non-monotone*: a locally more precise invariant at one point may result in a less precise one elsewhere.

3.1.2 Max-policy iteration

In contrast, max-policy iteration [GS07b] is guaranteed to compute the least *inductive* invariant in the given abstract domain. Note that it does not necessarily output the strongest (potentially non-inductive) invariant in an abstract domain, which in general entails solving the halting problem. To compute the bound d of the invariant $i \leq d$ for the initial example above, it considers that d must satisfy $d = \max i'$ s.t. $(i' = 0) \vee (i' = i + 1 \wedge i < 10000000 \wedge i \leq d)$ and computes the least inductive solution of this equation by successively considering separate cases:

- $d = (\max i' \text{ s.t. } i' = 0) = 0$, which is not inductive, since one can iterate from $i = 0$ to $i = 1$.
- $d = (\max i' \text{ s.t. } i' = i + 1 \wedge i < 1000000 \wedge i \leq d) = 1000000$, which is inductive and consequently a least upper bound on `i`.

Earlier presentations of policy iteration solve a sequence of global convex optimization problems whose unknowns are the bounds (here d) at every program location. Further refinements [GM12] allowed restricting abstraction to a cut-set [Sha79] of program locations (a set of program points such that the control-flow graph contains no cycle once these points are removed), through a combination with *satisfiability modulo theory* (SMT) solving. Nevertheless, a global view of the program was needed, hampering scalability and combinations with other analyses.

Contribution We present the new local-policy-iteration algorithm (LPI) for computing inductive invariants using policy iteration. Our implementation is integrated inside the open-source CPACHECKER [BK11] framework for software verification and uses the maximization-modulo-theory solver νZ [BPF15]. To the best of our knowledge, this is the first policy-iteration implementation that is capable of dealing with C code. We evaluate LPI and show its competitiveness with state-of-the-art analyzers using benchmarks from the International Competition on Software Verification (SV-COMP).

Our solution improves on earlier max-policy approaches:

- Scalability: LPI constructs optimization queries that are at most of the size of the largest loop in the program. At every step we only solve the optimization problem necessary for deriving the *local* candidate invariant. Moreover, casting the algorithm in terms of

standard Kleene worklist iteration allows us to use existing results on optimal iteration orders, improving the performance by avoiding redundant computations.

- Ability to cooperate with other analyses: LPI is defined within the Configurable Program Analysis (CPA) [BHT07] framework. This establishes a common ground with other approaches and allows communicating with other analyses.

- Precision: LPI uses large-block encoding [Bey+09], and thus benefits from the precision offered by SMT solvers, effectively checking executions of loop-free program segments. In Chapter 7 we show how to use LPI with adjustable block encoding [BKW10], which does not require pre-processing, making inter-analysis communicatino easier.

- Counterexample traces: in Section 4.4.1 we show how to generate an abstract reachability tree from the run of LPI analysis, thus obtaining the abstract counterexample trace.

3.1.3 Related Work

Policy iteration is not as widely used as classic abstract interpretation and (bounded) model checking. Roux and Garoche [RG13] addressed a similar problem of embedding the policy-iteration procedure inside an abstract interpreter, however their work has a different focus (finding quadratic invariants on relatively small programs) and the policy-iteration algorithm remains fundamentally unaltered. The tool ReaVer [MS14] also performs policy iteration, but focuses on efficiently dealing with logico-numerical abstract domains; it only operates on Lustre programs. The tool 2LS [Bra+15] applies an approach inspired by policy iteration, combined with k-induction and bounded model checking, yet it does not change the fundamental policy iteration algorithm. The ability to apply policy iteration on strongly connected components one by one was (briefly) mentioned before [Gau+07]. Our work takes the approach much further, as our value-determination problem is more succinct, we apply the principle of locality to the policy-improvement phase, and we formulate policy iteration as a classic fixpoint-iteration algorithm, enabling better performance and communication with other analyses. Finally, it is possible to express the search for an inductive invariant as a nonlinear constraint solving problem [CSS03] or as a quantifier elimination problem [Mon10], but both of these approaches scale poorly.

While this chapter is concerned with max-policy iterations, a similar *min-policy* [Cos+05] algorithm was developed by Goubault et al., which performs descending iterations, continuously refining the inductive invariant. Unlike max-policy, min-policy approaches do not guarantee achieving the global optimum, however every iteration step is an over-approximation and thus the iterations can be terminated early, and furthermore, min-policy was experimentally shown to be more effective for computing invariants over quadratic templates [RG14].

3.1.4 Chapter Overview

We start by deriving the background necessary for the policy iteration algorithm in Section 3.2. Our background presentation is largely unique to this work, and provides a complete and sufficient introduction to policy iteration (original complete presentation of policy iteration and background is also available in the journal paper [GS14] by Gawlitza and Seidl). In Section 3.3 we present the local policy iteration algorithm (LPI), which is an efficient policy iteration implementation stated in the abstract interpretation framework (effectively, we are synthesizing a widening operator which is guaranteed to converge with a least inductive invariant after

finitely many iterations). We outline the extensions and optimizations we have developed for LPI in Section 3.4. Finally, in Section 3.5 we present experimental evaluation of our LPI implementation, and we conclude in Section 3.6.

3.2 Background

3.2.1 Definitions

A vector space S is called *convex* if the line segment between any two points in S lies solely in S: $\forall \mathbf{a}, \mathbf{b} \in S, k \in [0,1]. k\mathbf{a} + (1-k)\mathbf{b} \in S$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *convex* if and only if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the following holds: $\forall k \in \mathbb{R}. f(k\mathbf{x} + (1-k)\mathbf{y}) \leq kf(\mathbf{x}) + (1-k)f(\mathbf{y})$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *concave* if and only if $-f$ is convex.

For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, $\mathbf{a} \preceq \mathbf{b}$ holds iff $\mathbf{b} - \mathbf{a} \in \mathbb{R}^n_+$: that is, the comparison $\leq$ holds for all components of $\mathbf{a}, \mathbf{b}$. We say that a strict inequality $\mathbf{a} \prec \mathbf{b}$ holds iff $\mathbf{a} \preceq \mathbf{b}$, and there exists an $i < n$ such that $\mathbf{a}|_i < \mathbf{b}|_i$. Additionally, we define $\mathbf{a} \ll \mathbf{b}$ to state that the strict inequality $<$ holds component-wise for *all* components of $\mathbf{a}, \mathbf{b}$. Lattice operators $\sqcup : \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^n$ and $\sqcap : \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^n$, are defined to return vectors of pairwise maximums or minimums of all input components respectively.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *monotone* if and only if for all $a, b \in \mathbb{R}^n$, $a \preceq b$ implies $f(a) \leq f(b)$. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called monotone if and only if every component of f is monotone. Strongest postcondition operator for a domain where the partial order is given by the element-wise vector comparison is always monotone, as larger elements have larger post-images. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called convex if and only if every component $f_i : \mathbb{R}^n \to \mathbb{R}$ is convex.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is a *pointwise maximum* over a family of functions F with the same signature if and only if for all $\mathbf{x} \in \mathbb{R}^n$, $f(\mathbf{x}) = \max_{g \in F} g(\mathbf{x})$. We generalize this notion to functions $\mathbb{R}^n \to \mathbb{R}^n$ by requiring that for all $\mathbf{x} \in \mathbb{R}^n$, there exists $g \in F$ such that for all $g' \in F$, $f(\mathbf{x}) = g(\mathbf{x}) \succeq g'(\mathbf{x})$). We refer to functions which are a pointwise maximum over a finite set F as having a *selection property*: that is, such a function f for every input $\mathbf{x} \in \mathbb{R}^n$ effectively *selects* $g \in F$ to produce an output ($f(\mathbf{x}) = g(\mathbf{x})$).

Consider the general optimization problem:

$$\min f(\mathbf{x}) \text{ s.t. } a(\mathbf{x}) \leq \mathbf{0} \wedge b(\mathbf{x}) = \mathbf{0} \tag{3.1}$$

The problem (3.1) is referred to as a *convex optimization problem* if and only if both the *objective* function f, and a function a defining the *feasible set* are convex, and a function b is affine. Many important classes of convex optimization problems, such as positive semidefinite programming (SDP, the constraint set is given by a positive semidefinite matrix), or linear programming (LP, both the objective and the feasible set are affine) are solvable in polynomial time. Modern solvers can analyze convex problems with tens of thousands of variables [Gur16; IBM10].

3.2.2 Least Invariant as a Convex Optimization Problem

The classical problem in program analysis is searching for the smallest inductive invariant in the domain $\mathcal{D}$. We assume $\mathcal{D} \subseteq \bar{\mathbb{R}}^n$ ($\bar{\mathbb{R}} \equiv \mathbb{R} \cup \{+\infty, -\infty\}$): that is, each domain element is representable as a tuple of n extended reals (e.g. a template constraints domain, Section 2.8.3). Consider analyzing a program with a single initial state $\mathbf{a_0} \in \mathcal{D}$, and a single transition τ with abstract semantics given by $\tau^\sharp : \mathcal{D} \to \mathcal{D}$ (Section 2.7.2).

In order to make the problem of finding the least invariant decidable, we need to introduce restrictions both on the abstract domain and the transition relation. We require $\tau^\sharp(\mathbf{a_0}) > \mathbf{a_0}$, which holds unless $\mathbf{a_0}$ is an inductive invariant.

Additionally, we restrict the allowed abstract transformers to a set of concave functions $\bar{\mathbb{R}}^n \to \bar{\mathbb{R}}^n$. In Section 3.2.3 we shall show that linear programs over rationals in a template constraints domain satisfy the requirement.

Traditionally, abstract interpretation attempts to find the least fixed point by explicitly computing the sequence $\mathbf{a_0}, \mathbf{a_0} \sqcup \tau^\sharp(\mathbf{a_0}), \mathbf{a_0} \sqcup \tau^\sharp(\mathbf{a_0}) \sqcup \tau^\sharp(\tau^\sharp(\mathbf{a_0})), \ldots$ until a fixed point, potentially using widening to enforce convergence (cf. Algorithm 2.1). Such a computation is referred to as *value* iteration.

In contrast, we aim to find the least inductive invariant exactly by exploiting the concavity property. In order to find the least post-$\mathbf{a_0}$ fixed point $\mathbf{d}^*$ of a concave function $\tau^\sharp : \mathbb{R}^n \to \mathbb{R}^n$ we need to solve the following optimization problem[1]:

$$\mathbf{d}^* \equiv \min \mathbf{d} \text{ s.t. } \tau^\sharp(\mathbf{d}) \preceq \mathbf{d} \wedge \mathbf{a_0} \preceq \mathbf{d} \tag{3.2}$$

Minimizing vectors is not a well defined operation, as $\mathbb{R}^n$ only imposes a partial order. However, from Tarski's fixed point theorem [Tar55] for monotone functions on complete lattices we know that such a minimum denoting least fixpoint larger than $\mathbf{a_0}$ exists.

The feasible set of Equation 3.2 is neither concave nor convex. Yet consider the following optimization problem, which finds the largest fixpoint [Tar55]:

$$\hat{\mathbf{d}}^* \equiv \max \mathbf{d} \text{ s.t. } \tau^\sharp(\mathbf{d}) \succeq \mathbf{d} \tag{3.3}$$

Equation 3.3 is a convex optimization problem, as $-\tau^\sharp$ is convex, affine function $\mathbf{d}$ does not affect the convexity, and any sub-level set of a convex function is convex. Furthermore, for a monotone and concave function these fixed points coincide.

Theorem 3.1. For a monotone concave $f : \mathbb{R}^n \to \mathbb{R}^n$, satisfying $f(\mathbf{a}) \gg \mathbf{a}$ there exists at most one fixed point $\mathbf{b}$ for which $\mathbf{b} \succ \mathbf{a}$ holds.

Proof. Without loss of generality, let $\mathbf{b}$ be the least fixed point of f. As $f(\mathbf{a}) \gg \mathbf{a}$, from Kleene [Kle52] theorem we have $\mathbf{b} \succeq f(\mathbf{a})$. Consider a line l from $\mathbf{a}$ to $\mathbf{b}$. As $\mathbf{a} \ll f(\mathbf{a}) \preceq \mathbf{b}$ holds, *all* coordinates strictly increase along l in the direction of $\mathbf{b} - \mathbf{a}$. Suppose there exists a fixpoint $\mathbf{c} \neq \mathbf{b}$. Without loss of generality we can assume that $\mathbf{c} \succ \mathbf{b}$ (as $(\mathbf{c} \sqcup \mathbf{b}) \succ \mathbf{b}$ is also a fixed point by Tarski [Tar55] theorem).

As l is increasing in all coordinates, there exists a point $\mathbf{d} \in l$ such that $\mathbf{d} \succeq \mathbf{c}$ and moreover for some dimension j the equality $\mathbf{d}|_j = \mathbf{c}|_j$ holds. Then necessarily have $\mathbf{c}|_j > \mathbf{b}|_j$, as otherwise $\mathbf{d}|_j = \mathbf{c}|_j = \mathbf{b}|_j$ implies that $\mathbf{d}$ and $\mathbf{b}$ coincide (as all dimensions increase along l), which contradicts the fact that $\mathbf{d} \succeq \mathbf{c} > \mathbf{b}$.

From concavity, $f(\mathbf{d})|_j < \mathbf{d}|_j$ (as there exists a strictly smaller pre-fixpoint on the same line). Yet from monotonicity $f(\mathbf{d})|_j \geq f(\mathbf{c})|_j = \mathbf{c}|_j = \mathbf{d}|_j$, yielding a contradiction $\mathbf{d}|_j < \mathbf{d}|_j$. Thus the fixpoint $\mathbf{c} \neq \mathbf{b}$ does not exist, and $\mathbf{b}$ is the unique post-$\mathbf{a}$ fixpoint. □

Thus we can find the least fixed point by solving the convex optimization problem of Equation 3.3.

[1] For precise treatment of infinities an interested reader can refer to [GS14].

3.2.3 Template Constraints Domains

In this section we describe under which conditions the abstract transformer defined by a CFA P and a template constraints domain described in Section 2.8.3 is concave.

Recall that abstract semantics for an operator `OP` and a TCD defined using a vector of templates $(t_1, \ldots, t_n)$ is:

$$[\![\texttt{OP}]\!]^\sharp(\mathbf{d})_i \equiv \sup t_i^\top \mathbf{x}' \text{ s.t. } \mathbf{x}' \in [\![\texttt{OP}]\!](\gamma(\mathbf{d})) \tag{3.4}$$

In order to prove the concavity of Equation 3.4, we give the formal abstract semantics definition for the programming language defined in Figure 2.1.

Firstly, we define the helper evaluation function *eval* for numerical and boolean expressions: $eval(\texttt{<expr>}) : (\mathbf{x} \to \mathbb{R}) \to \mathbb{R}$. This function performs the usual evaluation on a vector of values representing program variables. We define evaluation of boolean expressions in the same way: $eval(\texttt{<bool\_expr>}) : (\mathbf{x} \to \mathbb{R}) \to \mathbb{B}$. The evaluation of a boolean expression is a function which takes the value for all program variables and returns "true" if and only if the set of constraints is satisfied. With the helper *eval* function in place we define the evaluation of a statement to return a formula over $\mathbf{x} \cup \mathbf{x}'$ which evaluates to $\top$ if and only if the assignment to primed and unprimed variables corresponds to a valid transition. Formally, $eval(\texttt{<stmt>}) : (\mathbf{x} \to \mathbb{R}) \to (\mathbf{x}' \to \mathbb{R}) \to \mathbb{B}$. Thus we define the following evaluation rules (two input arguments represent regions over unprimed and primed variables respectively):

$$\begin{aligned}
eval(\texttt{x := input()}) &\equiv \lambda r, r'.\, \Big(\forall v \in (\mathbf{x} \setminus x).\, r(v) = r'(v)\Big) \\
eval(\texttt{x := e}) &\equiv \lambda r, r'.\, \Big(r'(x) = eval(e)(r) \wedge (\forall v \in (\mathbf{x} \setminus x).\, r(v) = r'(v))\Big) \\
eval(\texttt{assume(c)}) &\equiv \lambda r, r'.\, \Big(eval(c)(r) \wedge (\forall v \in \mathbf{x}.\, r(v) = r'(v))\Big)
\end{aligned} \tag{3.5}$$

Hence we can define the abstract semantics of a template constraints domain for a given statement (smallest upper bound for each template after the statement execution given the upper bounds on each templates before the execution) without resorting to the concrete semantics:

$$\begin{aligned}
&[\![\texttt{OP}]\!]^\sharp : \mathbb{R}^n \to \mathbb{R}^n \\
&[\![\texttt{OP}]\!]^\sharp_i \equiv \lambda \mathbf{d}.\, \sup t_i^\top \mathbf{x}' \text{ s.t. } eval(\texttt{OP})(\mathbf{x} \cup \mathbf{x}') \wedge \bigwedge_k t_k^\top \mathbf{x} \leq \mathbf{d}_k
\end{aligned} \tag{3.6}$$

The definition above allows us to formulate the condition for abstract semantics concavity.

Theorem 3.2 (TCD Concavity)**.** The abstract semantics of a template constraints domain is a concave function if all the template functions are linear, the only allowed comparison operators are `>=, ==`, boolean expressions do not contain any disjunctions or negations, and the *eval* function defined in (3.5) is linear for all numerical expressions occurring in the program.

With the constraints introduced in Theorem 3.2 the optimization problem in Equation 3.5 is a parametric linear program. The concavity proof for such a function is given in [GS14, Lemma 25].

3.2.4 Examples of Solvable Programs

For a restricted set of program semantics and initial conditions, we have shown how the problem of finding a least inductive invariant is equivalent to a convex optimization problem, which can be often directly solved by an LP or SDP solver. Let us consider the implications of this result and the programs for which we can now obtain explicit fixed points immediately.

Example 3.1 (Single Affine Transition)**.** Programs with a single looping transition consisting of a linear guard and any number of updates fall into this category. Consider analyzing the following program in the template constraints domain with a single template x:

```
1  int x = 0;
2  while (x < 100)
3      x++;
```

The abstract semantics of the loop transition including the guard and the update is given as a function from a previous to the new upper bound on `x`:

$$f(d) \equiv (\max x' \text{ s.t. } x' = x + 1 \wedge x \leq 99 \wedge x \leq d) \tag{3.7}$$

Using Theorem 3.1, the fixed point representing the upper bound may be found by solving the following optimization problem:

$$f(d) \equiv (\max f(x) \text{ s.t. } f(x) \geq x) \tag{3.8}$$

By combining Equation 3.7 and Equation 3.8 and dropping inner maximization due to redundancy we get the following LP:

$$\max x' \text{ s.t. } x' \geq x \wedge x' = x + 1 \wedge x \leq 99 \tag{3.9}$$

resulting in the upper bound $x \leq 100$. The lower bound is 0, given by the upper bound on the template $-x$, derived from the initial condition, and inductive under the loop.

Of course, the example program in Example 3.1 is trivial. However, using Theorem 3.1 we can already find interesting invariants not readily available to standard abstract interpretation techniques:

Example 3.2 (A More Interesting Invariant)**.** Again, consider analyzing in intervals the following program with a single variable, which performs a non-deterministic number of iterations:

```
1  double x = 0;
2  while (input()) {
3      x = x / 2 + 1;
4  }
```

The lower bound inductively stays at zero, while in order to find the upper bound d we have to solve another LP problem, obtained in the same way as in the Example 3.1:

$$d \equiv (\max x' \text{ s.t. } x' \geq x \wedge x' = x/2 + 1) \tag{3.10}$$

resulting in a non-trivial inductive invariant $x \in [0, 2]$.

3.2.5 Max Policy Iteration Algorithm

The class of programs for which an inductive invariant can be found with a single optimization query is very small, as an overall transition relation $\tau^\sharp$ is seldom concave. For example, consider

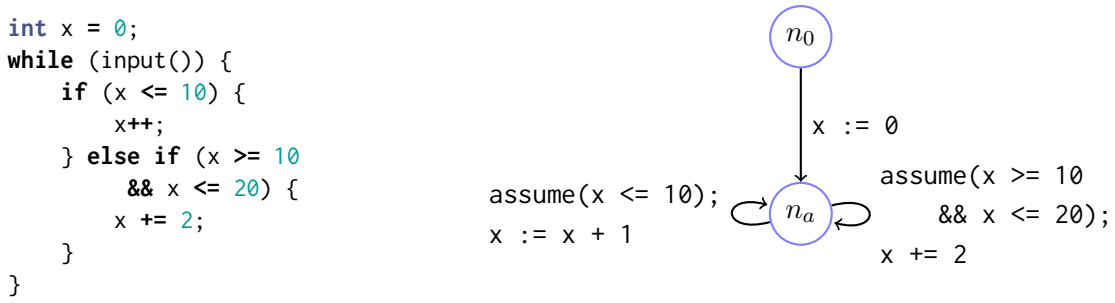


FIGURE 3.3: Example Program for Policy Iteration Demonstration

analyzing a program shown in Figure 3.3 with a single template $T \equiv \{x\}$. Let $\tau_1^\sharp : \mathbb{R} \to \mathbb{R}$ and $\tau_2^\sharp : \mathbb{R} \to \mathbb{R}$ denote the abstract semantics of statements associated with the first and second conditions in the loop respectively (recall that abstract semantics gives a new upper bound on x as a function of a previous upper bound after one transition). Their definition is:

$$\begin{aligned} \tau_1^\sharp(d) &\equiv \max x' \text{ s.t. } x \leq 10 \wedge x' = x + 1 \wedge x \leq d \\ \tau_2^\sharp(d) &\equiv \max x' \text{ s.t. } x \geq 10 \wedge x \leq 20 \wedge x' = x + 2 \wedge x \leq d \end{aligned} \tag{3.11}$$

And the resulting abstract transition $\tau^\sharp$ is a maximum over $\tau_1^\sharp, \tau_2^\sharp$:

$$\tau^\sharp(d) \equiv \max\{\tau_1^\sharp(d), \tau_2^\sharp(d)\} \tag{3.12}$$

By definition, $\tau^\sharp$ is a pointwise maximum over functions $\tau_1^\sharp, \tau_2^\sharp$ which are both monotone and concave. Observe that this is the case for programs with a single node and a single template, where multiple concave transitions are allowed. In this section we show how the policy iteration algorithm finds the least inductive invariant for programs where the abstract semantics of the transition relation is a pointwise maximum over a finite set of concave functions. We refer to such functions as *policies* or *under-approximations* of $\tau^\sharp$.

In general, we are looking for the least fixed point $\mathbf{d}^* \in \mathbb{R}^n$ of a function $\tau^\sharp$ which is greater than the initial state $\mathbf{a_0} \in \mathbb{R}^n$. By adding the initial policy $\tau_0^\sharp \equiv \lambda \mathbf{d}.\, \mathbf{a_0}$ to $\mathcal{F}$, we can ignore the initiation condition $\mathbf{d}^* \succeq \mathbf{a_0}$, as it would be implied by the consecution requirement $\tau^\sharp(\mathbf{d}) \preceq \mathbf{d}$, resulting in the following optimization problem:

$$\mathbf{d}^* \equiv \min \|\mathbf{d}\| \text{ s.t. } \tau^\sharp(\mathbf{d}) \preceq \mathbf{d} \tag{3.13}$$

The function $\tau^\sharp$ is not concave and instead is given as pointwise maximum of a set $\mathcal{F}$ of finitely many policies $(\tau_1^\sharp, \ldots, \tau_n^\sharp)$:

$$\tau^\sharp \equiv \max \mathcal{F} \tag{3.14}$$

Lemma 3.1. For any $\hat{\mathbf{d}} \ll \mathbf{d}^*$ for which $\hat{\mathbf{d}} \ll \tau^\sharp(\hat{\mathbf{d}})$ holds, there exists $\tau_i^\sharp \in \mathcal{F}$ such that $\hat{\mathbf{d}} \ll \tau_i^\sharp(\hat{\mathbf{d}})$ and furthermore $\mathbf{d}_i^* \equiv (\max \|\mathbf{d}\| \text{ s.t. } \tau_i^\sharp(\mathbf{d}) \succeq \mathbf{d})$ is less or equal to $\mathbf{d}^*$.

Proof. The function $\tau^\sharp$ is a pointwise maximum over $\mathcal{F}$. From selection property, there exists $\tau_i^\sharp \in \mathcal{F}$, such that $\tau_i^\sharp(\hat{\mathbf{d}}) = \tau^\sharp(\hat{\mathbf{d}})$. By Theorem 3.1, $\mathbf{d}_i^*$ is equal to $\mu_{\succeq \hat{\mathbf{d}}} \tau_i^\sharp$. Then $\tau^\sharp(\mathbf{d}_i^*) \succeq \mathbf{d}_i^*$ (as for all $\mathbf{d}$, $\tau^\sharp(\mathbf{d}) \succeq \tau_i^\sharp(\mathbf{d})$), from which we get $\mathbf{d}^* \succeq \mathbf{d}_i^*$, as $\mathbf{d}^*$ is the least fixed point of $\tau^\sharp$. □

Lemma 3.1 gives us a way of solving the non-convex optimization problem in Equation 3.13:

Algorithm 3.1 Policy Iteration Algorithm

Input: set of policies $\mathcal{F} \equiv (f_0, \ldots, f_n)$, initial policy $f_0 \equiv \lambda \mathbf{x}.\, \mathbf{x_0}$, $f_0 \in \mathcal{F}$.
Output: $x^* \equiv \min \|\mathbf{x}\|$ s.t. $(\max \mathcal{F})(x) \preceq \mathbf{x}$
$f_i \leftarrow f_0$
$\mathbf{x_i^*} = \mathbf{x_0}$
while $\neg(x_i^* \succeq f(x_i^*))$ **do**
    ▷ Policy Improvement
    $f_i \leftarrow$ function in $\mathcal{F}$ for which $f_i(x_i^*) = f(x_i^*)$
    ▷ Value Determination
    $x_i^* \leftarrow \max \|\mathbf{x}\|$ s.t. $\mathbf{x} \preceq f_i(\mathbf{x})$
end while
return x_i^*

if we have a point $\hat{\mathbf{d}}$ a priori known to be smaller then $\mathbf{d}^*$, we can find the corresponding $\tau_i^\sharp$ and its post-$\hat{\mathbf{d}}$ fixed point, which is also known to be smaller or equal to $\mathbf{d}^*$. Each such $\tau_i^\sharp$ we encounter during the iteration process is called a *feasible policy*. Thus the process can continue until we converge to the policy $\tau^{\sharp *}$ from which we can finally deduce $\mathbf{d}^*$. This process is known as a *policy iteration*, and the step of choosing a new $\tau_i^\sharp$ (*any* policy $\tau_i^\sharp$ for which $\tau_i^\sharp(\hat{\mathbf{d}}) \gg \mathbf{d}$ can be chosen) is called *policy improvement* while the step of generating new $\mathbf{d_i^*}$ is called *value determination*. We show the pseudocode for this process in Algorithm 3.1.

We revisit our example from Figure 3.3 with the policy iteration algorithm. We start with a value 0. The initial feasible policy is:

$$\tau_1^\sharp(d) \equiv \max x' \text{ s.t. } x \leq 10 \wedge x' = x + 1 \wedge x \leq d$$

as $\tau_1^\sharp(0) = 1$ which is a strict improvement over 0. The value determination process on $\tau_1^\sharp$ solves a linear programming problem:

$$\max x' \text{ s.t. } x \leq 10 \wedge x' = x + 1 \wedge x' \geq x$$

yielding a new value of 11. With this value, the policy $\tau_2^\sharp$ becomes feasible, as it $\tau_2^\sharp(11) = 12$. In order to find the value of $\tau_2^\sharp$ we solve the second linear programming problem:

$$\max x' \text{ s.t. } x \leq 10 \wedge x \leq 20 \wedge x' = x + 2 \wedge x' \geq x$$

which returns a new bound 22. There are no new feasible policies, and the result $x \leq 22$ is the least inductive invariant expressible in the given domain.

For illustration purposes, we additionally present a detailed application of a policy iteration algorithm on a non-convex optimization problem consisting of five policies in Figure 3.4. For comparison, we perform value iterations on the same problem, visualized in Figure 3.9.

Theorem 3.3 (Policy Iteration Convergence)**.** The algorithm shown in Algorithm 3.1 converges within at most $\|\mathcal{F}\|$ convex optimization queries with the smallest post-$\mathbf{x_0}$ fixed point.

Proof. The optimality follows from Lemma 3.1, as none of the intermediate fixed points can overshoot $\mathbf{x}^*$. The convergence follows from the fact that for any two subsequent intermediate values $\mathbf{x_i^*}$, $\mathbf{x_{i+1}^*}$ it follows that $\mathbf{x_{i+1}^*} \succ \mathbf{x_i^*}$, by definition. As there is only a such fixed point for each $f \in \mathcal{F}$ it imposes an order on policies, and each policy is considered at most once. □

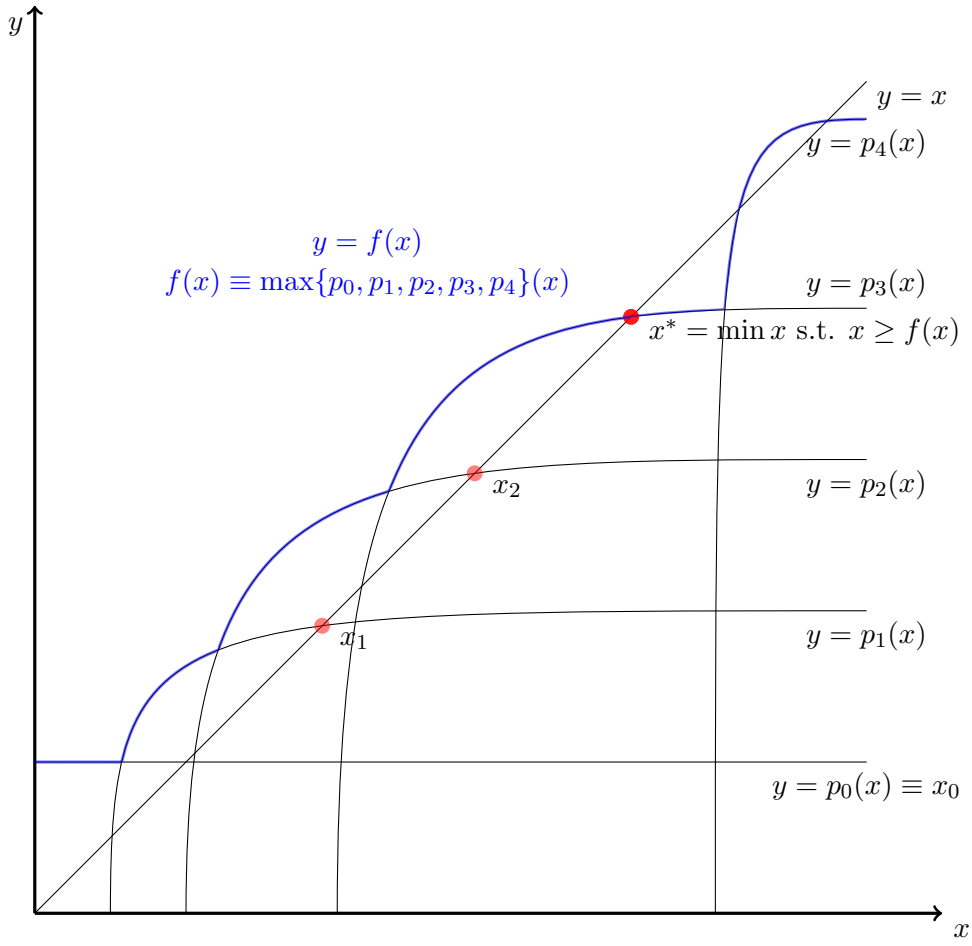


FIGURE 3.4: Visualization of the policy iteration algorithm. We are minimizing x subject to the constraint $x \geq f(x)$, where $f(x) : \mathbb{R} \to \mathbb{R}$ is a point-wise maximum of five concave functions (referred to as *policies*) p_0, p_1, p_2, p_3, and p_4. The function p_0 is a constant and represents the initial condition. Observe that adding the initial condition to the list of policies makes the initiation condition $x \geq x_0$ redundant, as universally $x \geq f(x) \geq x_0$. The iteration starts at the initial policy p_0 which is convex, and its value is x_0 for all inputs. However, by checking whether $f(x_0)$ is smaller than x_0 we discover that the inductiveness condition does not hold. As f has the selection property (for all x, $f(x) = p_i(x)$ for some i), we can find our next policy from $f(x_0)$. Suppose this is p_1. Using Theorem 3.1 we find the local optimum by solving the convex optimization problem $\max x$ s.t. $x \geq p_1(x)$. This gives us the point x_1 shown on the figure, referred to as the *value* of the policy. Again, by substituting x_1 into f, we observe that $f(x_1) > x_1$, from which we derive the third policy p_2. For p_2 we find the local value x_2, and we again discover that it does not satisfy $x \geq f(x)$. We again find the local optimum x^* for the policy p_3, but now by substitution we get $f(x^*) = x^*$, hence x^* is the global optimum. Observe that we did not need to examine the policy p_4, and that moreover, the value of the policy p_4 is larger than the global optimum x^*.

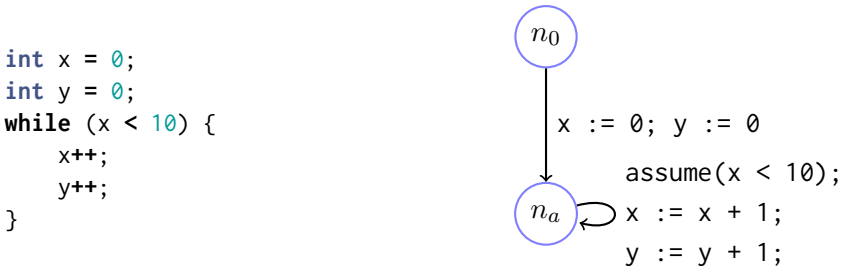


FIGURE 3.5: Program Requiring Different Policy Per Each Template

3.2.6 Selecting Multiple Policies

The assumption that $\tau^\sharp$ is a pointwise maximum over a set of transitions associated with separate statements we have used in Section 3.2.5 is too restrictive, and does not hold in general in the presence of multiple templates. In this section we relax this assumption by allowing the iteration process to select a different policy per each template (and additionally per each node) provided that the feasibility criterion (application of a new policy gives the value strictly greater than the current one) is satisfied for each policy.

As usual, we start with a motivating example. Consider analyzing the program shown in Figure 3.5 with a template set $T \equiv \{x, x-y\}$. By the initialization condition, the templates are always unbounded at the CFA node n_0, and we are only interested in the bounds at n_a. The abstract semantics function

$$\tau^\sharp : \mathbb{R}^2 \to \mathbb{R}^2 \tag{3.15}$$

returns new bounds on x and $x-y$ given previous bounds on x and $x-y$ respectively as an input.

Even though the abstract semantics of each transition associated with each edge is concave, the function $\tau^\sharp$ is not due to a disjunction caused by the two incoming transitions:

$$\begin{aligned} c_i &\equiv (x' = 0 \wedge y' = 0) \\ c_l &\equiv (x < 10 \wedge x' = x + 1 \wedge y' = y + 1) \\ \tau^\sharp(d_1, d_2)|_1 &\equiv \max x' \text{ s.t. } (c_i \vee c_l) \wedge x \leq d_1 \wedge x - y \leq d_2 \\ \tau^\sharp(d_1, d_2)|_2 &\equiv \max x' - y' \text{ s.t. } (c_i \vee c_l) \wedge x \leq d_1 \wedge x - y \leq d_2 \end{aligned} \tag{3.16}$$

Let $\tau_i : \mathbb{R}^2 \to \mathbb{R}^2$ and $\tau_l : \mathbb{R}^2 \to \mathbb{R}^2$ be the functions representing abstract transition relation for the initial and looping transition relations respectively:

$$\begin{aligned} \tau_i(d_1, d_2) &\equiv (0, 0) \\ \tau_l(d_1, d_2)|_1 &\equiv \max x' \text{ s.t. } c_l \wedge x \leq d_1 \wedge x - y \leq d_2 \\ \tau_l(d_1, d_2)|_2 &\equiv \max x' - y' \text{ s.t. } c_l \wedge x \leq d_1 \wedge x - y \leq d_2) \end{aligned} \tag{3.17}$$

Observe that abstract semantics $\tau^\sharp$ is *not* a pointwise maximum over $F \equiv \{\tau_i, \tau_l\}$ due to the fact that maximums for different templates may be reached on the *different* elements of F. For example $\tau^\sharp(0, -1) = (1, 0)$, while $\tau_i(0, -1) = (0, 0)$ and $\tau_l(0, -1) = (1, -1)$. This happens as different optimization directions *interfere*, and the optimum occurs on different disjuncts for different objectives. Hence we can not use F as a set of possible policies.

Instead, we construct a new set of policies $\mathcal{F}$ which is constructed from all possible combi-

nations of templates and the elements of F.

We first define a *new* tuple of objective functions $\mathbf{o}$ where all objectives are guaranteed to be *independent* and not share variables. The tuple $\mathbf{o}$ is generated from T by priming and prefixing each variable in each template with a fresh index corresponding to the template: e.g. $\mathbf{o} \equiv (x'_1, x'_2 - y'_2)$ in our example.

Let $M \subseteq T \times F$ be the set of all possible mappings from T to F:

$$M \equiv \Big\{\{x : \tau_i, x - y : \tau_i\}, \{x : \tau_i, x - y : \tau_l\}, \{x : \tau_l, x - y : \tau_i\}, \{x : \tau_l, x - y : \tau_l\}\Big\} \tag{3.18}$$

From each $m \in M$ we generate a new element π_m of $\mathcal{F}$:

$$\pi_m \equiv \max \mathbf{o} \text{ s.t. } \bigwedge_{(t_i \mapsto c_i) \in m} c_i[\mathbf{x}/t_i.\mathbf{x}] \tag{3.19}$$

where the a renaming adds a prefix derived from t_i to every free variable in c_i.

For example, the mapping $\{x : \tau_i, x - y : \tau_i\} \in M$ gets converted to the following policy (for readability, the prefix 1 corresponds to the template x, and the prefix 2 corresponds to the template $x - y$):

$$\pi_{(\tau_i,\tau_i)}(d_1, d_2) \equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } x'_1 = 0 \wedge y'_1 = 0 \wedge x'_2 = 0 \wedge y'_2 = 0 \tag{3.20}$$

The full set of policies

$$\mathcal{F} \equiv \{\pi_{\tau_i,\tau_i}, \pi_{\tau_i,\tau_l}, \pi_{\tau_l,\tau_i}, \pi_{\tau_l,\tau_l}\} \tag{3.21}$$

is given in Equation 3.22 using a set of helper constraint variables c:

$$\begin{aligned}
c^i_1 &\equiv x'_1 = 0 \wedge y'_1 = 0 \\
c^i_2 &\equiv x'_2 = 0 \wedge y'_2 = 0 \\
c^l_1 &\equiv x_1 < 10 \wedge x_1 \leq d_1 \wedge (x_1 - y_1) \leq d_2 \wedge x'_1 = x_1 + 1 \wedge y'_1 = y_1 + 1 \\
c^l_2 &\equiv x_2 < 10 \wedge x_2 \leq d_1 \wedge (x_2 - y_2) \leq d_2 \wedge x'_2 = x_2 + 1 \wedge y'_2 = y_2 + 1 \\
\pi_{\tau_i,\tau_i}(d_1, d_2) &\equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } c^i_1 \wedge c^i_2 \\
\pi_{\tau_i,\tau_l}(d_1, d_2) &\equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } c^i_1 \wedge c^l_2 \\
\pi_{\tau_l,\tau_i}(d_1, d_2) &\equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } c^l_1 \wedge c^i_2 \\
\pi_{\tau_l,\tau_l}(d_1, d_2) &\equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } c^l_1 \wedge c^l_2
\end{aligned} \tag{3.22}$$

Observe that by creating a set of fresh variables per each optimization objective we have stopped the objectives from "interfering" with each other, thus the function $\tau^\sharp$ is equal to the pointwise maximum over the set of policies $\mathcal{F}$:

$$\tau^\sharp = \max \mathcal{F} \tag{3.23}$$

We abuse the notation and treat $\mathcal{F}$ as the set of constraints of the contained functions, as all the optimization objectives contained in $\mathbf{o}$ are the same. Then we can drop the inner maximum due to redundancy and write Equation 3.23 as:

$$\tau^\sharp \equiv \max(x'_1, x'_2 - y'_2) \text{ s.t. } \mathcal{F} \tag{3.24}$$

Generating $\mathcal{F}$ comes at a cost: the size of the set of exploded policies is exponential in the

```
1  int i=0;
2  while (input()) {
3      int k=0;
4      while (input()) {
5          assert(k <= 1000);
6          if (k == 1000) break;
7          k++;
8      }
9      assert(i <= 1000);
10     if (i == 1000) break;
11     i++;
12 }
```

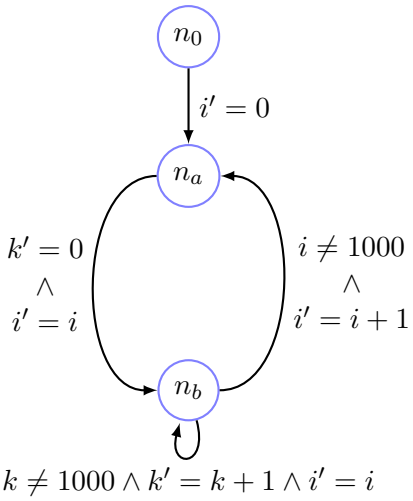


FIGURE 3.6: Running Example: C program and the corresponding CFA after the application of large block encoding (Section 2.9).

number of templates. Yet we do not generate $\mathcal{F}$ explicitly, but instead we create it *implicitly during the iteration*, by *choosing a different (partial) policy* for each template during each policy improvement step[2]. The value determination step is performed on a function from $\mathcal{F}$, as it needs to combine different policies in a single linear programming query. We apply the same approach for generating different invariants at different CFA nodes, by allowing a different policy to be selected per each node and per each template.

Going back to out example, the iteration process proceeds as follows: we start with an initial value $(0, 0)$. The bound for the template $x - y$ can not be increased any further, and stays associated with the (partial) policy τ_i. For the bound for x we choose a new policy τ_l based on a local improvement $1 > 0$. We run value determination on the policy π_{τ_l,τ_i} given by the map $\{x : \tau_i, x - y : \tau_l\}$, which requires solving the optimization problem

$$\max(d_1, d_2) \text{ s.t. } \pi_{(\tau_l,\tau_i)} \wedge d_1 \leq x'_1 \wedge d_2 \leq (x'_2 - y'_2) \tag{3.25}$$

yielding the final inductive invariant $x \leq 10 \wedge x - y \leq 0$.

Non-Strict Improvement Properties Observe that if we allow choosing a different policy per each template, we may often end up in a situation where the values for some templates are updated, while others remain constant. That has actually happened in our example, when the bound on x was increased, but the bound on $x - y$ was left constant, which violates the required condition of Lemma 3.1. In our example, we have found the fixed point by only using the value determination on the partial policies which were updated, and leaving the values which remained the same constant.

The proof of correctness for such a case is considerably more complicated, and can be found by an interesting reader in the original work by Gawlitza and Seidl [GS14].

3.2.7 Analyzing the Running Example with Policy Iteration

[2]Strictly speaking, the chosen function is not a policy, but the mapping from templates to chosen functions defines one. Yet we abuse the notation by calling "partial policies" policies as well.

Example 3.3 (Policy Iteration on Program in Figure 3.6)**.** We analyze this program with a set of templates $\{i, k\}$, and we look for the least inductive invariant $\mathbf{d} \equiv (d_a^i, d_b^i, d_b^k)$ that subsumes the upper bound for the variables `i, k` for all possible program executions on nodes n_a and n_b respectively.

Thus we minimize (d_a^i, d_b^i, d_b^k) subject to the consecution condition, stating that the set of states represented by $\mathbf{d}$ is larger or equal to their strongest postcondition:

$$\min(d_a^i, d_b^i) \text{ s.t. } \begin{cases} d_a^i \geq \sup i'_a \text{ s. t. } (i'_a = 0) \\ \quad \vee (i_a \leq d_b^i \wedge i_a \neq 1000 \wedge i'_a = i_a + 1) \vee \bot \\ d_b^i \geq \sup i'_b \text{ s. t. } (i_b \leq d_a^i \wedge i'_b = i_b \wedge k'_b = 0) \\ \quad \vee (i_b \leq d_b^i \wedge k_b \leq d_b^k \wedge k_b \neq 1000 \wedge k'_b = k_b + 1 \wedge i'_b = i_b) \vee \bot \\ d_b^i \geq \sup k'_b \text{ s. t. } (i_b \leq d_a^i \wedge i'_b = i_b \wedge k'_b = 0) \\ \quad \vee (i_b \leq d_b^i \wedge k_b \leq d_b^k \wedge k_b \neq 1000 \wedge k'_b = k_b + 1 \wedge i'_b = i_b) \vee \bot \end{cases} \tag{3.26}$$

In the constraints above, disjunctions arise from multiple incoming edges to each node, and an extra argument $\bot$ is added, which represents the case where the associated node is not reachable (in that case, the bound on the template is $-\infty$).

Each *policy* for the program in Figure 3.6 is an under-approximation of Equation 3.26, where each disjunction is replaced by one contained disjunct.

For example, for the policy π

$$\begin{aligned} &d_a^i \geq \sup i'_a \text{ s. t. } (i'_a = 0) \\ &d_b^i \geq \sup i'_b \text{ s. t. } (i_b \leq d_a^i \wedge i'_b = i_b \wedge k'_b = 0) \\ &d_b^k \geq \sup k'_b \text{ s. t. } (i_b \leq d_a^i \wedge i'_b = i_b \wedge k'_b = 0) \end{aligned} \tag{3.27}$$

from Theorem 3.1 we know that the corresponding *value* (smallest (d_a^i, d_b^i, d_b^k)) can be found by solving a single linear programming query derived from Equation 3.27 by changing the lower bound to upper bound, maximizing for $(d_a^i + d_b^i + d_b^k)$, namespacing variables according to associated nodes and templates (Section 3.2.6), and dropping the inner supremum operator due to redundancy:

$$\max d_a^i + d_b^i + d_b^k \text{ s.t. } \bigwedge \begin{array}{l} d_a^i \leq i'_{i,a} \wedge i'_{i,a} = 0 \\ d_b^i \leq i'_{i,b} \wedge i_{i,b} \leq d_a^i \wedge i'_{i,b} = i_{i,b} \wedge k'_{i,b} = 0 \\ d_b^k \leq k'_{k,b} \wedge i_{k,b} \leq d_a^i \wedge i'_{k,b} = i_{k,b} \wedge k'_{k,b} = 0 \end{array} \tag{3.28}$$

yielding the expected value $(0, 0, 0)$.

In order to test the policy π from Equation 3.27 for the possibility of *improvement*, we compute its *local value*, by substituting the unknowns (d_a^i, d_b^i, d_b^k) on the right hand side of the global optimization problem in Equation 3.26 with the value obtained from π, and checking whether the system of constraints holds. In our example, the right hand sides evaluate to $(1, 0, 1)$ and as $0 \geq 1$ does not hold, π can be *improved* for the template i at n_a, and for the template k at n_b.

Thus we generate a new policy, obtained by replacing the right hand side of each constraint of Equation 3.26 with the disjunct which evaluates to the value causing the inequality. The

resulting new policy is:

$$\begin{aligned} d_a^i &\geq \sup i'_a \text{ s. t. } i_a \leq d_b^i \wedge i_a \neq 1000 \wedge i'_a = i_a + 1 \\ d_b^i &\geq \sup i'_b \text{ s. t. } (i_b \leq d_a^i \wedge i'_b = i_b \wedge k'_b = 0) \\ d_b^k &\geq \sup k'_b \text{ s. t. } i_b \leq d_b^i \wedge k_b \leq d_b^k \wedge k_b \neq 1000 \wedge k'_b = k_b + 1 \wedge i'_b = i_b \end{aligned} \tag{3.29}$$

Recall that the iteration process terminates when the least solution can not be improved any further.

We give the full trace of the policy iteration algorithm on the running example. As disjunctions in Equation 3.26 arise from multiple incoming edges per each node, a policy can be defined by a choice of an incoming edge per node per template, or $\bot$ if no such choice is feasible. We represent a policy symbolically as a 3-tuple of predecessor nodes (or $\bot$), as there are two nodes, with a single policy to be chosen for each node. The order corresponds to the order of the tuple of the unknowns. The initial policy $\mathbf{p}_0$ is $(\bot, \bot, \bot)$ with the corresponding value $\mathbf{v}_0 = (-\infty, -\infty, -\infty)$.

The trace on the example is:

1. Policy improvement: $\mathbf{p}_1 = (n_0, \bot)$, obtained with a local value $(0, -\infty, -\infty)$. The value determination yields the value corresponds to the initial condition: $\mathbf{d}_1 = (0, -\infty, -\infty)$.

2. Policy improvement find a feasible policy for n_b: $\mathbf{p}_2 = (n_0, n_a, n_a)$, with value determination yielding $\mathbf{d}_2 = (0, 0, 0)$. The value corresponds to the initialization condition for both nodes.

3. Policy improvement selects the looping edge for both n_a and n_b: $\mathbf{p}_3 = (n_b, n_a, n_b)$, resulting in a value $\mathbf{d}_3 = (1000, 1000, 1000)$.

4. Finally, the policy cannot be improved any further and the iteration converges with an invariant $i \leq 1000$ at n_a, and $k \leq 1000 \wedge i \leq 1000$ at n_b, which is strong enough to verify the asserts.

Each policy improvement requires at least three local SMT queries, and each value determination requires one global LP query.

3.3 Local Policy Iteration (LPI)

There are algorithm inefficiencies which can be seen even in the toy example: the policy should be improved only on templates where the new information is locally available (there is no point in re-computing the same bound multiple times), and value determination should only be computed once any of the relevant policies was improved, and only on the strongly connected component given by the variable dependencies associated with the improved policy. Furthermore, in the presence of multiple dependencies between the variables the performance of the policy iteration algorithm crucially depends on the iteration order: e.g. if we don't stabilize the inner cycle before propagating the information further, many recomputations might be required.

While keeping track of variable dependencies and identifying strongly connected components is possible even for the framework given by Algorithm 3.1 [Gau+07], forcing an algorithm which operates over a system of equations to follow an iteration order defined by the structure of a CFA is non-trivial. Moreover, combining policy iteration with other analyses and exchanging invariant candidates during runtime also can not be done in an obvious way.

Yet the classical Kleene worklist iteration algorithm (Algorithm 2.1) addresses all of these concerns: it allows making use of optimal iteration orders, keeping track of the updates propagating through the CFA, and combining the intermediate results between multiple analyses.

We develop a new policy-iteration-based algorithm, based on the principle of *locality*, which aims to address the scalability issues and the problem of communicating invariants with other analyses. We call it *local policy iteration* or LPI. To make it scalable, we consider the structure of a CFA being analyzed, and we aim to exploit its *sparsity*.

Our formulation is based on the following observation: policy iterations can be seen as standard Kleene iterations in the template constraints domain, where the abstract state apart from the bounds contains the meta-information which can be used to reconstruct the used policy. With this meta-information available, value determination can be defined as a widening operator, which converges to the *least* fixed point after finitely many iterations, and no narrowing steps are required.

3.3.1 LPI Formalization

In order to formalize LPI we define the lattice of abstract states $\mathcal{L}$, strongest postcondition operator $\mathcal{L} \to \mathcal{F}(\mathbf{x} \cup \mathbf{x}') \to \mathcal{L}$, and a join operator $\mathcal{L} \to \mathcal{L} \to \mathcal{L}$. We assume that the large block encoding pre-processing (Section 2.9) was performed on the input CFA, and that each transition relation is encoded as an existentially quantified formula $f \in \mathcal{F}(\mathbf{x} \cup \mathbf{x}')$ within a decidable theory.

An LPI abstract state is an element of a template constraints domain with meta-information added to record the *policy* used for generating the state.

Definition 3.1 (LPI Abstract State)**.** An LPI abstract state is either a bottom state $\bot$, or a mapping from the externally given set T of templates to tuples $(d, \mathit{policy}, \mathit{backpointer})$, where $d \in \mathbb{R}$ is a bound for the associated template t (the represented property is $t(\mathbf{x}) \leq d$), *policy* is a formula representing the policy that was used for deriving d (*policy* has to represent a concave function connecting primed and unprimed variables), and *backpointer* is another LPI abstract state that is a starting point for the *policy* (base case is a top state associated with a program entry, modelled by an empty mapping $\{\}$).

Note that the bound d is redundant, as it can be re-derived from policy and backpointer by solving the optimization query

$$\max t^\top \mathbf{x}' \text{ s.t. } \mathit{policy}(\mathbf{x} \cup \mathbf{x}') \wedge [\![\mathit{backpointer}]\!]^\sharp(\mathbf{x}) \tag{3.30}$$

We include the bound in the tuple as an optimization for performing quick coverage checks and postcondition computations.

Observe that we do not need to include positive or negative infinities in the abstract state, as a single mapping to $-\infty$ implies that the entire state is $\bot$ and therefore unreachable, and mapping any template to ∞ does not add any constraints, and thus can be discarded. The partial order over $\mathcal{L}$ is defined by component-wise comparison of bounds associated with respective templates (lack of a bound corresponds to an unbounded template). The concretization is given by the conjunction of represented template linear constraints, ignoring *policy* and *backpointer* meta-information. For example, an abstract state $\{x : (10, \_, \_)\}$ (underscores represent information irrelevant to the example) concretizes to $\{c \mid c[x] \leq 10\}$, and the initial state $\{\}$ concretizes to all concrete states.

Algorithm 3.2 LPI Postcondition Computation

1: **Input:** state a_0, transition relation $\tau(\mathbf{x} \cup \mathbf{x}')$, set of templates T
2: **Output:** new state
3: $new \leftarrow$ empty map
4: $\hat{\tau} \leftarrow \tau$ with disjunctions annotated using a set of marking variables M
5: ▷ Formula describing the state space at a destination location.
6: $\phi(\mathbf{x}') \leftarrow \exists \mathbf{x}.\, \gamma(a_0)(\mathbf{x}) \wedge \hat{\tau}(\mathbf{x} \cup \mathbf{x}')$
7: ▷ Perform abstraction on ϕ.
8: **for all** template $t \in T$ **do**
9: ▷ Maximize subject to the constraints introduced by the formula
10: ▷ and the starting state.
11: $d \leftarrow \max t^\top \mathbf{x}'$ s.t. $\phi(\mathbf{x}')$
12: $\mathcal{M} \leftarrow$ model at the optimum point
13: ▷ Replace marking variables M in $\hat{\tau}$ with their value from the model $\mathcal{M}$,
14: ▷ generating a concave relation representing the policy.
15: Policy $\psi \leftarrow \hat{\tau}[M/\mathcal{M}|_M]$
16: $new[t] \leftarrow (d, \psi, a_0)$
17: **end for**
18: **return** *new*

The *postcondition* computation (Algorithm 3.2) operates by maximizing all templates $t \in T$ subject to the constraints introduced by a_0 and the transition relation $\tau(\mathbf{x} \cup \mathbf{x}')$ representing a (combination of) CFA operators. Backpointer and a *policy* are produced from the SMT model $\mathcal{M}$, corresponding to the resulting from the maximization query. The selected policy is a concave under-approximation of τ, obtained by replacing all disjunctions ($\bigvee D$) with their arguments ($d \in D$), such that the chosen disjunct is modelled at the optimum ($\mathcal{M} \models d$). To do so, we annotate τ with *marking variables* (line 4): each disjunction $\tau_1 \vee \tau_2$ in τ is replaced by $(p \wedge \tau_1) \vee (\neg p \wedge \tau_2)$ where p is a fresh propositional variable. A policy associated to a bound is then identified by the values of the marking variables at the optimum (subject to the constraints introduced by τ and a_0), and is obtained by replacing the marking variables in τ with their values from $\mathcal{M}$ (line 15). Thus the LPI postcondition computation effectively performs the *policy-improvement* operation for the given node, as only the policies which are feasible with respect to the current candidate invariant (given by the previous abstract state) are selected.

Example 3.4 (Postcondition Computation as a Policy Improvement)**.** We start with a state a:

$$a = \Big\{x : (100, \top, \{\})\Big\}$$

which concretizes to $\{c \mid c[x] \leq 100\}$, and a set $T \equiv \{x\}$ of templates.

We wish to compute a postcondition after traversing the following fragment:

```
x = (x <= 10) ? x + 1 : 0;
```

This line generates a formula $\tau \equiv (x \leq 10 \wedge x' = x + 1 \vee x > 10 \wedge x' = 0)$. Firstly, we annotate τ with *marking variables*, which are used to identify the selected policy, obtaining $\hat{\tau} \equiv x \leq 10 \wedge x' = x + 1 \wedge m_1 \vee x > 10 \wedge x' = 0 \wedge \neg m_1$. Then we optimize $\hat{\tau}$, together with the constraints from the starting state a for the highest value of the template. This amounts to a

single maximization modulo SMT query:

$$\sup x' \text{ s.t. } x \leq 100 \wedge (x \leq 10 \wedge x' = x + 1 \wedge m_1 \vee x > 10 \wedge x' = 0 \wedge \neg m_1)$$

The query is satisfiable with a maximum of 11, and an SMT model:

$$\mathcal{M} \equiv \{x' : 11, m_1 : \top, x : 10\}$$

Replacing the marking variable m_1 in τ with its value in $\mathcal{M}$ gives us a disjunction-free formula $x \leq 10 \wedge x' = x + 1$, which we store as a *policy*. Finally, the newly created state is $\{x : (11, x \leq 10 \wedge x' = x + 1, a)\}$.

In LPI, we use the join operator (Algorithm 3.3) to optionally perform the *value determination* which computes the fixpoint value for the given policy. This can be seen as an *exact widening operator*, which converges after finitely many steps. Multiple iterations through the loop might be necessary to find the optimal policy and reach the global fixpoint. In the presence of nested loops, the process is repeated in a fixpoint manner: we "close" the inner loop, "close" the outer loop with the new information from the inner loop available, and repeat the process until convergence. Each iteration selects a new policy, thus the number of possible iterations is bounded.

The join operator first computes the component-wise maximum (line 12), choosing the new bound only if it is strictly larger. Then it computes a strongly connected component of variable dependencies for value determination (line 23) If a non-empty SCC is found, the value determination step (Algorithm 3.4) is launched. It computes the least fixpoint for the chosen policy across the entire strongly connected component where the current node n lies. From the map M, the algorithm generates a global optimization problem, where the set of fresh variables $d^t_{n_i}$ represents the maximal value a template t can obtain at the node n_i using the policies selected. Variable $d^t_{n_i}$ is made equal to the namespaced[3] *output* value of the policy $\psi(\mathbf{x} \cup \mathbf{x}')$ chosen for t at n_i (line 13). For each policy ψ and the associated backpointer a_0, we *constrain* the *input* variables of ψ using a set of variables $d^{t_0}_{n_0}$ representing bounds at the node n_0 associated with a_0 (line 16). This set of "input constraints" for value determination results in a quadratic number of constraints in terms of the number of selected policies. Finally, for each template t we maximize for d^t_n (line 23), which is the maximum possible value for t at node n under the current policy, and we record the bound in the generated state (line 24), keeping the old policy and backpointer.

The local-value-determination algorithm is almost identical to the max-strategy evaluation [GM12], except for two changes: we only add potentially relevant constraints from the "closed" loop (found by traversing backpointers associated with policies), and we maximize objectives one by one, not for their sum (which avoids special casing infinities, and enables optimizations outlined in Section 3.4). Unlike classic policy iteration, we only run local value determination after merges on loop heads, because in other cases the value obtained by abstraction is the same as the value which could be obtained by value determination.

3.3.2 Properties of LPI

Property 3.1 (Soundness)**.** LPI terminates with an inductive invariant.

[3]Namespacing means creating fresh copies by attaching a certain *prefix* to variable names.

Algorithm 3.3 LPI Join Operator

Input: node n, previous abstract state a_0, new abstract state a_1, set of templates T
Output: new joined state a'
$a' \leftarrow \{\}$
$updated \leftarrow \emptyset$
for all template $t \in T$ **do**
    if $t \notin a_0 \vee t \notin a_1$ **then**
        ▷ The value of t is unbounded
        continue
    end if
    (bound v_0, policy p_0, backpointer b_0) $\leftarrow a_0[t]$
    (bound v_1, policy p_1, backpointer b_1) $\leftarrow a_1[t]$
    if $v_1 > v_0$ **then**
        ▷ The new value is strictly larger
        ▷ we should update to the new policy.
        $a'[t] \leftarrow (v_1, p_1, b_1)$
        $updated \leftarrow updated \cup \{t\}$
    else
        ▷ Otherwise keep using the old policy.
        $a'[t] \leftarrow (v_0, p_0, b_0)$
    end if
end for
▷ Strongly connected component of variable dependencies
$scc \leftarrow$ strongly connected component in the graph defined by policy backpointers which contains a'.
if $scc \neq \emptyset$ **then**
    $M \leftarrow$ map from nodes to the corresponding states in scc
    $a' \leftarrow$ LocalValueDetermination(n, scc, T)
end if
return a'

Proof. LPI terminates when no more updates can be performed, and newly produced abstract states are subsumed (in the preorder defined by the lattice) by the already discovered ones. Thus, it is an inductive invariant: the produced abstract states satisfy the initial condition and all successor states are subsumed by the existing invariant. □

Property 3.2 (Termination)**.** LPI terminates on any input program.

Proof. An infinite sequence of produced states would have to repeat at least one node infinitely often. However, each subsequent abstraction on the same node must choose a different policy to obtain a successively higher value, but the number of policies is finite. An infinite sequence is thus impossible, hence a run of LPI is always guaranteed to terminate. □

Property 3.3 (Optimality)**.** In rationals, LPI terminates with the smallest inductive invariant expressible in the given domain.

Proof Outline. LPI can be seen as an efficient oracle for selecting the next policy to update (note that policies selected by LPI are always *feasible* with respect to the current invariant candidate). Skipping value-determination steps when they have no effect, and attempting to

Algorithm 3.4 Local Value Determination

Input: node n, map M from nodes to states, set T of templates
Output: generated state *new*
$constraints \leftarrow \emptyset$
for all node $n_i \in M$ **do**
  state $s \leftarrow M[n_i]$
  for all template $t \in s$ **do**
    (bound d, policy ψ, backpointer a_0) $\leftarrow s[t]$
    Generate a unique string *namespace*
    ▷ Prefix all variables in ψ.
    ▷ $\mathbf{x'}_{namespace}, \mathbf{x}_{namespace}$ is a set of namespaced output/input variables for ψ.
    $constraints \leftarrow constraints \cup \{\psi[\mathbf{x}/\mathbf{x}_{namespace}][\mathbf{x'}/\mathbf{x'}_{namespace}]\}$
    $d^t_{n_i} \leftarrow$ fresh variable (upper bound on t at n)
    $constraints \leftarrow constraints \cup \left\{d^t_{n_i} \leq t(\mathbf{x'}_{namespace})\right\}$
    $n_0 \leftarrow$ location associated with a_0
    for all $t_0 \in a_0$ **do**
      $constraints \leftarrow constraints \cup \left\{t_0(\mathbf{x}_{namespace}) \leq d^{t_0}_{n_0}\right\}$
    end for
  end for
end for
$new \leftarrow$ empty state
for all templates $t \in T$ **do**
  $(d_0, \psi, a_0) \leftarrow M[n]$
  $d \leftarrow \max d^t_n$ subject to *constraints*
  $new[t] \leftarrow (d, \psi, a_0)$
end for
return *new*

include only relevant constraints in the value-determination problem do not alter the values of obtained fixed points. □

Example 3.5 (LPI Trace on the Running Example)**.** We revisit the running example (Figure 3.6) with the local version of policy iteration:

1. We start with an empty state $a_0 \equiv \{\}$.

2. We compute the postcondition for the edge $(n_0, i' = 0, n_a)$, producing a new state a_1, defining the region $i \leq 0$:
$$\{i : (0, i' = 0, a_0)\}$$
Observe that the resulting policy for the template i is equal to the formula associated with an input edge, as it does not contain any disjunctions. This calculation requires solving one LP problem.

3. We explore the incoming edge to n_b, resulting in a new abstract state a_2:
$$a_2 \equiv \{i : (0, i' = i \wedge k' = 0, a_1), k : (0, i' = i \wedge k' = 0, a_1)\}$$
This calculation requires two LP queries.

4. Exploring the looping edge on n_b results in a new state

$$a_3 \equiv \{i : (0, k \neq 1000 \wedge k' = k + 1 \wedge i' = i, a_2), k : (1, k \neq 1000 \wedge k' = k + 1 \wedge i' = i, a_2)\}$$

again requiring two LP queries.

5. The join on node n_b merges a_2 and a_3, yielding:

$$a_4 \equiv \{i : (0, i' = i \wedge k' = 0, a_1), k : (1, k \neq 1000 \wedge k' = k + 1 \wedge i' = i, a_2)\}$$

Value determination "closes" the loop, producing a new state:

$$a_5 \equiv \{i : (0, i' = i \wedge k' = 0, a_1), k : (1000, k \neq 1000 \wedge k' = k + 1 \wedge i' = i, a_2)\}$$

which requires solving one LP problem.

6. Postcondition under the edge $(n_b, i \neq 1000 \wedge i' = i' + 1, n_a)$ generates the state

$$a_6 \equiv \{i : (1, i \neq 1000 \wedge i' = i + 1, a_5)\}$$

This is performed by solving a single LP problems.

7. Join of states a_6 and a_1 and the subsequent value determination yields the new state associated with n_a:

$$a_7 \equiv \{i : (1000, i \neq 1000 \wedge i' = i + 1, a_6)\}$$

8. Finally, the final postcondition computation under the edge $(n_a, k' = 0 \wedge i' = i, n_b)$ yields:

$$a_9 \equiv \{i : (1000, i' = i \wedge k' = 0, a_7), k : (1000, k \neq 1000 \wedge k' = k + 1 \wedge i' = i, a_2)\}$$

which subsumes a_5 and concludes the iteration.

Compared to the original algorithm there are two value-determination problems instead of three, both on considerably smaller scale. The improvement in performance is more than a fixed constant: if the suboptimal iteration order was picked for a larger problem, the increase might be exponential.

3.4 Extensions and Optimizations

3.4.1 Extending to Integers

Original publications on max-policy iteration in template constraints domain deal exclusively with reals, whereas C programs operate primarily on integers[4]. Excessively naive handling of integers leads to poor results: with an initial condition $x = 0$, $x \in [0, 4]$ is inductive for the transition system $x' = x + 1 \wedge x \neq 4$ in integers, but not in rationals, due to the possibility of the transition $x = 3.5$ to $x = 4.5$. An heuristical workaround is to rewrite each strict inequality $a < b$ into $a \leq b - 1$: on this example, the transition becomes $x = x + 1 \wedge (x \leq 3 \vee x \geq 5)$ and

[4]Previous work [GS07a] deals with finding the exact interval invariants for programs involving integers, but only for a very restricted program semantics.

```
1  int x, x_new;
2  x=0;
3  x_new=input();
4  while (2 * x_new == x+2) {
5      x = x_new;
6      x_new = input();
7  }
```

FIGURE 3.7: Integer Imprecision in Policy Iteration

$x \in [0, 4]$ becomes inductive on rationals. However, such a heuristic is not capable of capturing a *congruence* information, e.g. proving that a statement guarded by `2 * x == 2 * y + 1` is unreachable over integral variables, as an even number can not be equal to an odd one. Thus we form all our OPT-SMT queries in the logic sorts similar to the variable types: integers for machine integers, and rationals for floats.

Use of integers has an additional benefit: in the implementation we run a parallel congruence analysis (Section 7.7.1), and we inject the obtained invariants into the value determination and postcondition computation queries, making the resulting invariant more precise.

Linear relations over the integers are not convex or concave, which is a requirement for the least fixpoint property of policy iteration. Thus the encoding described above may still result in an over-approximation. E.g. consider the program shown in Figure 3.7. LPI terminates with a fixpoint $x \leq 2$, yet the least fixpoint is $x \leq 1$. We have not found the imprecision over the integers to be a large problem in practice: the resulting algorithm is still more precise than traditional abstract interpretation, and gives much better results than rational relaxation for all variables. Moreover, empirically we have found that *interesting* invariants about integer variables often involve congruence facts which are better obtained explicitly with a separate congruence analysis, e.g. $x + y = 1 \pmod 2$.

3.4.2 Extending to Uninterpreted Functions

In program analysis the theory of *uninterpreted functions* [KS08] (or UFs) is often used: e.g. for encoding pointer arithmetic operations, or complex non-linear operations which can not be modelled directly. Like linear integer arithmetic, the theory of uninterpreted functions is not convex and a direct LPI application might result in a suboptimal invariant. However, uninterpreted functions can be removed using *Ackermann reduction* [KS08], which instantiates a number of fresh variables, and encodes the function axioms explicitly. For example, the transition relation $\tau \equiv (p(x) = p(y) + 1 \wedge x = y)$ can be converted to $(p_x = p_y + 1 \wedge x = y \wedge ((x = y) \implies p_x = p_y)$. The Ackermann reduction pre-processing potentially results in a quadratic increase in a formula size, yet allows LPI to produce least inductive invariants over the theory involving uninterpreted functions.

3.4.3 Reducing the Number of Value Determination Constraints

In Section 3.3 we have described the local value-determination algorithm which adds a quadratic number of constraints in terms of policies. In practice this is often prohibitively expensive. The quadratic blow-up results from the "input" constraints to each policy, which determine the bounds on the input variables. We propose multiple optimization heuristics which increase the performance.

As a motivation example, consider a long trace ending with an assignment `x = 1`. If this trace is feasible and chosen as a policy for the template x, the output bound will be 1, regardless

of the input. With that example in mind, consider further the postcondition computation (Algorithm 3.2) from which we derive the bound d for the template t. Let $a_0, \tau(\mathbf{x} \cup \mathbf{x}')$ be the function input.

Syntactic Check We perform the slicing based on variable dependencies of $t(\mathbf{x}')$ in τ and in the formula generated by a_0, and we only follow the backpointers for the templates which can potentially affect the resulting value of the template. When computing the strongly connected component of relevant states for the value determination problem, we only follow the backpointers for the policies which were syntactically shown to be capable of affecting the final bound. E.g. for a transition $\tau \equiv x' = x + 1$ and a template $t \equiv x$, the resulting bound on x after performing value determination is independent of templates associated with a_0 which do not contain the variable x. Thus when performing value determination at this node, we do not follow the backpointers for such templates. Moreover, if from the syntactic analysis we know that none of the variables of t occur in τ, we can skip the optimization step altogether, and return the bound $a_0[t]$. For example, for a template $t \equiv x$ and a transition relation $\tau \equiv y \leq 5$, the resulting bound on t is not modified by τ and is given by $a_0[t]$.

Semantic Check Suppose the strongest postcondition computation has returned the bound d for the template t. We check the satisfiability of $\tau(\mathbf{x} \cup \mathbf{x}') \wedge t^\top \mathbf{x}' > d$; if the result is unsatisfiable, then the bound of t is *input-independent*, that is, it is always d if the trace is feasible. Thus we do not add the *input constraints* for the associated policy in the value-determination stage. Also, when computing the strongly connected component of relevant states for the value-determination problem, we do not follow the backpointers for input-independent policies, potentially drastically shrinking the resulting constraint set. For example, for $t \equiv x$ and the transition relation $\tau \equiv x' = 1$, the resulting bound is always 1, regardless of values associated with a_0. Thus when performing value determination we do not follow backpointers at all for the template t at such a node.

3.4.4 Merging the Unknowns

Furthermore, we limit the size of the value-determination LP by merging some of the unknowns. Namely, when multiple templates associated with the same state share the same policy, we do *not* create fresh namespaced copies for each of those templates, but share the same set of variables between them. This is equivalent to equating these variables, thus strengthening the constraints. The result thus under-approximates the fixed point of the selected policy. If it is less than the policy fixed point (not inductive with respect to the policy), we fall back to the normal value determination. An example of such an optimization is shown in Example 3.6: note that unlike classical value determination our procedure requires $\|T\|$ optimization queries instead of 1, yet they are often performed on a much simpler constraint set.

Example 3.6 (Merging Unknowns for Value Determination)**.** We revisit the example in Figure 3.5 with the template set $T \equiv \{x, y\}$. We are running value determination for the policy given by selecting the looping transition τ_l for both templates. The construction of the value determination problem given in Section 3.2.6 requires us to solve the following optimization

problem:

$$\max(d_1, d_2) \text{ s.t. } d_1 \leq x_1' \wedge d_2 \leq y_2' \wedge x_1 \leq d_1 \wedge y_1 \leq d_2 \wedge x_1' = x_1 + 1 \wedge y_1' = y_1 + 1 \wedge x_1 < 10 \\ \wedge x_2 \leq d_1 \wedge y_2 \leq d_2 \wedge x_2' = x_2 + 1 \wedge y_2' = y_2 + 1 \wedge x_2 < 10 \tag{3.31}$$

The Equation 3.31 contains the constraint set resulting from the looping transition twice with two different namespaces. Instead, we optimize for a simpler constraint set by merging the variables associated with templates "sharing" the looping policy, and we get the following constraint set:

$$C \equiv d_1 \leq x' \wedge d_2 \leq y' \wedge x \leq d_1 \wedge y \leq d_2 \wedge x' = x + 1 \wedge y' = y + 1 \wedge x < 10 \tag{3.32}$$

In order to recover the bounds, we optimize for d_1 and d_2 in separate optimization queries with respect to the constraints in C. In our example (and as we have seen empirically, in many others), the resulting value of the objective function remains the same.

The resulting constraint set is, in general, stronger than the one originally associated with the value determination problem, and may even be unsatisfiable. Thus we switch to the more expensive procedure if the obtained invariant candidate is not inductive with respect to the chosen policies, preserving soundness.

3.4.5 Shrinking the Search Space

Additionally, during maximization we add a redundant lemma to the set of constraints that specifies that the resultant value has to be strictly larger than the current bound. This significantly speeds up the maximization by shrinking the search space.

Unfortunately, this approach does not combine well with the convex-hull based template synthesis, described in Section 4.5, as the redundant lower bounds might be required for the correct computation of the convex hull.

3.4.6 Ordering the Optimization Objectives

Consider emulating the octagon domain and synthesizing a set of templates $\pm x \pm y$ for all variables $x, y \in \mathbf{x}$. For most programs this set will be redundant: for instance, for describing the "cube" $\forall v \in \{x, y, z\} \,.\, 0 \leq v \leq 1$ only six templates are required: $x, -x, y, -y, z, -z$. Yet the octagons abstract domain would generate 36 templates instead, most of them *redundant*. We call an optimization objective redundant if it's value can be derived from the already computed objectives. In our example, the bound on e.g. $x + y$ is simply the sum of the bound on x and the bound on y. Fortunately, an optimization solver based on simplex can exploit this redundancy, as in our example the underlying simplex *tableau* will not require any further pivots for computing the bound on $x + y$ after the bounds on x and y were computed.

Thus we apply a length-based ordering to optimization objectives in the abstraction step (optimizing for objectives with fewer variables first).

	vs. PAGAI	LPI	BLAST	CPAchecker	Unique	**Verified**	**Incorrect**
PAGAI		4	13	15	1	52	1
LPI	13		20	20	7	61	1
BLAST	6	4		8	0	45	1
CPAchecker	21	17	21		12	58	2

TABLE 3.1: Number of verified programs for different tools. The first five columns represent *differences* between approaches: the cell corresponding to the row A and a column B (read "A vs. B") displays the number of programs A could verify and B could not. In the column *Unique* we show the number of programs only the given tool could verify (out of the analyzers included in the comparison). The column *Verified* shows the total number of programs a tool could verify. The column *Incorrect* shows false positives: programs that contained a bug, yet were deemed correct by the tool.

3.5 Experiments

We have evaluated our tool on the benchmarks from the category "Loops" of the International Competition on Software Verification (SV-COMP'15) [Bey15] consisting of 142 C programs, 93 of which are correct (the error property is unreachable). We have chosen this category for evaluation because its programs contain numerical assertions about variables modified in loops, whereas other categories of SV-COMP mostly involve variables with a small finite set of possible values that can be enumerated effectively. All experiments were performed with the following resource bounds: an Intel Core i7-4770 quad-core CPU with 3.40 GHz, and limits of 5 GB RAM, 100 s CPU time, and 4 cores per program. Our implementation is described in detail in Chapter 7, along with installation and usage instructions.

We compare LPI (with templates synthesis algorithms described in Chapter 4) with four tools representing different approaches to program analysis:

- BLAST [SMM12] running lazy abstraction with interpolants. The version used is v2.7.3.
- PAGAI [HMM12], git hash `254c2fc693`, running abstract interpretation with path focusing.
- CPACHECKER [BK11], version v.1.3.10-svcomp15, the winner of SV-COMP 2015 category "Overall", which uses an ensemble of different techniques: explicit value, k-induction, and lazy predicate abstraction.

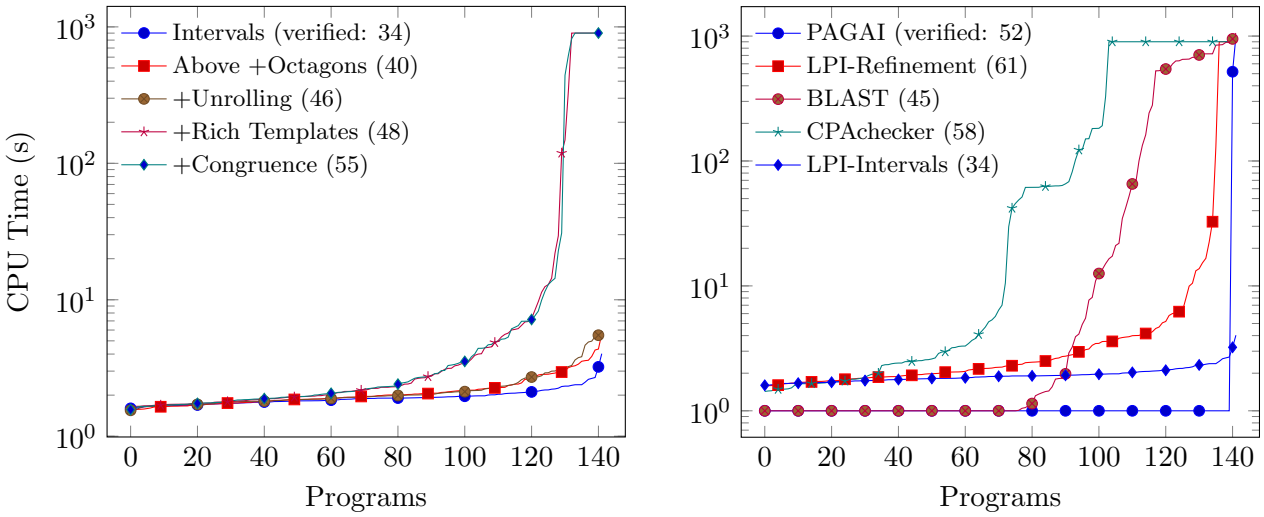
For LPI we use the CPACHECKER version `1.4.10-lpi-vmcai16`.

Because LPI is an incomplete approach, it can only produce safety proofs (no counter-examples). Thus in Table 3.1 we present the statistics on the number of safety proofs produced by different tools, with LPI running in abstraction refinement mode. From it we see that LPI verifies more examples than other tools can, including seven programs that no other tool could.

3.5.1 Timing Results

In Section 3.4 we have described the various possible configurations of LPI. As trying all possible combinations of features is exponential, tested configurations represent cumulative stacking of features. We present the timing comparison across those in the quantile plot in Figure 3.8a, and in the legend we report the number of programs each configuration could verify.

The quantile plot for timing comparison across different tools is shown in Figure 3.8b. We have included two LPI configurations in the comparison: fastest (LPI-Intervals) and the



(A) Different LPI Configurations (B) Different Tools

FIGURE 3.8: *Quantile* timing plots demonstrating the performance of different program analysis approach. Each data point corresponds to a processed verification task, with y coordinate given by the time taken to analyze the task, and x coordinate given by the program number (each series is sorted by time *separately* for each tool). Intuitively, the lower the line the faster the tool is. For readability there are less markers than programs, and all runtimes less than one second have been rounded up.

most precise one (LPI-Refinement, switches to a more expensive strategy out of the ones in Figure 3.8a if the program cannot be verified). From the plot we can see that LPI performance compares favorably with lazy abstraction, but that it is considerably outperformed by abstract interpretation. The initial difference in the analysis time between the CPACHECKER-based tools and others is due to JVM start-up time of about 2 seconds.

3.6 Conclusion

In this chapter we have demonstrated that LPI is a valuable method for code analysis, which can compete with the existing state-of-the-art techniques.

3.6.1 Future Work

Extending to Min-Policy Iteration The findings present in this chapter can not be directly applied to min-policy [Cos+05], as the CPA algorithm terminates whenever the candidate invariant is inductive, and thus stating the descending iterations is problematic (as even the starting state satisfies the coverage condition).

Supporting Non-Linear Templates A certain class of non-linear templates can be handled within the max-policy iteration framework by using *semi-definite programming* [VB96] during the value determination step. However, as value determination problems can be very large, max-policy approach with non-linear templates has troubles scaling to large programs [RG14].

Handling Non-Concave Operators Non-linear operations can be handled in usual ways, by e.g. using intervalization to replace them with a sound over-approximation.

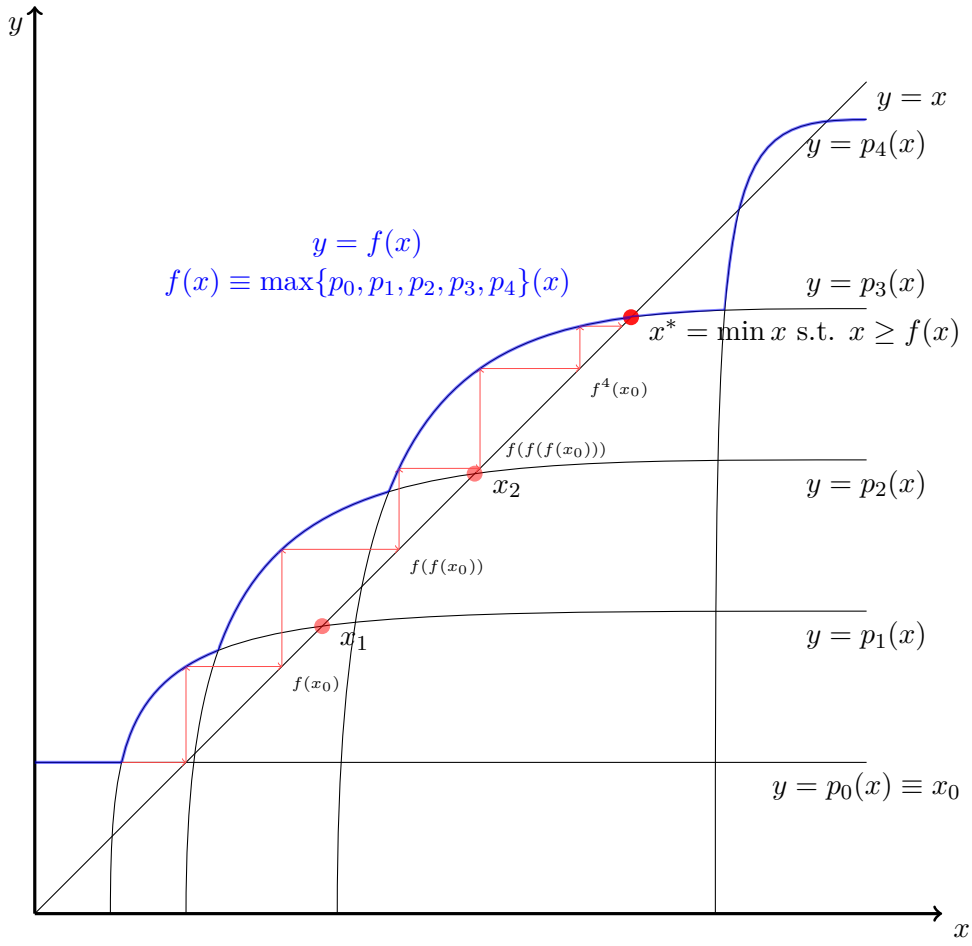


FIGURE 3.9: For comparison with policy iteration shown in Figure 3.4 we visualize value iterations (iteratively applying f until convergence) using red arrows. Observe that in our example value iteration does not converge in finite time and widening might be required.

Chapter 4

Template Synthesis

4.1 Introduction

As before, we are working on a task of synthesizing an inductive invariant I which entails unreachability of an error property E for a program modelled by a CFA P. Such an inductive invariant is called *separating*. In order to generate these invariants, we use abstract interpretation in a template constraints domain (TCD) initially presented in Section 2.8.3, as it offers a parameterizable compromise between precision and performance, and can be used in conjunction with policy iteration (Chapter 3).

An analysis in a TCD requires a *template annotation* for an input program. Each control location to which an inductive invariant is associated has to be annotated with a set of templates T, which defines the *expressible* invariants during the analysis. E.g. for $T \equiv \{x+y, y\}$ expressible inductive invariants are of the shape $x + y \leq d_1 \wedge y \leq d_2$ for $d_1, d_2 \in \bar{\mathbb{R}}$ (recall that $\bar{\mathbb{R}} \equiv \mathbb{R} \cup \{+\infty, -\infty\}$, which is used to denote unreachable states and unbounded templates). The set T is a parameter defining the precision to performance ratio: if T is too large, the analysis may become unfeasible, yet if T does not contain templates required for stating a separating inductive invariant, the property could not be proven even if it does hold.

In this chapter we present different techniques for generating a set of templates for a program, along with the evaluation and comparison. Each such technique is described in a separate, self-contained section. This work is performed in the context of using the resulting templates for the local policy iteration developed in Chapter 3, but the presented algorithms can be generalized to the problem of template synthesis in general.

Chapter Outline We develop an enumerative template synthesis algorithm and we study its properties in Section 4.2. In order to reduce the number of synthesized templates, in Section 4.3 we present an algorithm for filtering the templates based on the *liveness* data obtained from a separate dataflow analysis, and we describe its effect on the completeness property. In Section 4.4 we go further by using *interpolants* to synthesize the templates relevant to the property being proven. The last two algorithms we present use polyhedral analysis for template synthesis, and are given in Section 4.5. Unlike the previous approaches, the algorithm in Section 4.5.3 adds new templates on the fly (while other approaches require analysis restarts in CEGAR fashion). Finally, we present the evaluation comparing all of the approaches in Section 4.6, and we conclude in Section 4.7.

4.1.1 Related Work

The problem of choosing an abstract domain which is expressive enough to prove the desired property, yet efficient enough to be scalable is addressed in a influential paper [Cla+00]

on counterexample-guided abstraction refinement (CEGAR) by Clarke et al. Most of the algorithms we present are based on this idea. The approaches we present in sections 4.2 and 4.5.2 both perform abstraction refinement, yet not guided by the counterexample, while the interpolation-based approach (Section 4.4) does perform classical CEGAR refinement.

The octagons abstract domain described in Section 2.8.1 can be seen as an instance of a template constraints domain, as it tracks upper bounds on equalities $\pm x \pm y$ for all program variables $x, y \in \mathbf{x} \times \mathbf{x}$. In order to avoid having a quadratic number of constraints at each abstract state, the original publication [Min06] proposes variable *packing*, where according to a heuristical syntactic criteria variables are grouped into multiple sets, and octagonal constraints are tracked separately for each group.

The approach [Oh+14] of Oh et al. takes the packing further: they propose running a pre-analysis on a small, finite domain, from the result of which they extract the grouping of variables. Neither of the approaches is proven to have the same expressive power as the full octagonal analysis with no reductions.

Gawlitza et al. propose *parametric* policy iteration [SGS14] which finds the least solution for the set of semantical equations for all possible values of the parameters using the *region tree* datastructure. Their problem is more general, as they allow parameters to occur in the analyzed program, and the output of the analysis can be a non-convex inductive invariant.

4.2 Enumerative Template Synthesis

Observe that performing an analysis in the intervals domain can be emulated in TCD by synthesizing templates $\pm x$ at every program location n for every program variable $x \in \mathbf{x}$. An octagon domain [Min06] can be emulated in a similar way, enlarging the synthesized template space to $\pm x \pm y$. In this section we extend this enumerative synthesis method for synthesizing arbitrary templates with coefficients over $\mathbb{Z}$ in a way inspired by syntax-guided synthesis [Alu+13].

For a template t let $\|t\|$ denote the template *size*, which is a number of variables occurring in a template (e.g. 3 for $x + y + z$). Observe that for a fixed set of program variables $\mathbf{x}$, a template size is always less than $\|\mathbf{x}\|$, as there is at most one coefficient associated to each variable. Using an integer $n \in \mathbb{N}$ defining the magnitude of the largest allowed coefficient, we generate a set of templates based on linear expressions of size $\leq \|\mathbf{x}\|$, where all coefficients are in the set $\{c \mid c \in \mathbb{Z} \wedge |c| \leq n\}$. For a finite set of program variables $\mathbf{x}$ the resulting set T is finite. Observe that by continuously increasing n we can eventually synthesize all templates required for expressing any linear polyhedra with integral coefficients. Furthermore, this approach effectively generates all templates with rational coefficients, as a template over rationals could be represented in integers by multiplying all coefficients by a greatest common denominator (e.g. a template $x/3 + y/2$ has equivalent expressive power to $2x + 3y$).

Refinement synthesis procedure parameterized by an abstract interpretation in a template constraints domain forms a (semi) algorithm for safety checking of the error property, which we state in Algorithm 4.1. For a given set of templates we can find the least inductive invariant using policy iteration (line 8), and then if it does not provide separation from the error property we can perform refinement (line 12), extending the set of allowed templates. Note that algorithm 4.1 does not guarantee termination: in fact, it never terminates on programs where the error property is reachable and no separating inductive invariant exists in the domain of convex polyhedra with rational coefficients. In practice, we enforce termination by having a threshold on the largest possible value of n, and terminating with an `UNKNOWN` verdict once this threshold

Algorithm 4.1 Semialgorithm for Finding Separating Inductive Invariant

```
1: Input: CFA P ≡ (nodes, edges, n_0, x), error state E
2: Output: separating inductive invariant I : nodes → F(x)
3: n ← 1
4: while true do
5:     T ← set of all templates with integral coefficients of absolute size less than n
6:     ▷ Generate an inductive invariant for P
7:     ▷ running abstract interpretation in the TCD T.
8:     I ← ANALYZE(P, T)
9:     if I entails unreachability of E then
10:        break
11:    end if
12:    n ← n + 1
13: end while
14: return I
```

is reached.

4.2.1 Beyond Rationals

Background We rely on basic number classification: a real number $n \in \mathbb{R}$ is called *rational* ($n \in \mathbb{Q}$) iff it can be written as a fraction p/q where p is a positive integer, and q is a non-zero integer. A number n is called *algebraic* iff it is a root of a polynomial in one variable with rational coefficients. Algebraic numbers can be programmatically defined using the polynomial for which they represent roots, and manipulated programmatically. Non-algebraic real numbers (e.g. π) are called *transcendental*.

Recall that policy iteration, described in Chapter 3, guarantees to find the least inductive invariant in a given template constraints domain. This begs the question of whether the parameterization of Algorithm 4.1 with policy iteration gives a semidecidable approach for finding a separating linear inductive invariant, as for a given set of templates such a parameterization always finds the least inductive invariant. In turn, this depends on whether Algorithm 4.1 guarantees to eventually synthesize all templates required for expressing an arbitrary separating inductive invariant. In this section we show that the answer is "no", as there exist programs where all the coefficients are integral and all operations are linear, yet every non-trivial linear inductive invariant has irrational coefficients. Nevertheless, Algorithm 4.1 allows us to derive useful safety conditions for many programs.

Theorem 4.1 (Templates are Expressible Using Algebraic Numbers)**.** If there is a linear inductive invariant I expressed using real coefficients for a CFA P where all transitions are linear and involve only rational numbers, then there exists an invariant I' for P such that $I' \implies I$ and I' is expressible in algebraic numbers.

Proof. If I is linear, then it is expressible as a conjunction of linear inequalities over a set of templates T. For a fixed number of templates (given by $\|T\|$) and a CFA P all inductive invariants have to satisfy the first-order arithmetic constraints generated from P [CSS03]. Such a system of constraints is satisfiable over reals, and is consequently satisfiable over any real closed field, such as algebraic reals. As the obtained invariant I' is the least possible one expressible in $\|T\|$ templates, it has to imply the original invariant I. □

As algebraic numbers are countable (cf. [Niv56], Theorem 7.5), the refinement approach (Algorithm 4.1) can be extended to synthesizing *all* possible templates, and consequently a semidecidable algorithm for proving safety, yet such an enumeration would be unfeasible in practice.

```
1  int x = 1;
2  int y = 0;
3  while (input()) {
4      x = -x + y;
5      y = x + y;
6  }
```

FIGURE 4.1: Program Requiring Irrational Coefficients for Expressing Inductive Invariant

Example 4.1 (Irrational Coefficients in Templates)**.** Consider a program P shown in Figure 4.1. The only non-trivial linear inductive invariant it admits is:

$$y \geq (1 - \sqrt{2})x \wedge y \leq (1 + \sqrt{2})x$$

This result is counter-intuitive, as for many programs inductive invariants of interest are trivial and may be even already syntactically present in form of guards.

We now explain from where the inductive invariants from Example 4.1 come from, and why they are unique. Consider a program P consisting of a single loop performing an update $\mathbf{x} = A\mathbf{x}'$, where A is a square matrix of dimension $\|\mathbf{x}\|$. The evolution of the values of program variables of P corresponds to the dynamics of the difference equation, and is determined by the eigenvalues and eigenvectors of the matrix A [Ela96]. The phase portrait of the difference equation corresponding to the program discussed in Example 4.1 is shown in Figure 4.2. It is easy to see that for a two-by-two matrix A with eigenvalues of different signs the only possible linear convex inductive invariants are given by the eigenvectors of A. As the eigenvalues can be calculated by solving the characteristic polynomial $\det[A - \lambda I] = 0$, which requires finding the roots of the polynomial of degree of the size of A, irrational numbers may appear in eigenvalues and consequently in eigenvectors.

4.3 Filtering Templates Using Live-Variables Analysis

The template generation scheme in Algorithm 4.1 generates a very large number of templates. However, for real programs most of those templates are redundant, or irrelevant for proving the target property. Intuitively, the information about the variables which are not going to be used again should not be relevant for the safety invariant generation. Thus we use the data obtained from the *live variables* analysis to reduce the number of considered templates. Recall that the variable x is alive at a CFA node n, iff there exists an execution proceeding from n which depends on the value of x (either directly through read, or indirectly using pointers).

Unfortunately, not considering templates containing dead variables can affect the precision when *relational* templates are used: that is, templates with more than one variable, where the bound on one template may influence the other. For example, consider the program in Figure 4.3. Suppose the program is analyzed using a set of templates T consisting of all possible templates of size up to three, containing only constants $\{0, 1, -1\}$. Using such a set of templates, the assertion violation in the considered program can be proven to be unreachable using the invariant $a = b + c$, which requires *supporting* templates $\pm(a - b - c)$. Observe that if those

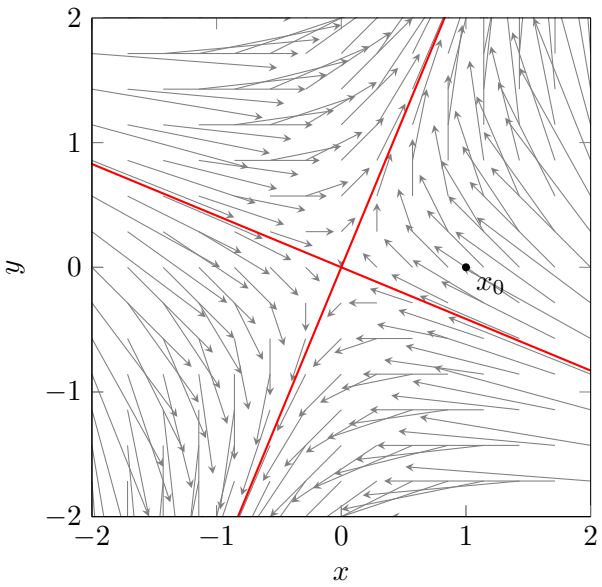


FIGURE 4.2: Phase Portrait showing evolution of variable values of program in Figure 4.1. Red lines show the directions of eigenvectors, which define the linear inductive invariants with irrational coefficients. Grey vectors show the evolution step: the next value of a point at the arrow origin corresponds to one application of the loop iteration. Black point shows the initial state.

templates are removed from T, a separating inductive invariant can no longer be found, even though the variable a is not live at the loop entry.

```
1  int b = input();
2  int c = input();
3  int a = b + c;
4  int d = 2 * a;
5  while (input()) {
6      b++;
7      c--;
8      assert(d * d = 4 * (b + c) * (b + c));
9  }
```

FIGURE 4.3: Example of Relevance of Dead Variables for Template Generation

The loss of the precision may be avoided by performing the *projection* operation on the dead variables first. As any TCD is a subset of the polyhedra domain, any variables can be projected away without the loss of precision with respect to other variables. Thus our example in Figure 4.3 can be proven using templates $\pm(d - 2 * (b + c))$, which are given by projecting the dead variable a out.

Furthermore, in two important cases the projection operation is not necessary. In the intervals domain, the template set is not relational, and all dead templates can be discarded. In the octagons domain, as the projection of an octagon is always an octagon [Min06], performing the projection is not necessary if all the supporting templates for the octagon are already specified.

In short, liveness filtering provides a large increase in performance (cf. Section 4.6.1) without any precision penalty in case of octagonal and interval templates, or at the cost of a projection operation.

4.4 Interpolation-Based Template Synthesis

Consider proving the unreachability of an error property E using an inductive invariant. In this context, we are not interested in finding the smallest inductive invariant, but merely one which is strong enough to show the unreachability of E. We are additionally interested in finding such inductive invariants for programs so large, that even emulating the intervals domain (using templates $\pm x$ for every program variable x) becomes too expensive, especially in the context of running policy iteration (Chapter 3). Intuitively, only a fraction of the program variables are relevant to the property, and in this section we aim to find templates *relevant to the unreachability proof of* E.

Background Many approaches exist for using *interpolants* (Definition 2.13) in order to perform abstract domain refinement, both in the context of predicate abstraction [McM06] and explicit value analysis [BL13]. Recall that for two formulas ϕ and ψ such that the conjunction $\phi \wedge \psi$ is unsatisfiable an interpolant I is a formula over the shared variables of ϕ and ψ such that $\phi \implies I$ and $I \implies \neg\psi$ both hold. This definition is extended to *sequences* [McM06]: for a sequence of formulas $S \equiv (s_1, \ldots, s_n)$ where $\bigwedge S$ is unsatisfiable, a sequence $I \equiv (i_1, \ldots, i_n)$ is an interpolant for S if and only iff $i_1 = \top$, $i_n = \bot$, for all $i \in [1, n]$ $i_{i-1} \wedge s_i \implies i_i$ and additionally, i_i only contains the shared symbols of $\bigwedge_{i\in[1,i]} s_i$ and $\bigwedge_{i\in[i+1,n]} s_i$. If all formulas in S are quantifier free, such a sequence I exists for many theories, including linear rational arithmetic [McM05].

We present an algorithm for generating new templates from interpolants. Two difficulties arise when applying an interpolation-based refinement to an abstract interpretation running over a template constraints domain. Firstly, template constraints domain is restricted to the linear expressions over program variables, and can not be refined directly using arbitrary predicates contained in an interpolant. Secondly, the output of an abstract interpretation in case the separating inductive invariant could not be found does not contain a path from the program start to the property violation. The second limitation extends to most analyses based on abstract interpretation.

We present two simple solutions to these problems. In Section 4.4.1 we provide an algorithm for dynamically generating an abstract reachability tree from a given abstract interpretation analysis. This allows us to generate interpolants from analysis runs, as discussed in Section 4.4.2. In Section 4.4.3 we describe and evaluate a method for *guiding* the interpolation procedure towards less overfitting interpolants by weakening the formulas given to the solver.

4.4.1 Abstract Reachability Tree Generation

We describe an algorithm for generating an abstract reachability tree [Bey+07] from an analysis formulated as an abstract interpretation. Such a generation can not be done in an obvious way, as abstract interpretation relies on the presence of joins between multiple states for convergence, and an ART-generating analysis can not perform joins (as otherwise the resulting graph might contain cycles). Instead, we *emulate* the joins in the postcondition operator: if during the postcondition calculation resulting in a state s_1 we can find a state s_0 on the same tree "branch" associated with the same CFA node (we call such a state a "neighbour"), instead of returning s_1 we return the joined state $s_0 \sqcup s_1$. Such a construction allows us to guarantee termination, while generating abstract counterexample traces.

Definition 4.1 (Abstract Reachability Tree)**.** An ART for a CFA $(nodes, edges, n_0, \mathbf{x})$ is a set of tree nodes N. Each tree node $n \in N$ is a triple, consisting of a CFA node $p \in nodes$, defining which location n corresponds to, an abstract domain element $d \in \mathcal{D}$, defining the reachable state space at n, and a pointer to the parent node of $b \in (N \cup \{\emptyset\})$ ($\emptyset$ for the tree root), defining the tree structure. The starting tree node is $(n_0, \top, \emptyset)$.

An ART N is *sound* iff the output of each transition over-approximates the strongest postcondition: that is, for each tree node $n \equiv (a, d, b) \in N$ where $b = (a_0, d_0, b_0) \neq \emptyset$, there exists an edge $(a_0, \texttt{OP}, a) \in edges$, and the abstract state d associated with n over-approximates the strongest post-condition of b under $\texttt{OP}$: $[\![\texttt{OP}]\!]^\sharp(d_0) \preceq d$. A node $n \equiv (a, d_0, b_0) \in N$ is *fully expanded* iff for all edges $(a, \texttt{OP}, c) \subseteq edges$ where $[\![\texttt{OP}]\!]^\sharp(d_0) \neq \bot$ there exists a node $(c, d, n) \in N$, where $[\![\texttt{OP}]\!]^\sharp(d_0) \preceq d$. A node $(a, d_1, b_1) \in N$ *covers* another node $(a, d_2, b_2) \in N$ iff $d_2 \preceq d_1$. A sound ART where all nodes are either fully expanded or covered represents an inductive invariant.

Tree Generation Algorithm We generate a tree for any analysis defined by an abstract domain $\mathcal{D}$ equipped with a partial order $\preceq$, strongest postcondition $[\![\texttt{OP}]\!]^\sharp : \mathcal{D} \to \mathcal{D}$ parameterized by an operator $\texttt{OP} \in \texttt{OPS}$, join function $\sqcup_{\mathcal{D}} : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$, and an initial state $d_0 \in \mathcal{D}$.

An algorithm listing for generating an abstract reachability tree given such a parametrization is shown in Algorithm 4.2. We maintain three stateful datastructures: a set of all tree nodes N, a set of expanded nodes E and a set of covered nodes C (line 5). Then for all tree nodes which are not expanded and not covered (line 8) we calculate the postcondition using the parameterized analysis, and then check the coverage with respect to existing nodes (line 13).

To generate the postcondition for an element $n \equiv (n_a, d, b)$ under a given CFA edge, we first generate an abstract state d' using the abstract interpretation postcondition operator (line 28). Then we traverse a chain of "parent" pointers upwards from n (line 36), until we either hit the tree root, or a "neighbour" element $s \equiv (n_a, d_0, b_0)$ (two tree noes are called neighbours if they share a CFA node). If such an s exists, we return the result of the join application to d_0 and d' (line 32). Otherwise, we simply return the element with an abstract state given by d'.

Property 4.1 (Termination)**.** Algorithm 4.2 is guaranteed to terminate whenever $\mathcal{D}$ has finite height (and termination can be enforced by using widenings during the join application otherwise).

Proof. For any ART branch B, for any sequence S of ART states associated to the same CFA node, all elements of S apart from the first one were created using the join function application. As any infinite sequence of states would have to repeat a CFA node infinitely often, and repeatedly applying joins in a finite-height domain ensures convergence, this forces the tree height (size of the largest branch) to be finite. The tree width is also finite for a finite CFA, as at each tree level we only create as many neighbours as there are outgoing CFA edges for a processed node. As every iteration of Algorithm 4.2 creates a new ART node, the finiteness of the resulting tree guarantees the algorithm termination. □

Example 4.2 (Abstract Reachability Tree Generation Example)**.** We show the run of the Algorithm 4.2, for the input program in Figure 4.4, parametrized with an analysis running local policy iteration (LPI, Chapter 3) in the template constraints domain $T \equiv \{\pm x, \pm y, \pm(x - y)\}$ (recall that negating the template gives us the lower bound). The resulting reachability tree describes the progress of the analysis computation and is shown in Figure 4.5. In order to

Algorithm 4.2 Abstract Reachability Tree Generation

Input: CFA $(nodes, edges, n_0, \mathbf{x})$, postcondition operator $[\![.]\!]^\sharp : \texttt{OPS} \to \mathcal{D} \to \mathcal{D}$,
  join operator $\sqcup_{\mathcal{D}} : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$, initial state $d_0 \in \mathcal{D}$
$N \leftarrow \{(n_0, d_0, \emptyset)\}$
$E \leftarrow \emptyset$
$C \leftarrow \emptyset$
while $\exists n \equiv$ (node a, abstract state d, backpointer b) $\in (N \setminus E \setminus C)$ **do**
    ▷ Expand all outgoing edges from n.
    for all edge $e \equiv$ (node a, operator OP, node c) $\in$ *edges* **do**
        $n' \equiv (c, d', n) \leftarrow$ Post$(a, \texttt{OP}, n_b, d, n)$
        if $d' \neq \bot$ **then**
            $N \leftarrow N \cup \{n'\}$
            ▷ Check Coverage.
            for all $n_0 \equiv$ (node a, abstract state d_0, backpointer b_0) $\in (N \setminus C)$ **do**
                if $d_0 \preceq d'$ **then**
                    ▷ Newly created tree node covers n_0.
                    $C \leftarrow C \cup \{n_0\}$
                end if
                if $d' \preceq d_0$ **then**
                    ▷ Newly created tree node is covered by n_0.
                    $C \leftarrow C \cup \{n'\}$
                end if
            end for
        end if
    end for
    $E \leftarrow E \cup \{n\}$
end while
function Post(node n_a, operator OP, node n_b, abstract state d, ART node n)
    $d' \leftarrow [\![\texttt{OP}]\!]^\sharp(d)$
    neighbour $s \equiv (n_a, d_0, b_0) \leftarrow$ FindNeighbour(b, n_a)
    if $s \neq \emptyset$ **then**
        ▷ If a neighbour was found, set output to the merge result.
        $d' \leftarrow d' \sqcup_{\mathcal{D}} d_0$
    end if
    return (n_b, d', n)
end function
function FindNeighbour(state $b \equiv$ (node n_0, abstract state d, backpointer b_0), node n_a)
    if $n_0 = n_a$ **then**
        return b
    else if $b_0 = \emptyset$ **then**
        return $\emptyset$
    else
        return FindNeighbour(b_0, n_a)
    end if
end function

```
1  int x = 0;
2  int y = 0;
3  while (x < 10 && y < 10) {
4      x++;
5      y++;
6  }
7  assert(x <= 10);
```

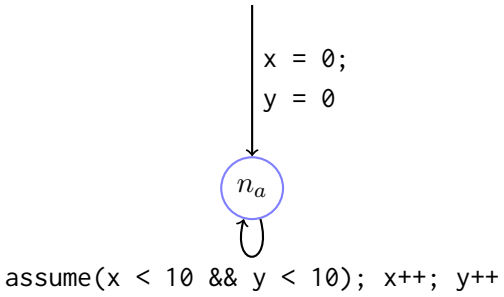


FIGURE 4.4: Two Counter Program Example

save space, inside each ART node we only display the corresponding LPI invariant, as all the generated nodes correspond to the CFA node n_a. After the first transition we generate the LPI state $x = 0 \wedge y = 0$, given by the loop precondition. The second ART node is given by taking the looping transition, and performing the join with the first ART node, resulting in $x \in [0, 1] \wedge y \in [0, 1] \wedge x = y$. After taking the third transition, LPI runs value determination on join, resulting in the state $x \in [0, 10] \wedge y \in [0, 10] \wedge x = y$ which is finally inductive, as indicated by the coverage relation with respect to its successor.

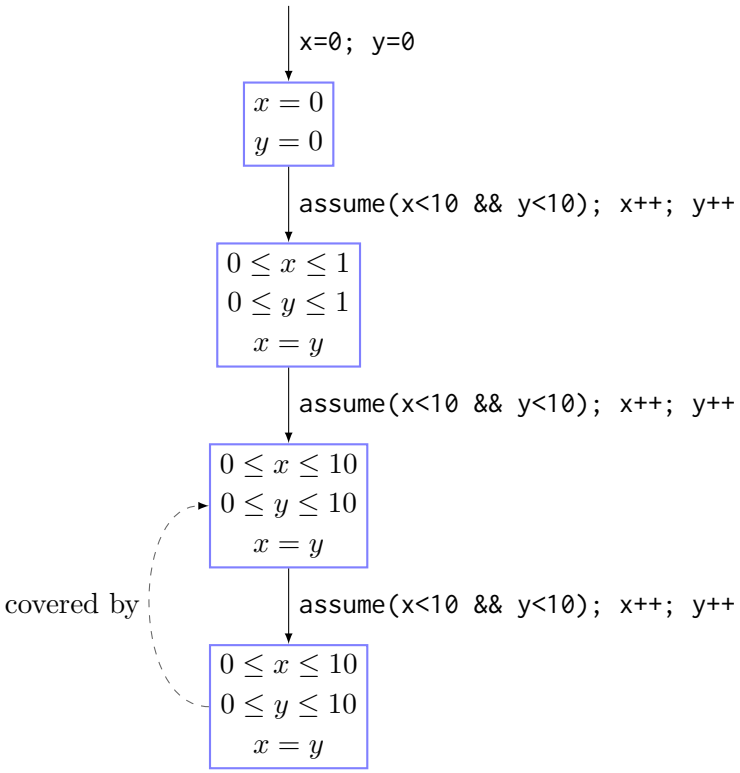


FIGURE 4.5: ART generated by local policy iteration algorithm for the input program in Figure 4.4.

4.4.2 Generating Templates from Interpolants

The overall algorithm for generating templates from an ART generation algorithm parameterized by an abstract interpretation and an interpolation procedure is given in Algorithm 4.3. As before in Algorithm 4.1, we start with an empty set of templates and repeatedly restart the analysis with an updated set (line 8). If the ART-generating analysis produces an abstract

Algorithm 4.3 Template Refinement Using Interpolants

```
1: Input: CFA P, error property E
2: Output: verification verdict
3: ▷ Set of TCD templates
4: T ← ∅
5: while true do
6:     ▷ Generate a sound fully expanded ART for P
7:     ▷ by abstract interpretation in the TCD T.
8:     A ← ARTAnalyze(P, T)
9:     if E is not reachable in A then
10:        return TRUE
11:    else
12:        P ← path from root to E in A
13:        ▷ Concrete semantics of CFA transitions in P
14:        π ← [[P]]
15:        if π ∧ E is feasible then
16:            return FALSE
17:        else
18:            I ← interpolation sequence over elements of π and E
19:            T′ ← T ∪ GenerateTemplates(I)
20:            if T′ = T then
21:                ▷ No further refinements are possible
22:                return UNKNOWN
23:            else
24:                T ← T′
25:            end if
26:        end if
27:    end if
28: end while
```

reachability tree entailing the unreachability of the error property, we conclude that the program is safe (line 10). Otherwise, we can find a *path* in the generated tree from the initial to the error state (line 12). In CEGAR [Cla+00] spirit, if the path is *feasible* (that is, composition of concrete semantics along the path returns a non-empty set of states), we have found a concrete counterexample (line 16). Otherwise, we generate *interpolants* from the unfeasible sequence of formulas associated with *concrete* transitions along the tree branch, and the final error state (line 18). We enforce termination by requiring that each iteration updates the set of templates (line 24), and adding a size restriction on templates generated from interpolants (line 19).

Again, consider analyzing the program in Figure 4.4 using an empty set of templates $T \equiv \emptyset$. Suppose we wish to verify the property $P \equiv x \leq 10$. In that case, performing the postcondition computation after the very first edge yields an abstract state $\top$, which violates P. Thus we ask an SMT solver to generate sequential interpolants for the sequence of formulas $x' = 0 \wedge y' = 0, x' > 10$. A possible interpolant is $I \equiv x' = 0$.

In order to synthesize a new set of templates from I we consider all variables occurring in I, and we perform enumerative template synthesis on those. In our example, that simply means adding the templates $\{x, -x\}$ to T. Rerunning the analysis with the updated template set lets us verify that universally at n_a we have $x \leq 10$.

4.4.3 Guiding the Interpolation Procedure

Unlike predicate abstraction or explicit state model checking, the *abstract transitions* in abstract interpretation between two consecutive states in the generated ART may result from the widening transition (or value determination in case of LPI), which might not correspond to a concrete transition *regardless* of what templates are included in T. In our running example, a transition from ART node with an associated state $x \leq 1$ to a new node with a state $x \leq 10$ always remains unfeasible. This is similar to the *leaping counterexamples* of LoopFrog [Kro+08].

Unfortunately, such jumps have the potential to confuse the interpolation procedure. We continue with the running example in Figure 4.4, verifying the property $P \equiv x \leq 5 \vee y \leq 10$ with a set of templates $T \equiv \{x\}$. The analysis generates the path over the states ($x = 0, 0 \leq x \leq 1, 0 \leq x \leq 10$) before the property violation happens. Thus we perform sequence interpolation over S:

$$S \equiv (x = 0 \wedge y = 0, x' = x + 1 \wedge y' = y + 1, x'' = x' + 1 \wedge y'' = y' + 1, x'' > 5 \wedge y'' > 10) \quad (4.1)$$

Unfortunately, due to the "leaping" transition from $x \leq 1$ to $x \leq 10$, a valid interpolation sequence for S may be:

$$I \equiv (x = 0, x' = 1, x'' = 2) \quad (4.2)$$

In this case, I contains only the variable x, and our refinement procedure can not proceed. We address this issue by performing *weakening* on the formulas used for interpolation. Consider computing an interpolant I for a path given by a semantics of a sequence of edges $\tau_0 \dots \tau_n$. Let T be a set of templates describing the used TCD, and let V be the set of all variables occurring in T. As we compute I for the purpose of *mining* it for the interesting variables not already present in V, we would like to enforce I not to contain variables from V in the first place. We do so by replacing each formula τ_i with its weakening $\exists V.\, \tau_i$, which enforces the interpolant to only contain the variables from $\mathbf{x} \setminus V$.

Going back to our example, after performing the weakening and quantifier elimination, the sequence given to the interpolation procedure is:

$$S \equiv (y = 0, y' = y + 1, y'' = y + 1, y'' > 10) \quad (4.3)$$

which *forces* a solver to generate an interpolant over y.

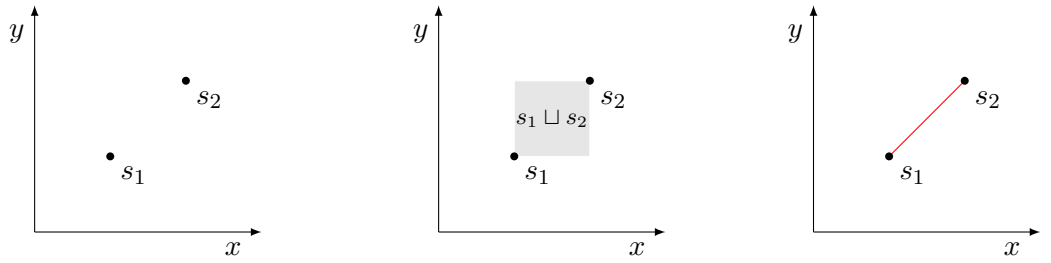
Such a modification increases the applicability of our technique, yet raises the question of *completeness*: in general, using the interpolant-based invariant synthesis, can we eventually synthesize all templates which would be given by the enumerative invariant synthesis? Unfortunately, the result is negative even for a non-relational domain, as we can still get an interpolant giving no useful refinement, which is demonstrated in Example 4.3.

Example 4.3 (Lack of Completeness)**.** Again, consider analyzing a modified two-counter program P from the Figure 4.6. We wish to establish a property $y \leq 10$. Such a program can be verified using abstract interpretation with the set of templates $T \equiv \{x, y, z, -z\}$, as it supports to generate an inductive invariant $x \leq 10 \wedge y \leq 10 \wedge z = 0$, as the looping transition relation already "connects" the values of x, y and z.

Consider starting to analyze P with a template constraints domain $T \equiv \{y, -y\}$. The analysis produces a sequence of states $y = 0, 0 \leq y \leq 1, y \geq 0$ before a property violation happens. After applying the weakening, this gives a following sequence to the interpolation

```
1  int x = 0;
2  int y = 0;
3  int z = 0;
4  while (x < 10) {
5      x++;
6      y++;
7      z = x - y;
8  }
9  assert(y <= 10);
```

FIGURE 4.6: Modified Two-Counter Program



(A) states in a TCD (B) join of s_1, s_2 in a "box" TCD (C) convex hull of s_1, s_2

FIGURE 4.7: Generating Templates from Convex Hull. Consider analyzing the program in Figure 4.4 in a template constraints domain $T \equiv \{\pm x, \pm y\}$. The constraints associated with two produced states s_1 and s_2, corresponding to the initialization and a single loop transition respectively are shown in Figure 4.7a. When performing the join in the domain given by T we end up with a box $x \in [1, 2] \wedge y \in [1, 2]$ as shown in Figure 4.7b. However, if we instead find the convex hull of the constraints associated with s_1 and s_2 we get a stronger constraint $x = y \wedge x \in [1, 2]$ shown in Figure 4.7c.

procedure, again generated from the concrete semantics of CFA edges and the property violation:

$$S \equiv (x = 0 \wedge z = 0, x' = x + 1, x'' = x' + 1, y'' > 10) \tag{4.4}$$

Unfortunately, at this point the interpolation procedure is stuck, as the weakened sequence becomes feasible, and no new interpolant can be produced.

We present the evaluation of interpolation-based template synthesis in Section 4.6.3, which shows its effectiveness as compared to a refinement-based approach.

4.5 Template Synthesis Using Convex Hull

As a motivating example, we analyze the program shown in Figure 4.4 in the template constraints domain $T \equiv \{\pm x, \pm y\}$. Compared to the polyhedral abstract interpretation, our analysis arrives at a suboptimal invariant.

We present two approaches for synthesizing templates by performing polyhedral manipulations. Our *offline* approach, described in Section 4.5.2, records the templates generated using convex hull, and then restarts the exploration if the previous precision was not sufficient for proving the target property. The *online* approach, described in Section 4.5.3, instead injects the discovered templates *during* the analysis, by heavily modifying the postcondition and join operation, effectively running a hybrid of local policy iteration in TCD and abstract interpretation in polyhedra.

4.5.1 Background: Abstract Interpretation in Polyhedra Domain

A polyhedron P with n constraints over m variables could be defined using a *constraint*-based representation by a matrix $A \in \mathrm{Mat}^{n \times m}$ and a vector $\mathbf{b} \in \mathbb{R}^n$:

$$A\mathbf{x} \leq \mathbf{b} \tag{4.5}$$

Alternatively, P can be defined using a *generator*-based description, using a set of points $S \subseteq \mathbb{R}^m$ and a set of *rays* $R \subseteq \mathbb{R}^m$, where P is defined to be a convex hull over S and R.

Traditional polyhedral abstract interpretation [CH78] proceeds as follows. The post-image computation requires constraint representation, and adds a new constraint on the existing polyhedron imposed by the processed statement. For example, the post-image of the polyhedron $P \equiv x > 0$ under the transition `assume(y > 10)` is a new polyhedron $P' \equiv x > 0 \wedge y > 10$. Projection can be performed on the result to get rid of the constraints associated with dead variables.

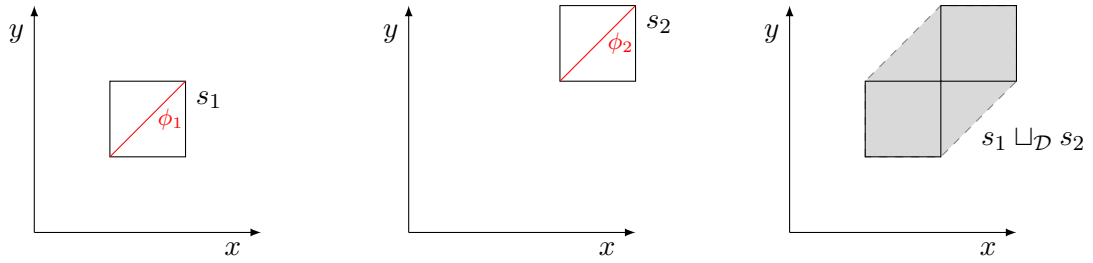
The join on two input polyhedra P_1 and P_2 is performed by computing the convex hull over the two: e.g. the join of $P_1 \equiv x = 1 \wedge y = 1$ and $P_2 \equiv x = 2 \wedge y = 2$ is $x = y \wedge 1 \leq x \leq 2$, as demonstrated in Figure 4.7. As seen from this example, the result of the join is not obvious from the constraints representation, and traditionally the polyhedra are converted to the generator form first [CH78]. However, recently new algorithms were proposed for performing join purely in the constraints representation [FB14].

Let T be the set of templates which appear in the constraints representation of P (e.g. $x+3y$ for a constraint $x + 3y \leq 10$). Now consider how new templates can appear in T throughout the analysis, if P is regularly updated to the value of a candidate inductive invariant:

- *Postcondition:* the postcondition computation for assignments and guards generates the constraint syntactically present in the guard: e.g. processing a statement `x = y + z` results in a constraint $x = y + z$ supported by templates $\pm(x - y - z)$.
- *Projection*: as described in Section 4.3, new directions can be generated by performing the projection operation on a candidate invariant in order to remove irrelevant variables and reduce the number of constraints. For example, projecting away the variable y from a constraint system $y = 2z \wedge x = y$ generates the new templates $\pm(x - 2z)$.
- *Convex Hull*: the result of the convex hull computation gives rise to new templates not originally syntactically present in the program. Such new templates can be seen as *generalizations*, aiming to extrapolate the evolution under the loop.

4.5.2 Offline Refinement Approach

Consider analyzing a program P with an abstract interpretation in a template constraints domain in order to prover a property $\neg E$ over a set of templates T. In the offline refinement approach, during each join operation over TCD states s_0, s_1 we compute the convex hull over their associated constraints, resulting in the polyhedron H. We widen H with respect to a set of constraints appearing in s_0 and then convert the result to a constraints-based representation C. Let T' be the set of all templates which appear in C during the analysis. If the found inductive invariant does not imply $\neg E$, we *restart* the analysis with a set of templates $T \cup T'$. The refinement continues until either the desired property can be proven, or no new templates are generated. Consider applying the offline refinement algorithm to the running example in Figure 4.4 and starting with a template set $T \equiv \{\pm x, \pm y\}$ in order to prove the property $x = y$.



(A) state space after initialization is ϕ_1, abstraction produces s_1

(B) after one iteration state space is ϕ_2, abstraction is s_2

(C) gray box is a convex hull of s_1 and s_2: the constraint $x = y$ is not recovered

FIGURE 4.8: Illustration of the precision loss when analyzing the program in Figure 4.9 with the template set $T \equiv \{x, y\}$. States ϕ_1 and ϕ_2 denote the least convex invariant on program variables after one and two iterations respectively, while s_1 and s_2 are their abstractions in the template constraints domain.

The first analysis run returns an invariant $0 \leq x \leq 10 \wedge 0 \leq y \leq 10$ and a new set of templates $T' \equiv \{\pm(x - y)\}$ derived from the convex hull, as shown in Figure 4.7. After restarting the algorithm with a template set $T \cup T' \equiv \{\pm x, \pm y, \pm(x - y)\}$ the required property $x = y$ is proven.

4.5.3 Online Injection Approach

The offline approach as presented in Section 4.5.2 is potentially wasteful, as it may require many restarts, recomputing the same invariant many times over. Thus it is more desirable to use the templates discovered from the convex hull operation *during* the analysis. Let s_1 and s_2 be two TCD states on which we are performing the join, and let s' be the resulting state. The set T' denotes templates which appear in the convex hull h of states s_1, s_2. We want to include a bound on a template $t \in T'$ in state s'.

Though it seems natural to set $s'[t]$ to $h[t]$ (the bound on the template t in the constraint-based representation of the convex hull), such a construction leads to a precision loss, as shown in Figure 4.8. The loss is caused by the fact that the new template t is discovered *too late*: by the time we are calculating the join, the analysis has already propagated the over-approximating bound on t, potentially even reporting spurious reachability of the error property. Thus in order to keep the precision of the polyhedra domain, more fundamental modifications to the analysis are required.

Recall that we are primarily interested in the template constraints domain as it can be analyzed using policy iteration (Chapter 3). In this section we show instead how abstract interpretation in the polyhedra abstract domain can be modified to benefit from the precision resulting from the policy iteration algorithm. Unlike the previous sections, a familiarity with Chapter 3 is necessary for understanding.

Policy Iteration in Polyhedra Abstract Domain To address this loss of precision, we modify the local policy iteration algorithm to perform different strongest postcondition and join operations.

We modify the postcondition computation (previously described in Algorithm 3.2) to return the *least abstraction* in the polyhedra domain using the path focusing [MG11] approach. Recall that the smallest polyhedral abstraction does not exist in general (Section 2.8.2), but it *does* in case all atoms of the abstracted formula τ are linear constraints over program variables, This

```
1  int x = input();
2  assume(0 <= x && x <= 1);
3  int y = input();
4  assume(x == y);
5  while (input()) {
6      x++;
7      y++;
8  }
9  assert(x == y);
```

FIGURE 4.9: Example Program For Online Convex Hull Synthesis

Algorithm 4.4 Exact Postcondition for Template Constraints Domain

1: **Input:** LPI state s_0, transition $\tau(\mathbf{x} \cup \mathbf{x}')$
2: **Output:** LPI state s'
3: $\phi_0 \leftarrow \exists \mathbf{x}. [\![s_0]\!](\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}')$
4: $s' \leftarrow \bot$
5: **while** $\exists \mathcal{M}. \mathcal{M} \models \phi_0 \wedge \neg [\![s']\!]$ **do**
6: $\quad \pi \leftarrow$ disjunction-free strengthening of $\phi_0 \wedge \neg [\![s']\!]$ modelled by $\mathcal{M}$
7: $\quad h \leftarrow \textsc{ConvexHull}(s', \pi)$
8: $\quad \mathit{constraints} \leftarrow \textsc{ToConstraints}(h)$
9: $\quad$ **for all** (template t, bound d) $\in \mathit{constraints}$ **do**
10: $\quad\quad$ **if** $s'[t] < d$ **then**
11: $\quad\quad\quad s'[t] \leftarrow (d, \pi, s_0)$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: **end while**
15: **return** s'

can be seen by converting τ into a DNF form: then each disjunct represents a polyhedra, and a union over a finitely many polyhedrons admits a polyhedral convex hull.

Our postcondition algorithm constructs an output LPI state s' (Definition 3.1) from an input LPI state s_0 and a transition relation τ. Recall that in order to construct an LPI state, we need to reconstruct the policy for each template. As before, we reconstruct the policies from models returned by the solver.

In Algorithm 4.4 we start by converting the input state to a formula (line 3), and by temporarily setting an output s' to an empty state $\bot$ (line 4). Then while there exists a vector $\mathcal{M}$ inside ϕ_0 but outside of s' (line 5), we derive a disjunction free strengthening π of ϕ_0 modelled by $\mathcal{M}$. The procedure for generating π described in Algorithm 3.2 (effectively, π is a *policy*): basically, each disjunction in ϕ_0 is recursively replaced with a disjunct modelled by a $\mathcal{M}$. For example, for $\phi_0 \equiv x > 10 \vee x = 0$ and $\mathcal{M} \equiv \{x : 11\}$, we have $\pi \equiv x > 10$. Observe that for a formula ϕ_0 where all atoms are linear inequalities, π is a polyhedron.

We proceed to compute a convex hull h of π with a state space described by s' (line 7). Then for each template t and its corresponding bound d in the constraints based representation of h which are not already subsumed by s', we grow s' by adding a mapping from t to the new bound d and the policy π, where the backpointer is the previous state s_0 by definition (line 11). The process is repeated until a fixed point is reached. Algorithm 4.4 has the following properties:

- *Termination:* at each iteration the algorithm selects a *new* under-approximation π of ϕ_0,

Algorithm 4.5 Online Refinement Join Operator

1: **Input:** node n, previous abstract state a_0, associated formula ϕ_0, new abstract state a_1, associated formula ϕ_1
2: **Output:** new joined state a'
3: ▷ Join in the template constraints domain, Algorithm 3.3.
4: $a' \leftarrow \textsc{Join}(n, a_0, a_1)$
5: ▷ Convex hull in constraints representation.
6: $h \leftarrow \textsc{ConvexHull}(a_0, a_1)\ \nabla a_1$
7: **for all** template $t \in h$ **do**
8: $\quad d \leftarrow \max t^\top \mathbf{x}$ s.t. ϕ_0
9: $\quad \pi \leftarrow$ policy used to derive d
10: $\quad a'[t] \leftarrow (d, \pi, a_0)$
11: **end for**
12: **return** a'

and as there are only finitely many such under-approximations (finitely many policies), the algorithm terminates.

- *Precision:* the algorithm computes the convex hull of all polyhedrons in the disjunctive normal form of ϕ_0, which is its least polyhedral abstraction.

Once we have changed the strongest postcondition operator to perform a modified version of path in the polyhedra domain, we can change the "Join" operator (previously defined in Algorithm 3.3) of the local policy iteration to insert the newly derived templates without the precision loss. The new algorithm listing is given in Algorithm 4.5. We start by calling the join operation in the template constraints domain, which simply computes pairwise upper bound for each template (line 4). Then we compute the convex hull of two input states (line 6), and we perform polyhedra widening of the resulting state with a_1 in order to enforce termination. For all templates t occurring in h, we derive the new bound for the resulting state by performing maximization of ϕ_0 in the direction of t (line 8), deriving the new policy from the resulting model (same approach as before). Finally, the generated state a' is returned.

The resulting algorithm defined by the new join and postcondition operators, effectively runs abstract interpretation in the polyhedra domain, using value determination as a more precise widening. The widening operation is used to avoid generating infinitely many templates.

Example 4.4 (Online Injection Approach for Figure 4.9)**.** Applying the online injection approach to the motivating example leads to the following sequence of steps:

- Initial postcondition: computing the post-image of the initial state $a_0 \equiv \{\}$ under the transition relation $\tau_i \equiv 0 \leq x' \leq 1 \wedge y' = x'$ using Algorithm 4.4 generates a new LPI state:

$$a_1 \equiv \{x : (1, \tau_i, a_0), -x : (0, \tau_i, a_0), y : (1, \tau_i, a_0), -y : (0, \tau_i, a_0),$$
$$y - x : (0, \tau_i, a_0), x - y : (0, \tau_i, a_0)\}$$

which represents the polyhedral abstraction $0 \leq x \leq 1 \wedge y = x$ of τ_i, and records the policy meta-information.

- Postcondition after looping transition: the post-image of a_1 under $\tau_l \equiv x' = x{+}1 \wedge y' = y{+}1$ results in a state:

$$a_2 \equiv \{x : (2, \tau_l, a_0), -x : (1, \tau_l, a_1), y : (2, \tau_l, a_1), -y : (1, \tau_l, a_1),$$
$$y - x : (0, \tau_l, a_1), x - y : (0, \tau_l, a_1)\}$$

- Join of a_0 and a_1 keeps the bound on $\pm(x - y)$, while the value determination widens the bounds on $\pm x, \pm y$ to $+\infty$, resulting in a state a_3:

$$a_3 \equiv \{y - x : (0, \tau_i, a_0), x - y : (0, \tau_i, a_0)\}$$

- The new postcondition computation produces the state subsumed by a_3, and the computation converges. The resulting precision is sufficient to verify the `assert` statement.

4.5.4 Algorithm Properties

As stated in Section 2.8.2, least inductive invariant may not exist even for a linear transition system, and thus our refinement algorithm can not in principle find the tightest inductive invariant (unlike the case for a fixed set of templates T).

Observe that the termination of both approaches is ensured by the use of polyhedra widening operators, and as widening on a single location can be applied only finitely many times (cf. Section 2.7.3), the analysis sequence guaranteed to converge.

4.6 Evaluation

All experiments were performed on a cluster of machines with Intel Xeon E5-2650 CPU @ 2.60 GhZ, 32 cores and 135 GB of RAM.

4.6.1 Live Variables

We compare three different filtering strategies for templates derived from the enumerative template synthesis (Algorithm 4.1):

All Live Synthesize only those templates where all variables are alive.

One Live Synthesize only those templates where at least one variable is alive.

No Filtering Synthesize all templates, do not take liveness into account.

As before in Section 3.5, the experiments were performed on the "Loops" category of SV-COMP. For evaluation, each verification task was given a time limit of 200 seconds and a memory limit of 16GB. The quantile plot comparing these filtering approaches is shown in Figure 4.10, with the number of programs each approach could successfully verify in brackets. The difference in time clearly shows a large performance gain when only the templates consisting purely of live variables were considered. Surprisingly, experiments show only little performance difference between "All live" and "One Live" strategies, perhaps due to the relatively small program size in the considered dataset.

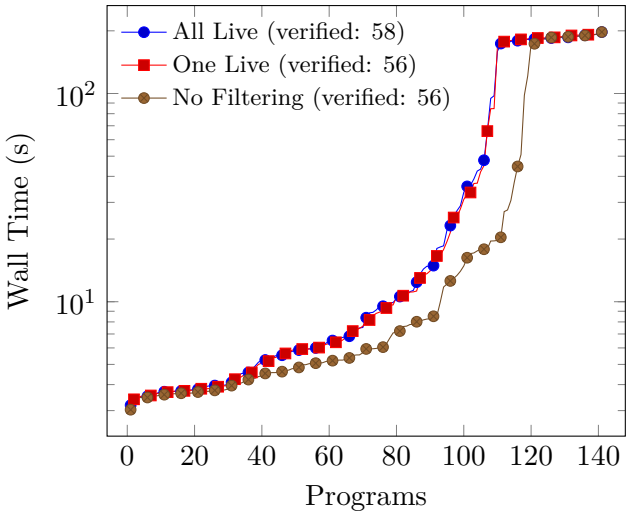


FIGURE 4.10: Timing evaluation for liveness filtering. Each data point corresponds to every fifth analyzed program, each data series is sorted separately.

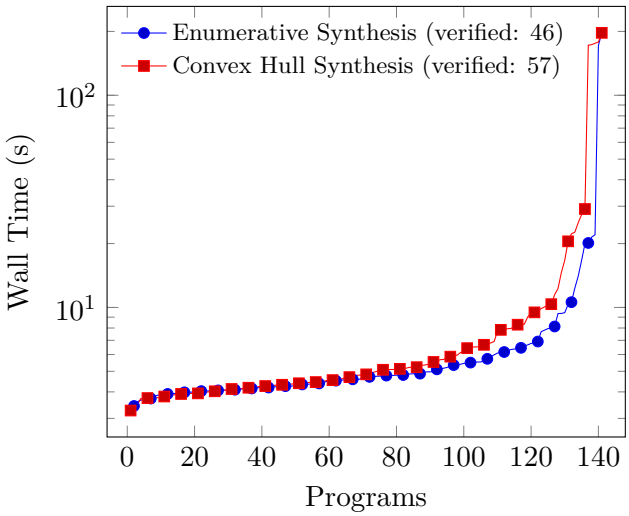


FIGURE 4.11: Timing evaluation for convex hull template synthesis against refinement-based template generation.

4.6.2 Convex Hull Template Synthesis

We compare the template synthesis using the offline convex-hull based refinement described in Section 4.5.2 against enumerative synthesis template synthesis from Section 4.2 filtered by liveness. Again, the experiments were run on the "Loops" category of SV-COMP, with a time limit of 200 seconds per verification task. The resulting quantile plot is shown in Figure 4.11. The graph shows us that the convex-hull based approach is slower, as it requires more refinements, yet it could successfully verify considerably more programs.

Due to the lack of time, the online injection approach was not implemented.

4.6.3 Interpolation-Based Template Synthesis

We compare the analysis within intervals abstract domain (filtered by liveness) against the interpolation-based refinement on the "DeviceDrivers64" benchmarks set of SV-COMP. We have chosen this benchmark set as it contains large programs (many thousands of lines of code),

	vs. Interval	vs. Refinement	**Verified**
Interval Domain		40	1047
Interpolation Refinement	60		1067

TABLE 4.1: Results for Interpolation Refinement

the interval domain is sufficient for proving many properties, and due to the program size plain LPI fails to analyze many programs under the time limit even using the intervals. Unlike the other evaluation data sets, we set the time limit to 100 seconds per benchmark. The results are shown in Figure 4.12. We can see from the graph that applying the interpolation-based synthesis speeds up the performance, which lets the analysis handle more programs before the time limit. From the results in Table 4.1 we can see that the refinement-based approach is able to prove 60 benchmarks LPI with the intervals domain could not handle under the time limit. Yet the procedure diverges for 40 programs which could have been proven before.

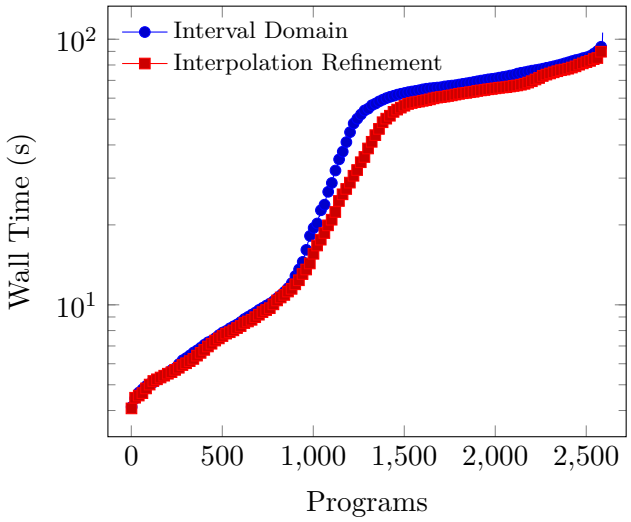


FIGURE 4.12: Timing evaluation for interpolation-based template synthesis against the intervals domain simulation.

4.7 Conclusion

In this chapter we have shown that even most naive template synthesis approaches based on brute-force enumeration can be surprisingly effective for program verification in combination with LPI. We have demonstrated that subsequent filtering based on liveness can greatly increase the performance without any loss in precision, and that template synthesis using interpolation may be used to successfully verify large programs.

The algorithms presented in Section 4.5 bridge the gap between template-based approaches requiring manual annotation, and polyhedra-based approaches which can synthesize new directions automatically, and can be applied outside of the context of LPI.

Practical implementation of the online injection-based template synthesis approach remains an item for future work.

Chapter 5

Generating Summaries Using Policy Iteration

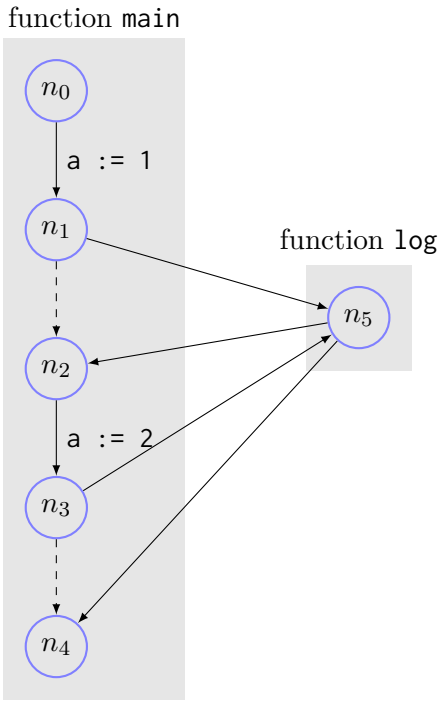
5.1 Introduction

As discussed in the previous chapters, traditional approaches based on abstract interpretation compute an inductive assertion map (Section 2.5.3) from CFA nodes to predicates over the program variables. Such an analysis is called *intra*procedural, as it only considers the states inside the analyzed procedure.

A naïve approach for supporting *inter*procedural analysis is to simply encode function calls and returns as ordinary CFA edges, and to obtain an inductive assertion map for a resulting CFA. Yet such an encoding results in a large precision loss, as shown in Figure 5.1. A value of the variable `a` is always 2 at the program location n_4, yet due to the *spurious* program path (Definition 2.4) $\pi \equiv (n_0, n_1, n_5, n_4)$ being feasible in the shown CFA, an analysis would not be able to prove the assert statement.

```
1  void log() {}
2  int main() {
3      int a = 1;
4      log();
5      a = 2;
6      log();
7      assert(a == 2);
8  }
```

(A) analyzed program



(B) naive CFA encoding

FIGURE 5.1: Example of precision loss due to `goto` function call encoding.

$$\frac{\{P\}b = f(a)\{Q\} \vdash \{P \wedge p = a\}B_f\{Q \wedge p = a \wedge b = r\}}{\{P\}b = f(a)\{Q\}}$$

FIGURE 5.2: Hoare rule for summary instantiation. Recall that a triple $\{P\}\texttt{OPS}\{Q\}$ states that "if P holds, and the control passes through the statement `OPS`, then Q holds. The variable p denotes the set of parameters of f, r is a set of return values of f, and B_f is the set of instructions contained in the body of f. The rule states that if the fact that all calls to f are described by the pair (P, Q) entails the fact that body of the function f (B_f) also satisfies (P, Q) (subject to the parameter and return variable renaming), then (P, Q) summarizes the effect of f.

The path π is spurious as it can never occur during the execution of the program 5.1, due to the fact that the control has to return to the calling context in the node n_2 instead of jumping to n_4. A program path is called *valid* if it respects that each procedure returns to the site of the most recent call. Thus in interprocedural program analysis we are interested in finding smallest invariants which are inductive with respect to all valid program paths.

The classical paper of Sharir and Pnueli [SP81] proposes two solutions for finding such invariants: the *summary* approach, and the *callstrings* approach. In the callstring approach, an abstract state is extended to include the traversed call-sites, which allows an analysis to only propagate the information along the valid program paths. Observe that without applying an abstraction this amounts to dynamically inlining the program during the analysis, as we store a separate invariant candidate per each program location and callstack. While inlining does offer fully context sensitive interprocedural analysis, the obvious downsides include inability to handle recursive programs, and exponential state-space explosion for large programs with many procedures.

The *summary* approach, which we use in this chapter, is based on computing a *two-state* invariant for each function f, which over-approximates all possible transitions within f. Instead of associating a predicate over $\mathbf{x}$ with each program location, summary-based approaches associate predicates over *input* variables $\mathbf{x}$ and *output* variables $\mathbf{x}'$ with program functions. As opposed to stating all possible values for variables at a given location, summaries over-approximate all possible transitions through the function: e.g. a summary $x \geq 0 \wedge x' = x + 1$ states that the program variable `x` is always positive at the function entry, and is incremented by one by the time control reaches the function exit.

The Hoare rule for proving programs using summaries [Hoa71] is shown in Figure 5.2. It states that if a body of a recursive function f satisfies the Hoare tuple (P, Q) *assuming* that S holds for all recursive calls, then f satisfies (P, Q). Such a summary (P, Q) is called *inductive*.

For example, for a recursive function

```
int sum(int i) { return i <= 0 ? i : i + sum(i - 1)}
```

a summary $i \geq 0 \implies r \geq 0$ is inductive (r models the returned variable), while a summary $r \geq 0$ is not, due to the possibility of returning a negative value when the input is negative, even assuming that all recursive calls satisfy $r \geq 0$.

Once a summary s for a function f has been computed, a summary based analysis effectively replaces every call $\mathbf{o} = f(\mathbf{p})$ with a statement $\mathbf{o}$ `= input(); assume(`$s(\mathbf{o}, \mathbf{p})$`)`, where `input` is function returning $\|\mathbf{o}\|$ non-deterministic values.

5.1.1 Contribution

The contributions of this chapter are two-fold. Firstly, we develop a top-down algorithm for summary generation using an intraprocedural abstract interpretation analysis. Our algorithm computes a single summary per each called procedure which has a union of all feasible calling contexts in the analyzed program as a precondition (that is, we only summarize behaviors which were deemed feasible by our analysis). We rely on the "top-down" summary generation approach, where summaries are derived from the memoized under-approximating invariant candidates, which assume that all nested function calls satisfy the existing summaries. In many ways our algorithm is similar to a context-sensitive interprocedural approach described in *Principles of Program Analysis* [NNH99, Figure 2.10], yet it is performed in a more general context of abstract interpretation with a relational domain of a potentially infinite height.

We develop the algorithm in a framework which assumes pass-by-value semantics (no aliasing) and lack of global variables (modelled using implicit return of to-return variables at function exit). No further restrictions are introduced (we support loops, recursion, mutual recursion, etc.), and furthermore in Section 5.9 we extend the framework to global variables, and in Section 5.9.4 we outline how multiple summaries can be supported.

The second contribution is that we provide an efficient application of policy iteration to the summary generation problem by parameterizing the developed algorithm with the local policy iteration (LPI, Chapter 3). Such a parameterization guarantees that the resulting inductive invariant is smallest possible for the *chosen summary structure*[1], and a better result is unattainable inside the given abstract domain.

Chapter Outline We start by developing a new formalism for interprocedural programs in Section 5.3.1. In Section 5.3.2 we state a computation model for such programs, redefining the notion of a *program path* and subsequently of an invariant. Then in Section 5.3.3 we specify the equation system predicates over program states and *summaries* have to satisfy in order to form an inductive invariant for an interprocedural program. In Section 5.4 we get back into the abstract interpretation domain, restating the notion of an inductive invariant assuming *both* summaries and candidate invariants are represented as elements of an abstract domain.

We show how pure policy iteration (Chapter 3) can be applied to solve an equation system describing such an inductive summary in Section 5.5; this part can be skipped by a reader only interested in the overall summary generation algorithm.

In Section 5.6 we state Algorithm 5.1 for summary generation using an intraprocedural analysis (including LPI), and we revisit our running example with it. Then we discuss the algorithm properties in Section 5.7 and our implementation in Section 5.8. We describe a number of extensions and optimizations in Section 5.9. Finally, we evaluate our algorithm on the SV-COMP dataset in Section 5.10, and we conclude in Section 5.11.

5.2 Related Work

The literature on traditional abstract interpretation contains a large body of works on interprocedural analysis. The key difference for many such approaches is that in a case of a finite lattice and associative program operators it is possible to find the least inductive invariant with

[1] In this chapter we mostly assume the simplest structure where a single summary is generated per function. Naturally, generating more summaries (e.g. one summary per callsite per function) can achieve greater precision, as discussed in Section 5.9.4.

respect to all valid program paths exactly, even in polynomial time, as done in an influential paper by Reps et al. [RHS95]. Many of such approaches are summarized in a book *Principles of Program Analysis* [NNH99].

Ball and Rajamani [BR00] extend the summary generation algorithm on a finite lattice of dataflow facts to make it *path-sensitive* by including the strongest postcondition implied by all procedure calls in the summary. Their approach relies on strongest postcondition computation.

The first extension of abstract interpretation to recursive procedures [CC77b] dates back to the same year abstract interpretation was introduced.

A top-down algorithm for context-sensitive summary generation in an infinite abstract domain was proposed by Apinis et al. [ASV12]. The authors generate fully context sensitive summaries (effectively, a new summary for each new different abstract state at a callsite), and propose constraints-with-side effects as a formalism for expressing such analysis. Their work is similar to ours, with a number of key differences. Firstly, we focus on a *relational* domain, where often a single summary per procedure can provide a desired result (e.g. "a return value is one bigger than the input parameter"). Thus our main algorithm description only generates a single summary, yet we demonstrate an extension to multiple summaries in Section 5.9.4. During the computation procedure, at each step we compute the *outgoing successors* of each state (as opposed to re-calculating constraints on each state subject to incoming edges), which avoids the problem of "infinite number of variables affecting the constraint" [ASV12] entirely and does not require any additional formalism.

Ancourt et al. [ACI10] propose computing summaries in a "bottom-up" manner: each program statement is seen as a block which can be summarized, and by composing these blocks the summaries are obtained for all procedures, and eventually for the entire program. Their approach scales better than top-down summaries, yet suffers from the fact that no context information is available during the summary computation, and summaries can potentially include spurious behaviors which never occur during the run of a program. Zhang et al. [Zha+14] propose a combination of a top-down and a bottom-up analysis, which combines performance and precision of the two.

M. Müller-Olm and H. Seidl present a precise context-aware inter-procedural algorithm for inductive invariant generation in linear arithmetic [MS04], yet they heavily over-approximate the program semantics by abstracting away all program guards.

In a predicate abstraction domain, the SPACER algorithm [Kom+13] was proposed by Komuravelli et al. which uses predicates derived interpolants as well as a mixture of under- and over-approximations to effectively synthesize summaries for recursive procedures.

The constraint system we create in order to describe the summary applicability is very similar to a Horn clause encoding of programs with procedure calls [Bjø+15].

5.3 Background

In order to support interprocedural analysis we extend the program model from Section 2.2 with the definitions of a program and a function.

5.3.1 Interprocedural Program Model

The function definition extends that of a control flow automaton (Definition 2.3) by adding a set of *calledges* associated with function calls, a set of input and output variables, and a unique return point.

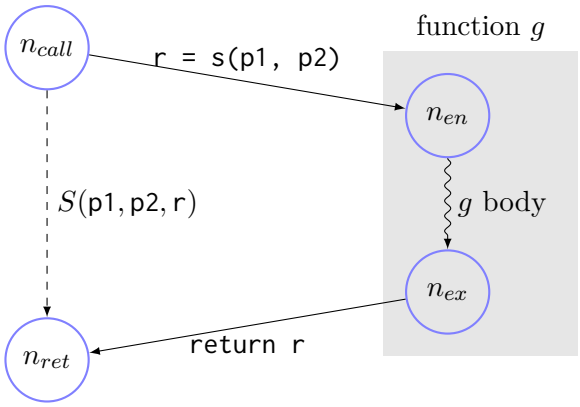


FIGURE 5.3: Illustration of a function call edge $(g, n_{call}, n_{ret}, \{p1, p2\}, \{r\})$. A call edge specifies the location of a function call, affected variables, and a return location. A valid over-approximation of semantics of a function call is an over-approximating summary $S(p1, p2, r)$ over the function parameters and the return value.

Definition 5.1 (Function)**.** A function f is a tuple $(nodes, edges, calledges, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$ where *nodes* is a set of control states modelling the program counter, $n_{ex} \in nodes$ is a unique function entry point, and $n_{ex} \in nodes$ is a unique exit point. The set $edges \subseteq nodes \times \mathtt{OPS} \times nodes$ denotes all possible transitions within the function together with their corresponding operators. The set $\mathbf{x}$ denotes all variables local to f, additionally $\mathbf{x}_i \subseteq \mathbf{x}$ is a tuple of input parameters, and $\mathbf{x}_r \subseteq \mathbf{x}$ is a tuple of returned variables. We assume that none of the parameters in $\mathbf{x}_i$ are modified by any transition in *edges*, which can be easily enforced by copying the parameter variables at the function start. The set *calledges* describes all function calls from f, where for each $(g, a, b, \mathbf{x}_p, \mathbf{x}_o) \in calledges$ an element $g \in F$ is a called function, $a \in nodes$ is a *callsite*, $b \in nodes$ is a *return* node: a node to which the control return once the function finishes its execution, $\mathbf{x}_p \subseteq \mathbf{x}$ is a tuple of passed parameters, and $\mathbf{x}_o \subseteq \mathbf{x}$ is a tuple of variables which assume the value given by the return variables of g after the function call.

An example of a call edge is given in Figure 5.3. We use the following vocabulary when referring to the nodes involved in a function call: the node from which there is an outgoing function call is called a *callsite*, the first node of a function is called an *entry* node, the last node is referred as an *exit* node, and finally the control gets back to the *return* node.

Definition 5.2 (Program)**.** A program is a tuple (F, f_m) where F is a set of *functions*, and $f_m \in F$ is a function which is run on a program start.

Note that our language does not have global variables, and writes to globals are modelled using multiple return values. Additionally, we do not define a `return` operator, as for a function $(nodes, edges, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$ all variables in $\mathbf{x}_r$ are returned once the control reaches n_r.

5.3.2 Invariants and the Computation Model

Recall that we have previously defined the invariant (Definition 2.5) with respect to all possible program paths (Definition 2.4). As demonstrated in the example of Figure 5.1 this definition is not sufficient for an interprocedural program, as it does not take the fact that the control must return to the node associated with the most recent function call into account. Thus we introduce a notion of a *valid interprocedural path* which respects the nesting of function calls.

We first have to extend the definition of a concrete state (Definition 2.2) to model the program callstack. We define a *stack frame* to be a tuple consisting of a concrete state and

a program call-edge (call-edge is replaced by an empty set for the first frame representing the program entry). A *program state* is a finite sequence of stack frames $p \equiv \langle f_0, \ldots, f_n \rangle$. In order to refer to the last frame of a program state p we use the notation $p|_{head}$, and a program state consisting of all other elements is referred to as $p|_{rest}$. Two frames $c_1 \equiv ((m_1, n_1), e_1)$, $c_2 \equiv ((m_2, n_2), e_2)$ (recall that a concrete state is a product of a variable-to-values mapping and a CFA node) can occur consequently in p only if n_1 belongs to a function f, n_2 belongs to a function g, and there exists a call-edge $e_2 \equiv (g, n_1, n_3, \mathbf{x}_p, \mathbf{x}_r)$ in f.

Definition 5.3 (Interprocedural Program Path)**.** An *interprocedural program path* is a sequence of program states $\langle p_0, \ldots, p_m \rangle$ where $p_0 \equiv \langle (m_0, n_0) \rangle$ and for any two consecutive states p_i, p_{i+1} we have that one of the following holds:

- *Consecution:* $p_i|_{rest} = p_{i+1}|_{rest}$, and $p_i|_{head} \equiv ((m_1, n_1), e_1)$, $p_{i+1}|_{head} \equiv ((m_2, n_2), e_1)$, and nodes n_1, n_2 belong to the same function f, and there exists an edge $(n_1, \mathtt{OP}, n_2)$, such that $m_2 \in [\![\mathtt{OP}]\!](m_1)$. Informally, only the head of the two stacks differs, and the transition taken from p_i to p_{i+1} happens inside f.

- *Function Call:* $p_{i+1}|_{rest} = p_i$, and $p_i|_{head} \equiv ((m_1, n_{call}), e_1)$, $p_{i+1}|_{head} \equiv ((m_2, n_{en}), e_2)$, n_{call} belongs to a function f, n_{en} belong to a function g, and $e_2 \equiv (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o)$ is a call-edge in f, such that $m_1|_{\mathbf{x}_p}[\mathbf{x}_p/\mathbf{x}_i^g] = m_2$, where $\mathbf{x}_i^g$ is a tuple of input parameters of g (recall that $m|_{\mathbf{a}}$ denotes the projection of a state described by m to the variables present in $\mathbf{a}$). Informally, the transition represents a call of a function g from f using the call-edge e_2, where we create a new element on the stack such that it is equal to the head of the previous stack, modulo projection to the parameter variables, and renaming of the passed parameters to the input variables of g.

- *Function Return:* $p_{i+1} = p_i|_{rest}$, and $p_i|_{head} \equiv ((m_1, n_{ex}), e_1)$, $p_{i+1}|_{head} \equiv ((m_2, n_{ret}), e_2)$, $p_{i+1}|_{rest}|_{head} \equiv ((m_3, n_{call}), e_2)$ n_{ex} belongs to a function g, n_{ret} belongs to a function f, and $e_2 \equiv (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o)$ is a call-edge in f, such that $m_1|_{\mathbf{x}_r^g}[\mathbf{x}_r^g/\mathbf{x}_o] \coprod m_3|_{\mathbf{x} \setminus \mathbf{x}_o} = m_2$, where $\mathbf{x}_r^g$ is a tuple of return variables of g, and $\coprod$ operator performs the disjoint union of two maps. Informally, the transition represents the return from g to f, where we pop from the stack, and the new assignment to variables is given by a disjoint union of two maps: the one containing the assignment to variables modified by the function call (state at the return value of the function, projected to return variables, and renamed to output variables of the call-edge), and the one containing the assignment to all the untouched variables (state at the callsite, projected to all local variables excluding those modified by the function call).

Finally, we are equipped to introduce an *interprocedural invariant* definition: a property I is an interprocedural invariant for a program P if and only if for all interprocedural program paths for P, for all program states of these paths, for all frames $f \equiv ((m, n), e)$ contained in program states, we have $m \models I$. Note that as all feasible intraprocedural program paths over-approximate all feasible interprocedural program paths, every intraprocedural invariant is also interprocedural. Yet the converse does not necessarily hold. In this chapter we shall refer to an interprocedural invariant as simply an "invariant".

5.3.3 Inductive Invariant and Semantics Equations

Recall that during an intraprocedural analysis we were looking for an invariant defined using an inductive assertion map (Definition 2.7), which is a mapping from CFA nodes to predicates

over program variables, satisfying the constraints of Equation 2.4. As before, we have to adapt the inductive invariant definition to the interprocedural analysis. We perform the analysis using *summaries*, which are predicates over the input and return variables for each procedure, over-approximating all possible transitions.

Definition 5.4 (Summary)**.** A predicate $S(\mathbf{x}_i \cup \mathbf{x}_r)$ is a *summary* for a function

$$f = (\mathit{nodes}, \mathit{edges}, \mathit{calledges}, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$$

occurring in a program P if and only if in all for all valid interprocedural program paths in P, for all program states of those paths, for all frames $f \equiv ((m, n_{ex}), e)$ contained in those states $m|_{\mathbf{x}_i \cup \mathbf{x}_r} \models S$.

Note that our summaries are only affected by the concrete data states associated with the function exit node. This is done for the following reasons: (i) we do not wish to summarize program paths which never leave the procedure due to endless loops or recursion; (ii) as our intraprocedural operators do not change the input parameters, it is sufficient to look at the states associated with the exit node.

For example, a summary $p \geq 0 \wedge r \geq 0$ for a function where the set of parameters is $\equiv \{p\}$ and the set of return variables is $\{r\}$ states that "the function is only called with a positive input, and the output for such calls is always positive". Note that Definition 5.4 states that the summary has to over-approximate all possible states within the function *with respect to all valid program paths*: that is, the summary does *not* over-approximate all behaviours of the procedure, if those behaviours do not actually occur in the program.

We define an inductive invariant for an interprocedural program using a set of summaries, one per function, and a set of inductive assertion maps, also one per function[2]. As before, every inductive program invariant is a program invariant.

Definition 5.5 (Inductive Program Invariant)**.** A set of summaries S_f indexed by the function name, and a set of inductive assertion maps I_f also indexed by the function name, form an inductive invariant for a program $P \equiv (F, f_m)$ if and only if the following rules universally hold for all functions $f = (\mathit{nodes}, \mathit{edges}, \mathit{calledges}, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$ and all values of $\mathbf{x}$:

$$\begin{aligned}
\text{Program Initiation:}\quad & I_{f_m}(n_i^{f_m}) = \top \\
\text{Consecution:}\quad & \text{for all } (a, \texttt{OPS}, b) \in \mathit{edges}: \\
& I_f(a)(\mathbf{x}) \wedge [\![\texttt{OPS}]\!](\mathbf{x} \cup \mathbf{x}') \implies I'_f(b)(\mathbf{x}') \\
\text{Function Call:}\quad & \text{for all } (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{calledges}: \\
& (\exists(\mathbf{x} \setminus \mathbf{x}_p).\, I_f(n_{call})(\mathbf{x}))[\mathbf{x}_p/\mathbf{x}_i^g] \implies I_g(n_{en}^g)(\mathbf{x}_i^g) \\
\text{Summary Coverage:}\quad & \exists \mathbf{x} \setminus (\mathbf{x}_i \cup \mathbf{x}_r).\, I_f(n_{ex})(\mathbf{x}) \implies S_f(\mathbf{x}_i \cup \mathbf{x}_r) \\
\text{Function Application:}\quad & \text{for all } (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{calledges}: \\
& \exists \mathbf{x}_o.\, I_f(n_{call}) \wedge S_g[\mathbf{x}_r^g/\mathbf{x}_o][\mathbf{x}_i^g/\mathbf{x}_p] \implies I_f(n_{ret})
\end{aligned} \tag{5.1}$$

In the rules above, we use existential quantification in order to perform projection: existentially quantifying a variable is equivalent to performing a projection on all other variables.

The "Program Initiation" and "Consecution" rule simply mirror those of Equation 2.4, stating that the invariant is inductive with respect to intraprocedural transitions and the

[2]In Section 5.9 we show how the model can be extended to support multiple summaries.

initiation condition is fulfilled. The "Function Call" rule ensures that the invariant associated with the function entry of the called function subsumes the invariant associated with the calling context (after projecting to parameter variables and renaming).

In the "Summary Coverage" rule we state that the invariant associated with the function entry (recall that we have postulated that function parameters can not be modified inside the function) and the function exit are both smaller than the corresponding summary. This fact ensures that all function calls which occur in a program are captured by the existing summaries, and that existing summaries capture all possible transitions expressed by the inductive assertion map associated within the called functions. The "Function Application" rule requires that if we replace a function call by its summary, the invariant at the return site will subsume the summary application to the callsite (given by conjunction of the invariant at the callsite with variables $\mathbf{x}_o$ projected out, and the summary application with input and parameter variables renamed).

5.4 Summaries as Abstract States

We would like to find the least inductive invariant for an interprocedural program in a given abstract domain $\mathcal{D}$. That is, we are looking for the smallest set of inductive assertion maps $I_f : \mathit{nodes} \to \mathcal{D}$ where *nodes* is a set of nodes in f, and all summaries are elements of the domain $\mathcal{D}$ as well.

Consequently, all inductive assertion maps for a program $P \equiv (F, f_m)$ are represented as a set of unknowns $I_f^n \in \mathcal{D}$ for a corresponding function $f \in F$ and for all n in the set of the nodes of f. A summary for a function f is represented as an unknown $S_f \in \mathcal{D}$.

We proceed to rewrite Equation 5.1 with those assumptions. We replace the implication with a $\preceq$ partial order relation associated with $\mathcal{D}$, we use the abstract semantics for the intraprocedural operator application, and we use the intersection operator in place of conjunction. Additionally, we replace existential quantification with an explicit projection operation. We obtain the following constraint set for a program P, which holds for every $f \equiv (\mathit{nodes}, \mathit{edges}, \mathit{calledges}, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$ in F (let m be the starting node for the main function f_m):

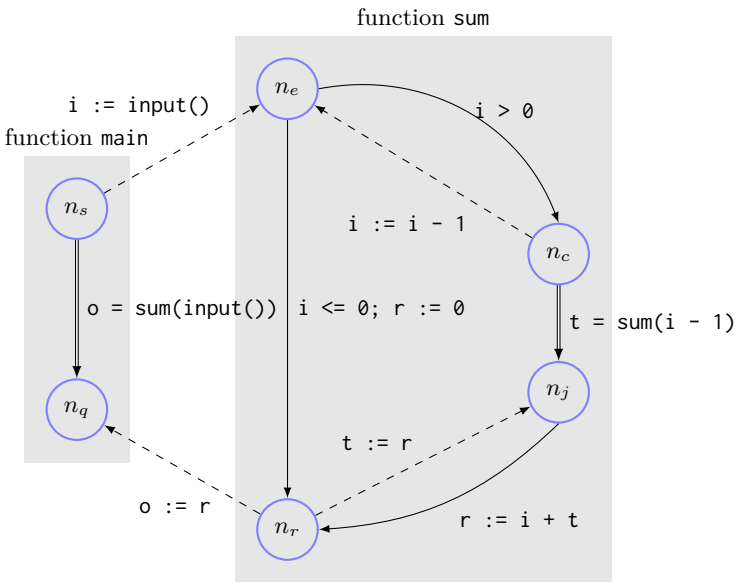
$$\begin{aligned}
\text{Program Initiation:}\quad & I_{f_m}^m = \top \\
\text{Consecution:}\quad & \text{for all } (a, \texttt{OPS}, b) \in \mathit{edges}: \\
& [\![\texttt{OPS}]\!]^\sharp (I_f^a) \preceq I_f^b \\
\text{Function Call:}\quad & \text{for all } (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{calledges}: \\
& I_f^{n_{call}}|_{\mathbf{x}_p}[\mathbf{x}_p/\mathbf{x}_i^g] \preceq I_f^{n_{en}} \\
\text{Summary Coverage:}\quad & I_f^{n_{ex}}|_{\mathbf{x}_i \cup \mathbf{x}_r} \preceq S_f \\
\text{Function Application:}\quad & \text{for all } (g, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{calledges}: \\
& I_f^{n_{call}}|_{\mathbf{x} \setminus \mathbf{x}_o} \sqcap S_g[\mathbf{x}_i^g/\mathbf{x}_p][\mathbf{x}_r^g/\mathbf{x}_o] \preceq I_f^{n_{ret}}
\end{aligned} \tag{5.2}$$

5.5 Applying Policy Iteration

Finding the smallest inductive assertion maps subject to the constraints of Equation 5.2 is solvable by policy iteration (Chapter 3) in case where an abstract domain is an instance of a

```
1  int main() {
2      int o = sum(input());
3  }
4
5  int sum(int i) {
6      int r, t;
7      if (i <= 0) {
8          r = 0;
9      } else {
10         t = sum(i - 1);
11         r = i + t;
12     }
13     return r;
14 }
```

(A) source code



(B) interprocedural dataflow graph

FIGURE 5.4: Running Example: Recursive Sum Program

template constraints domain (Section 2.8.3). The resulting inductive invariant and the summary are the *strongest possible* for a given program expressed in a given domain. The optimization problem which policy iteration is solving corresponds to minimizing a finite set of unknowns used to represent abstract states I_f^n in a complete lattice where the right hand side of every equation contains a monotone concave expression.

Example 5.1 (Running Example for Summary Generation)**.** We demonstrate the application of policy iteration for summary generation using the running example of Figure 5.4. We analyze the given program using the template $-i$ at node n_c, templates $-i, -t$ at n_j, templates $-i, -r, -(r-i)$ at n_r, and a single template $-o$ at n_q. Recall that negated templates are simply used to obtain the lower bounds on expressions. We do not add any templates to nodes n_s or n_e as the associated invariant is always $\top$.

A summary $S(i, r)$ is used to model the application of a function `sum`, and it is also tracked using the templates $-i$, $-r$ and $-(r-i)$.

In order to perform the minimization, we write down the system of semantical equations for control locations with a non-zero number of templates, using unknowns d_t^n for each location n and template t, and additionally unknowns s_t for each t mentioned in the summary. Thus we are looking for the smallest tuple

$$\mathbf{d} \equiv (d^c_{-i}, d^j_{-i}, d^j_{-t}, d^r_{-i}, d^r_{-r}, d^r_{-(r-i)}, d^q_{-o}, s_{-r}, s_{-(r-i)}) \tag{5.3}$$

satisfying the constraints of Equation 5.4:

$$\begin{cases} d^c_{-i} \geq \sup -i \text{ s.t. } \bot \vee i > 0 \\ d^j_{-i} \geq \sup -i \text{ s.t. } \bot \vee -i \leq d^c_{-i} \wedge -t \leq s_{-r} \wedge -(t-(i-1)) \leq s_{-(r-i)} \\ d^j_{-t} \geq \sup -t \text{ s.t. } \bot \vee -i \leq d^c_{-i} \wedge -t \leq s_{-r} \wedge -(t-(i-1)) \leq s_{-(r-i)} \\ d^r_{-i} \geq \sup -i \text{ s.t. } \bot \vee i \leq 0 \wedge r = 0 \\ \qquad \vee -i \leq d^j_{-i} \wedge -t \leq d^j_{-t} \wedge r' = i + t \\ d^r_{-r} \geq \sup -r \text{ s.t. } \bot \vee i \leq 0 \wedge r = 0 \\ \qquad \vee -i \leq d^j_{-i} \wedge -t \leq d^j_{-t} \wedge r' = i + t \\ d^r_{-(r-i)} \geq \sup -(r-i) \text{ s.t. } \bot \vee i \leq 0 \wedge r = 0 \\ \qquad \vee -i \leq d^j_{-i} \wedge -t \leq d^j_{-t} \wedge r' = i + t \\ d^q_{-o} \geq \sup -o \text{ s.t. } \bot \vee -o \leq s_{-r} \\ s_{-r} \geq d^r_{-r} \\ s_{-(r-i)} \geq d^r_{-(r-i)} \end{cases} \tag{5.4}$$

The constraint set of Equation 5.4 was generated from Equation 5.2 in a following way: the "Program Initiation" rule is implicitly fulfilled by the lack of templates associated with the program entry point, which therefore can take any value. Similarly, the "Function Call" rule is also fulfilled implicitly, as the starting invariant candidate associated with the node n_e is already $\top$. The "Consecution" rule is satisfied by encoding the transition relations in the constraint system along with the previous abstract value, as previously performed in Example 3.3. Similarly, the "Function Application" rule is satisfied by adding the constraints resulting from the summary modulo renaming to the unknowns associated with return nodes (n_r and n_q). Finally, we satisfy the "Summary Coverage" rule using the last two constraints. That entails that the policies associated with the summary are given by the policies associated with the function exit node, which are usually the choice between the base case and the recursive case for a simple recursive function.

We solve the given optimization problem by using policy iteration, which proceeds through the following steps (the order on the elements of $\mathbf{d}$ corresponds to the order given in Equation 5.3, similarly to Example 3.3 the policy is given by a tuple of nodes and $\top, \bot$ elements). For readability, both policies and values are grouped by the node they refer to (the tuple representation can be reconstructed by maintaining the order and ignoring the keys).

1. Initial policy, given by the initiation rule:

$$\mathbf{p} = \{n_c : (\bot), n_j : (\bot, \bot), n_r : (\bot, \bot, \bot), n_q : (\bot), s : (\bot, \bot)\}$$

2. Value determination:

$$\mathbf{d} = \{n_c : (-\infty), n_j : (-\infty, -\infty), n_r : (-\infty, -\infty, -\infty), n_q : (-\infty), s : (-\infty, -\infty)\}$$

3. Policy improvement, the nodes n_r and n_c becomes reachable:

$$\mathbf{p} = \{n_c : (n_e), n_j : (\bot, \bot), n_r : (n_e, n_e, n_e), n_q : (\bot), s : (\bot, \bot)\}$$

4. Value determination:

$$\mathbf{d} = \{n_c : (0), n_j : (-\infty, -\infty), n_r : (0, 0, 0), n_q : (-\infty), s : (-\infty, -\infty)\}$$

5. Policy improvement, we can determine the lower bound on the variable i at the "join" node. Additionally, we get new policies for the summary:

$$\mathbf{p} = \{n_c : (n_e), n_j : (\bot, \bot), n_r : (n_e, n_e, n_e), n_q : (\bot), s : (n_r, n_r)\}$$

6. Value determination, we get finite bounds for the summary:

$$\mathbf{d} = \{n_c : (0), n_j : (\bot, \bot), n_r : (\bot, \bot, \bot), n_q : (\bot), s : (0, 0), \}$$

7. Policy improvement:

$$\mathbf{p} = \{n_c : (n_e), n_j : (n_c, s_{-r}), n_r : (n_e, n_e, n_e), n_q : (\bot), s : (n_r, n_r)\}$$

8. Value determination:

$$\mathbf{d} = \{n_c : (0), n_j : (0, 0), n_r : (0, 0, 0), n_q : (-\infty), s : (0, 0), \}$$

9. Policy improvement, we are finally able to select the policy for n_q:

$$\mathbf{p} = \{n_c : (n_e), n_j : (n_c, s_{-r}), n_r : (n_e, n_e, n_e), n_q : (s), s : (n_r, n_r)\}$$

10. Value determination:

$$\mathbf{d} = \{n_c : (0), n_j : (0, 0), n_r : (0, 0, 0), n_q : (0), s : (0, 0), \}$$

11. The policy can not be improved any further, and the iteration converges.

Let's reconstruct the meaning of the last tuple obtained by value determination. The first value gives the upper bound on $-i$ (or, equivalently the negated lower bound on i) at the node n_c, stating that it is always greater than zero. Similarly, the next two values give (negated) lower bounds on expressions i and t at node n_j, stating that "i and t are always greater than zero". Again, the next three values repeat the same statement about the node n_r, adding that "the value of r is always greater or equal than the value of i. The value associated with n_q simply states that o is always positive. Finally, the last two values give the resulting summary, which states "the output of `sum` is always greater or equal to zero and is always greater or equal to its input".

5.6 Generating Summaries using Intraprocedural Analysis

As before, in order to get an efficient local analysis procedure we want to apply policy iteration to the program directly, without converting it first to a set of semantical equations. To perform that, we develop an algorithm for generating summaries from the results of the intraprocedural analysis. Our algorithm is not specific to policy iteration, and can be parameterized with

any abstract interpretation equipped with strongest postcondition operator and convergence guarantees, in particular the LPI algorithm developed in Chapter 3 .

We say that a summary S_f for a function $f \equiv (\mathit{nodes}, \mathit{edges}, \mathit{calledges}, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex})$ is *generated* by an inductive assertion map $I_f : \mathit{nodes} \to \mathcal{D}$ iff S_f represents the strongest invariant in $\mathcal{D}$ implying the invariant at the return node $I_f(n_r)$, and implied by the invariant at the function start $I_f(n_i)$. For example, for a function $f \equiv (\{a, b\}, \{(a, r = x + 1, b)\}, \emptyset, \{r, x\}, \{x\}, \{r\}, a, b)$ the inductive assertion map $\{a : x \geq 0, b : x \geq 0 \wedge r \geq 1 \wedge r - x = 1\}$ generates the summary $x \geq 0 \wedge r \geq 1 \wedge r - x = 1$.

Effectively, our summaries are given by intraprocedural abstract states associated with the function exit, with variables local to a function projected out. Thus instead of storing summaries explicitly, we use the (projection of) the abstract state associated with the return location of each function during the summary application. Applying this rule leads to Algorithm 5.1, which is essentially the generalization of the Kleene worklist algorithm from background (Algorithm 2.1) to the interprocedural case.

Algorithm Description Algorithm 5.1 can be parameterized with any abstract interpretation which provides a partial order, a strongest postcondition operator, an intersection operator, and a join operator, including LPI, making scalable application of policy iteration possible.

The algorithm operates over two stateful datastructures: a mapping from all program to nodes to associated abstract states I, and a waitlist of nodes Q from which the information was not yet propagated. Initially, we associate each node with an unreachable state $\bot$ (line 9), except for the entry point of the main function, which is set to $\top$ (line 13). During the main fixed point computation, while the waitlist is not empty (line 14) we pop a new node from Q.

Firstly, we process all outgoing intraprocedural edges (line 18): as in usual abstract interpretation, the output state is given by the application of the abstract semantics of the operator. Then, a helper "update" function (line 40) merges the new abstract state with the one previously associated with the new node, and updates the waitlist, unless the new state is subsumed by the already existing one.

Afterwards, we check whether n is the exit node of a function (line 22). In such a case, we use the function application rule, and the new state is given by the intersection of the calling context with output variables of the call projected out, and our state with parameter and return variables renamed. Similarly, the "update" function is called to propagate the information.

Finally, we process all the outgoing call edges (line 30). The new state is derived by projecting the processed state on the parameter variables and subsequent renaming to the called function g input parameters. As before, we call the "update" function to join the new state with the existing one associated with the entry node of g. Furthermore, we update the return site n_{ret} of the calling edge using the existing summary. Observe how we had to apply the function application rule for the second time: that happens because if the update to the called function entry state is subsumed by an existing invariant, we can not rely on the previous "if" statement to perform the application, as the update will not be propagated up to the function exit.

Once the iteration converges, our algorithm returns a global mapping from nodes to abstract states.

Revisiting Running Example Similarly to Example 3.5, we are in the position to revisit the running example from Figure 5.4 with Algorithm 5.1 parameterized with LPI. We use the same template mapping we have previously used in Example 5.1.

Algorithm 5.1 Summary Generation

Input: program $P \equiv (F, f_m)$, abstract domain $\mathcal{D}$,
  partial order $\preceq: \mathcal{D} \to \mathcal{D} \to \mathbb{B}$,
  strongest postcondition operator $[\![.]\!]^\sharp : \mathcal{D} \to \texttt{OPS} \to \mathcal{D} \to \mathcal{D}$,
  join operator $\sqcup : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$,
  intersection operator $\sqcap : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$
Output: map I from nodes of P to $\mathcal{D}$
$\mathit{allnodes} \leftarrow$ all nodes of P
$\mathit{allcalledges} \leftarrow$ all call edges of P
map from nodes to abstract states $I \leftarrow \{n : \bot \mid n \in \mathit{allnodes}\}$
queue of nodes $Q \leftarrow \emptyset$
$m \leftarrow$ starting node of f_m
$I[m] \leftarrow \top$
$Q \leftarrow Q \cup \{m\}$
while $Q \neq \emptyset$ **do**
  $n \leftarrow$ pop from Q
  $s \leftarrow I(n)$
  $f \equiv (\mathit{nodes}, \mathit{edges}, \mathit{calledges}, \mathbf{x}, \mathbf{x}_i, \mathbf{x}_r, n_{en}, n_{ex}) \leftarrow$ function containing n
  for all $(n, \texttt{OP}, n') \in \mathit{edges}$ **do**
    $s' \leftarrow [\![\texttt{OP}]\!]^\sharp(s)$
    UPDATE(n', s')
  end for
  if $n = n_{ex}$ **then**
    ▷ update on summary entails updates on all return sites of calling edges
    for all $c \equiv (f, n_{call}, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{allcalledges}$ **do**
      $(\mathit{gnodes}, \mathit{gedges}, \mathit{gcalledges}, \mathbf{x}^g, \mathbf{x}_i^g, \mathbf{x}_r^g, n_{en}^g, n_{ex}^g) \leftarrow$ function containing c
      $s' \leftarrow s|_{\mathbf{x}_i \cup \mathbf{x}_r}[\mathbf{x}_i/\mathbf{x}_p][\mathbf{x}_r/\mathbf{x}_o] \sqcap I(n_{call})|_{\mathbf{x}^g \setminus \mathbf{x}_o}$
      UPDATE(n_{ret}, s')
    end for
  end if
  for all $c \equiv (g, n, n_{ret}, \mathbf{x}_p, \mathbf{x}_o) \in \mathit{calledges}$ **do**
    ▷ n is a callsite for a function g
    $(\mathit{gnodes}, \mathit{gedges}, \mathit{gcalledges}, \mathbf{x}^g, \mathbf{x}_i^g, \mathbf{x}_r^g, n_{en}^g, n_{ex}^g) \leftarrow g$
    $s' \leftarrow s|_{\mathbf{x}_p}[\mathbf{x}_p/\mathbf{x}_i^g]$
    UPDATE(s', n_{en}^g)
    ▷ use the existing summary
    $s'' \leftarrow I(n_{ex}^g)|_{\mathbf{x}_i^g \cup \mathbf{x}_r^g}[\mathbf{x}_i^g/\mathbf{x}_p][\mathbf{x}_r^g/\mathbf{x}_o] \sqcap s|_{\mathbf{x} \setminus \mathbf{x}_o}$
    UPDATE(s'', n_{ret})
  end for
end while
function UPDATE(node n, state s)
  if $s \not\preceq I(n)$ **then**
    $I(n) \leftarrow I(n) \sqcup s$
    $Q \leftarrow Q \cup \{n\}$
  end if
end function
return I

Note that in the presence of large block encoding (Section 2.9), the abstraction has to be performed at call- and return- nodes: in our example that entails calculating abstraction at each node.

Example 5.2 (Re-Analyzing the Running Example with Interprocedural LPI)**.**

1. We start with an empty state $a_0 \equiv \{\}$ associated with the node n_s.

2. Following the function call $(\mathsf{sum}, n_s, n_q, \emptyset, \{o\})$ produces another top state $a_1 \equiv \{\}$ associated with n_e.

3. From the intraprocedural edge $(n_e, i \leq 0 \wedge r' = 0 \wedge i' = i, n_r)$ we get the state a_2 associated with the node n_r:

$$\tau_1 \equiv i \leq 0 \wedge r' = 0 \wedge i' = 0$$
$$a_2 \equiv \{-i : (0, \tau, a_1), -r : (0, \tau, a_1), -(r - i) : (0, \tau, a_1)\}$$

   Note how we have introduced a helper relation τ_1 to record the policy.

4. By following another intraprocedural edge $(n_e, i > 0 \wedge i' = 0, n_c)$ we get the following state a_3 associated with the node n_c:

$$a_3 \equiv \{-i : (0, i > 0 \wedge i' = 0, a_1)\}$$

5. The function call from n_c is subsumed by existing abstract state $\top$ associated with n_e.

6. We perform the summary application at the node n_c and arrive at the new abstract state a_4 associated with n_j:

$$a_4 \equiv \{-i : (0, i' = i, a_3), -t : (0, t' = r, a_2)\}$$

   Note that the identity assignment $i' = i$ is generated as a policy in order to propagate the bound from the callsite.

7. After traversing the intraprocedural edge $(n_j, r' = i + t, n_r)$ we get the state a_5:

$$\tau_2 \equiv r' = i + t$$
$$a_5 \equiv \{-i : (0, \tau, a_4) - r : (0, \tau, a_4) - (r - i) : (0, \tau, a_4)\}$$

   Again, we have used an auxiliary relation τ_2 to record the used policy. As a_4 is subsumed by a_2 no updates are generated.

8. Finally, we perform the function application for sum at the node n_s. The generated abstract state associated with n_q is:

$$\{-o : (0, o' = r, a_2)\}$$

   No new updates are possible, and the exploration is concluded. The returned invariant is:

$$\{n_s : \top, n_e : \top, n_c : a_3, n_j : a_4, n_r : a_2, n_q : a_5\}$$

Despite the fact that the amount of steps is the same as for Example 5.1, each step is considerably simpler and only involves the local updates, and the exploration progress can be

seen much easier.

The exploration in Example 5.2 is remarkably similar to the one which would happen if the naive `goto` encoding shown in Figure 5.1 was used. The crucial difference is that we only pass the relevant parameters along the call and return edges, and we perform merge instead of intersection at the return site. Applying the `goto` encoding directly to the example in Figure 5.4 would not less us prove that i is always positive at the node n_j, which is crucial for establishing that the return value is always greater or equal than the input in the `sum` summary.

5.7 Algorithm Properties

Property 5.1 (Soundness)**.** The invariant map computed by Algorithm 5.1 satisfies the constraints of Equation 5.2.

Proof. Setting the abstract state associated with the program entry to $\top$ fulfils the "Program Initiation" rule, as the invariant map is updated only through the "update" function, which only enlarges the contained states. The "Consecution" rule is satisfied due to the intraprocedural update in line 18. The updates associated with call edges in line 30 guarantee the "Function Call" rule, and the "Function Application" rule is fulfilled by treating the exit node in line 22. The "Summary Coverage" rule is satisfied implicitly, as our summaries *are* given by the (projection of) the abstract states associated with exit nodes. □

Property 5.2 (Termination)**.** Algorithm 5.1 is guaranteed to terminate if the abstract interpretation parameterization terminates after finitely many join applications (which may include widening) with the growing invariant sequence of abstract states.

Proof. Updates causing the main fixpoint loop to run are only performed in the "update" function (line 40), which only happens when the new abstract state is not subsumed by the old one, and a join is subsequently performed. If such a sequence converges for every node, the algorithm terminates. □

In particular, Property 5.2 entails that LPI parameterization of Algorithm 5.1 terminates in time bounded by the number of policies in Equation 5.1.

Property 5.3 (Optimality)**.** If the parameterization of Algorithm 5.1 does not introduce any imprecision during the join or the abstract postcondition calculation, the obtained invariant is the strongest possible one in the given domain.

Similarly, Property 5.3 entails that LPI terminates with the strongest inductive invariant satisfying the constraints.

Proof Outline. The proof is similar to that for the standard Kleene iteration algorithm: we are simply performing the updates subject to the constraints we have to satisfy. □

5.8 Implementation

We have implemented the parameterization of Algorithm 5.1 with LPI inside the CPAchecker framework. As before, detailed installation and usage instructions are available in Chapter 7.

Our implementation provides a generic framework which can be implemented by any configurable program analysis (Section 2.10) in order to support summary generation. To use

the framework, the client CPA has to implement two methods: one for generating the entry abstract state for the summary (applied in line 34) and one for applying the function summary to the given callsite (applied in line 27 and line 37). The implementation consists of a top-level configurable program analysis, which applies these two functions provided by the wrapped analysis during the successor computation.

Our tool supports programs which modify global variables inside function, or modify the parameter variables by running a pre-analysis CFA-to-CFA transformation which assigns the current value of all modified globals and parameters to a temporary variable, which may then be used inside the summary. Aliasing is currently not supported.

5.9 Extensions

In this section we describe various extensions we have studied to increase the applicability, precision and performance of our algorithm.

5.9.1 Supporting Parameter and Return Expressions

Our program definition (Definition 5.2) only allows variables to be passed as parameters and to be returned. Most C-like languages allow using arbitrary expressions for both, which may even include function calls. Using the "Function Application" and "Function Call" rules from Equation 5.2 becomes problematic as they require renaming operations which semantics is not clear in the presence of such expressions.

Introducing temporary variables to hold the values of expressions the function is called with, and expressions the function is returning solves the issue (e.g. replacing `return a + b + c` with `tmp = a + b + c; return tmp`). Introducing auxiliary variables also allows our analysis to support complex expressions involving function calls (e.g. `a = b + f(c)` is replaced with `tmp = f(c); a = b + tmp`).

5.9.2 Supporting Globals using Pre-Analysis

The program model we have described in Section 5.3 does not support global variables. This simplifies the summarization procedure, as each function explicitly returns all the variables it modifies. Globals can be easily removed using a syntactic transformation, which causes all functions to accept and return all declared global variables. Yet for programs extensively using globals that would not be very different than the naive goto encoding, due to the joins arising from spurious paths.

Instead, we run a pre-analysis which finds an over-approximation of all variables which can be affected by the function (or the functions called by it), and we extend the "output variables" set $\mathbf{x}_o$ associated with the call-edge with those.

5.9.3 Abstract Reachability Tree Generation

Recall the technique we have described in Section 4.4.1 for generating an abstract reachability tree from the abstract interpretation run. The technique is still applicable in the presence of summaries, yet the produced graph is no longer a tree, but a directed acyclic graph due to a fact that the return-site has two predecessors: summary and a callsite. Acyclic counterexample traces can be still found in a resulting DAG, and the domain can be refined using e.g. tree interpolation [HHP10]. Additionally, the procedure for finding "neighbour states" (Algorithm 4.2,

line 36) has to be updated to perform a breadth-first-search instead of traversing a chain of backpointers). Thus the interpolation-based template synthesis described in Section 4.4 is still applicable for the approach described in this chapter, yet it was not implemented.

5.9.4 Generating Multiple Summaries

As shown by the benchmarks we have looked at, disjunctive summarize are often needed to verify the properties of interest. Such summaries can be derived from convex abstract states by generating a number of different convex summaries, yet our description of Algorithm 5.1 only generates a single summary per each function.

Yet Algorithm 5.1 can be trivially modified to support multiple summaries. Let $\mathcal{D}'$ be a finite partitioning of the abstract domain $\mathcal{D}$, where we wish to generate a separate summary for each element of $\mathcal{D}'$ (in general, $\mathcal{D}'$ does not have to be finite, yet the computation is not guaranteed to converge if this is not the case).

In order to generate multiple summaries with such a partitioning, we require the analysis to *provide* a function $\mathit{partition} : \mathcal{D} \to \mathcal{D}'$, and we change the main stateful datastructure $I : \mathit{nodes} \to \mathcal{D}$ (line 9) storing the global inductive assertion map to $I : \mathit{nodes} \to \mathcal{D}' \to \mathcal{D}$, where the second argument is the calling context of the function (always $\top$ for the main function f_m).

When applying the "Entry" rule for creating a summary in line 34 we update the entry node in the corresponding partition element, and likewise, when applying the summary application rule in lines 27, 37 we apply the summary from the partition corresponding to the callsite.

In the extreme case where the $\mathcal{D}' = \mathcal{D}$ and *partition* is an identity function the precision of the resultant algorithm is equal to that of inlining, yet it is not guaranteed to converge in the presence of recursion.

Due to a lack of time, this extension was not implemented.

5.9.5 Large Block Encoding Support and Inlinement

Recall that LPI operates over CFA encoded using large block encoding (Section 2.9), which can get both higher precision and performance by reducing the number of abstraction points.

In order to support this encoding, *call-node* and *return-node* for each summarized procedure are added to the cut-set, which is sufficient for breaking all (interprocedural) cycles, potentially caused by recursion.

However, extra abstraction can negatively affect precision and performance. Thus, like a compiler, for each processed function we can apply a heuristic deciding whether it should be inlined or summarized. Due to a lack of time, dynamic inlinement was not implemented.

5.10 Evaluation

We evaluate our implementation on programs in the "Recursive" category of the International Competition on Software Verification [Bey16]. All benchmarks were run using the following resource bounds: Intel Xeon E5-2650 v2 @ 2.60 GHz, and a limit of 10 GB RAM and 100 s CPU time per program.

The category contains 98 verification tasks (each task includes a program, property and a verification verdict), and in 53 of those the expected verdict is "true". Our implementation was able to verify 24 of those, with no incorrect verdicts produced. We present the quantile plot showing the tool performance in Figure 5.5.

For comparison, state of the art verification tool SeaHorn [GKN15] using Spacer [Kom+13] algorithm achieves a better result with 49 benchmarks verified, mostly due to the ability to generate non-convex summaries with predicates forming the abstract domain discovered dynamically using interpolation. We believe this is partly caused by a dataset which does not contain complex convex properties (e.g. proving lack of overflows), and our approach could be very useful in such cases.

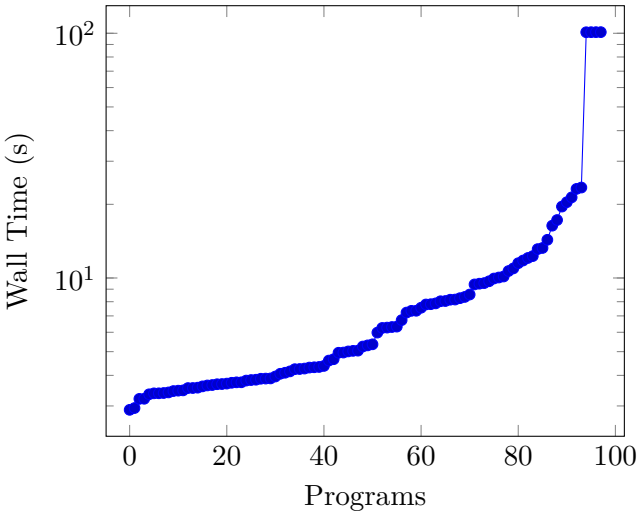


Figure 5.5: Quantile timing plots showing the performance of the summary generation algorithm parameterized by LPI. As before, each data point corresponds to a processed verification task, with y coordinate given by the time taken to analyze the task, and x coordinate given by the program number, with series sorted by time.

5.11 Conclusion and Future Work

We have developed an interprocedural analysis algorithm which can be parameterized by any abstract interpretation. We have studied this algorithm in context of LPI parameterization, as it guarantees finding the least fixed point for the fixed number of summaries. The proof-of-concept implementation was provided, and we have shown how it can be used to verify many recursive benchmarks.

Our work is not directly applicable to the min-policy [Cos+05] approach, as greatest fixed points are notoriously over-approximating for recursive procedures.

5.11.1 Future Work

While we have obtained interesting results, engineering work still needs to be done in order for our algorithm to be applicable in practice. Firstly, the extensions described in Section 5.9.5 and Section 5.9.4 were not implemented. Additionally, the problems of supporting function pointers and aliasing in general have to be addressed.

Aliasing Support Unlike a bottom-up approach for summary generation, our algorithm has an advantage of knowing the calling context during the summary generation, which can already include aliasing information. This advantage can be used by e.g. deciding to generate a new summary whenever a calling context has different aliasing, obtaining greater precision.

Supporting Function Pointers Function pointers can be supported in a usual way: namely, either using a pre-analysis tracking which function each pointer can be aliased to, and then trating the resulting call as a non-deterministic choice between different possibly aliased functions, or even tracking the aliasing directly in the analysis parameterization. Note that in a case of function pointers the analysis finding modified and read global variables (Section 5.9.2) would need a pre-analysis itself in order to resolve function pointers.

Chapter 6

Formula Slicing: Inductive Invariants from Preconditions

6.1 Introduction

In this chapter we present new method for invariant synthesis which we call "formula slicing". The findings were already published [KM16], and the chapter largely follows the publication with some new results added.

Abstract-interpretation-based approaches restrict the class of expressible inductive invariants to a predefined abstract domain, such as products of intervals, octagons, or convex polyhedra, all of which can only express convex properties. Any candidate invariants which can not be expressed in the chosen abstract domain get over-approximated. This is a severe restriction: if a property flows from the beginning of the program to a loop head, and holds inductively after, but is not representable within the previously chosen abstract domain, it is discarded. In contrast, our idea exploits the insight that many loops in the program affect only a small part of the memory, and many invariants which were valid before the loop are still valid afterwards.

```
1  int x = input();
2  int p = input();
3  if (p) {
4      assume(x >= 0);
5  } else {
6      assume(x < 0);
7  }
8  for (int i=0; i < input(); i++) {
9      x *= 2;
10 }
```

FIGURE 6.1: Motivating Example for Finding Inductive Weakenings

Consider finding an inductive invariant for the motivating example in Figure 6.1. Symbolic execution up to the loop-head can precisely express all reachable states at the loop entry:

$$i = 0 \wedge (p \neq 0 \implies x \geq 0) \wedge (p = 0 \implies x < 0) \tag{6.1}$$

Yet abstraction in a numeric convex domain at the loop head yields $i = 0$, completely losing the information that x is positive iff $p \neq 0$. Observe that this information loss is not *necessary*, as the sign of x stays invariant under the multiplication by a positive constant (assuming mathematical integers for the simplicity of exposition). To avoid this loss of precision, we develop a "formula slicing" algorithm which computes inductive *weakenings* of propagated

formulas, allowing to propagate the formulas representing inductive invariants *across* loop heads. In the motivating example, formula slicing computes an inductive weakening of the initial condition in Equation 6.1), which is $(p \neq 0 \Rightarrow x \geq 0) \wedge (p = 0 \Rightarrow x < 0)$, and is thus true at every iteration of the loop. The computation of inductive weakenings is performed by iteratively filtering out conjuncts falsified by *counterexamples-to-induction*, derived using an SMT solver. In the example, transition $i = 1$ from $i = 0$ falsifies the constraint $i = 0$, and the rest of the conjuncts are inductive.

The formula slicing fixpoint computation algorithm is based on performing abstract interpretation on the lattice of conjunctions over a finite set of predicates. The computation starts with a seed invariant which *necessarily* holds at the given location on the first time the control reaches it, and during the computation it is iteratively weakened until inductiveness.

6.1.1 Contributions

We present a novel insight for generating inductive invariants, and a method for creating a lattice of weakenings from an arbitrary formula describing the loop precondition using a *relaxed conjunctive normal form* (Definition 6.1) and best-effort quantifier elimination (Section 6.2.1).

We evaluate (Section 6.6) our implementation of the formula slicing algorithm on the "Device Drivers" benchmarks from the International Competition on Software Verification [Bey16], and we demonstrate that it can successfully verify large, real-world programs which can not be handled with traditional numeric abstract interpretation, and that it is competitive with state of the art techniques.

6.1.2 Related Work

The *Houdini* [FL01] algorithm mines the program for a set of predicates, and then finds the largest inductive subset, dropping the candidate non-inductive lemmas until the overall inductiveness is achieved. The optimality proof for *Houdini* is present in the companion paper [FJL01]. A very similar algorithm is used by Bradley et al. [BM07] to generate the inductive invariants from negations of the counter-examples to induction.

Inductive weakening based on counterexamples-to-induction can be seen as an algorithm for performing predicate abstraction [GS97]. Generalizing inductive weakening to *best abstract postcondition computation* Reps et al. [RSY04] use the weakening approach for computing the best abstract transformer for any finite-height domain, which we also perform in Section 6.1.4.

Generating inductive invariants from a number of heuristically generated lemmas is a recurrent theme in the verification field. In *automatic abstraction* [Kom+13] a set of predicates is found for the simplified program with a capped number of loop iterations, and is filtered until the remaining invariants are inductive for the original, unmodified program. A similar approach is used for synthesizing bit-precise invariants by Gurfinkel et al. [GBM14].

The complexity of the inductive weakening and that of the related template abstraction problem are analyzed by Lahiri and Qadeer [LQ09].

6.1.3 Counterexample-to-Induction Weakening Algorithm

The approaches [BM07; FL01; GBM14; Kom+13] mentioned in Section 6.1.2 are all based on using counterexamples to induction for filtering the input set of candidate lemmas. For completeness, we restate this approach in Algorithm 6.1.

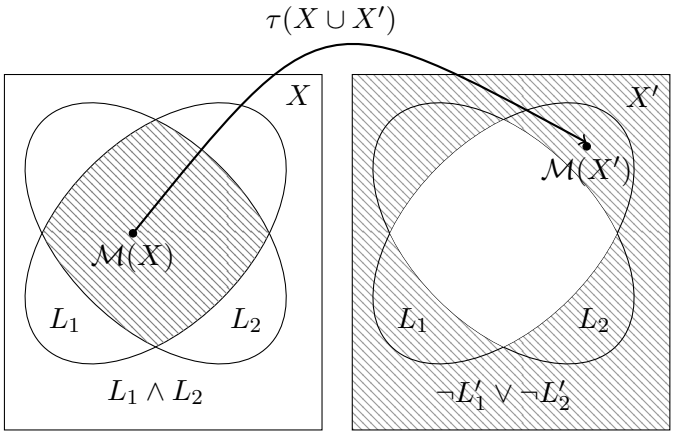


FIGURE 6.2: Formula $\phi(\mathbf{x}) \equiv L_1(\mathbf{x}) \wedge L_2(\mathbf{x})$ is tested for inductiveness under $\tau(\mathbf{x} \cup \mathbf{x}')$. Model $\mathcal{M}$ identifies a counter-example to induction. From $\mathcal{M} \models \neg L'_2(\mathbf{x}')$ we know that the lemma L_2 has to be dropped. As weakening progresses, the shaded region in the left box is growing, while the shaded region in the right box is shrinking, until there are no more counterexamples to induction.

In order to perform the weakening without syntactically modifying ϕ during the intermediate queries, we perform *selector variables* annotation: we replace each lemma $l_i \in \phi$ with a disjunction $s_i \vee l_i$, using a fresh boolean variable s_i. Observe that if all selector variables are assumed to be false the annotated formula $\phi_{\text{annotated}}$ is equivalent to ϕ, and that assuming any individual selector s_i is equivalent to removing (replacing with $\top$) the corresponding lemma l_i from ϕ. Such an annotation allows us to make use of *incrementality* support by SMT solvers, by using the *solving with assumptions* feature.

Algorithm 6.1 iteratively checks input formula ϕ for inductiveness using Equation 2.3 (line 14). The solver will either report that the constraint is unsatisfiable, in which case ϕ is inductive, or provide a counterexample-to-induction represented by a model $\mathcal{M}(\mathbf{x} \cup \mathbf{x}')$ (line 15). The counterexample-driven algorithm uses $\mathcal{M}$ to find the set of lemmas which should be removed from ϕ, by removing the lemmas modelled by $\mathcal{M}$ in $\neg\phi'$ (line 21). The visualization of such a filtering step for a formula ϕ consisting of two lemmas is given in Figure 6.2.

Algorithm 6.1 terminates with the *strongest* possible weakening [FJL01] within the linear number of SMT calls with respect to $\|\phi_{\text{annotated}}\|$.

6.1.4 From Weakenings to Abstract Postconditions

As shown by Reps et al. [RSY04], the inductive weakening algorithm can be generalized for the abstract postcondition computation for any finite-height lattice.

For given formulas $\psi(\mathbf{x})$, $\tau(\mathbf{x} \cup \mathbf{x}')$, and $\phi(\mathbf{x})$ consider the problem of finding a weakening $\hat{\phi} \subseteq \phi$, such that all feasible transitions from ψ through τ end up in $\hat{\phi}$. This is an abstract postcondition of ψ under τ in the lattice of all weakenings of ϕ (Section 2.8.6). The problem of finding it is very similar to the problem of finding an inductive weakening, as we can check whether a given weakening of ϕ is a postcondition of ψ under τ using Equation 6.2,

$$\psi(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}') \wedge \neg\phi'_{\text{annotated}}(\mathbf{x}') \tag{6.2}$$

Algorithm 6.1 can be adapted for finding the *strongest* postcondition in the abstract domain of weakenings of the input formula with very minor modifications. The required changes are accepting an extra parameter ψ, and changing the queried constraint (line 6) to Equation 6.2. The found postcondition is indeed strongest [RSY04].

Algorithm 6.1 Counterexample-Driven Weakening.

Input: Formula $\phi(\mathbf{x})$ to weaken in RCNF, transition relation $\tau(\mathbf{x} \cup \mathbf{x}')$
Output: Inductive $\hat{\phi} \subseteq \phi$
▷ Annotate lemmas with selectors, S is a mapping from selectors to lemmas they annotate.
$S, \phi_{\text{annotated}} \leftarrow$ ANNOTATE(ϕ)
context $\leftarrow$ new context of SMT solver
query $\leftarrow \phi_{\text{annotated}} \wedge \tau \wedge \neg\phi'_{\text{annotated}}$
Assert *query* in *context*
assumptions $\leftarrow \emptyset$
removed $\leftarrow \emptyset$
▷ In the beginning, all of the lemmas are present
for all $(selector, lemma) \in S$ **do**
    assumptions $\leftarrow$ *assumptions* $\cup \{\neg selector\}$
end for
while *context* is satisfiable with *assumptions* **do**
    $\mathcal{M} \leftarrow$ model of *context*
    assumptions $\leftarrow \emptyset$
    for all $(selector, lemma) \in S$ **do**
        if $\mathcal{M} \models \neg lemma'$ or $lemma'$ is *irrelevant* to satisfiability **then**
            ▷ *lemma* has to be removed.
            assumptions $\leftarrow$ *assumptions* $\cup \{selector\}$
            removed $\leftarrow$ *removed* $\cup \{lemma\}$
        else
            assumptions $\leftarrow$ *assumptions* $\cup \{\neg selector\}$
        end if
    end for
end while
▷ Remove all lemmas which were filtered out
return $\phi[removed/\top]$

This adaptation effectively runs a modified version of cartesian [BPR03] *predicate abstraction* [GS97]. Unlike a classical approach where the negation of each predicate is tested for unsatisfiability (like in [BPR03]), we test the negation of the disjunction over all the predicates, and we used the model to filter out multiple predicates at once, thereby speeding up the convergence.

6.2 The Space of All Possible Weakenings

We wish to find a *weakening* of a set of states represented by $\phi(\mathbf{x})$, such that it is inductive under a given transition $\tau(\mathbf{x} \cup \mathbf{x}')$. For a single-node CFA defined by an initial condition ϕ and a loop transition τ such a weakening would constitute an *inductive invariant* as by definition of weakening it satisfies the initial condition and is inductive.

We start with an observation that for a formula in NNF replacing any subset of literals with $\top$ results in an over-approximation, as both conjunction and disjunction are monotone operators. E.g. for a formula $\phi \equiv (l_a \wedge l_b) \vee l_c$ such possible weakenings are $\top$, $l_b \vee l_c$, and $l_a \vee l_c$.

The set of weakenings defined in the previous paragraph is redundant, as it does not take the formula structure into account — e.g. in the given example if l_c is replaced with $\top$ it is irrelevant

what other literals are replaced, as the entire formula simplifies to $\top$. The most obvious way to address this redundancy is to convert ϕ to CNF and to define the set of all possible weakenings as conjunctions over the subsets of clauses in ϕ_{CNF}. E.g. for the formula $\phi \equiv l_a \wedge l_b \wedge l_c$ possible weakenings are $l_a \wedge l_b$, $l_b \wedge l_c$, and $l_a \wedge l_c$. This method is appealing due to the fact that for a set of lemmas the *strongest* (implying all other possible inductive weakenings) inductive subset can be found using a linear number of SMT checks [BM07]. However (Section 2.3) polynomial-sized CNF conversion (e.g. Tseitin encoding) requires introducing existentially quantified boolean variables which make inductiveness checking Π_2^p-hard.

The arising complexity of finding inductive weakenings is inherent to the problem: in fact, the problem of finding *any* non-trivial ($\neq \top$) weakening within the search space described above is Σ_2^p-hard (see proof in Section 6.7).

Thus instead we use an over-approximating set of weakenings, defined by all possible subsets of lemmas present in ϕ after the conversion to *relaxed conjunctive normal form.*

Definition 6.1 (Relaxed Conjunctive Normal Form (RCNF))**.** A formula $\phi(\mathbf{x})$ is in *relaxed conjunctive normal form* if it is a conjunction of quantifier-free formulas (lemmas).

For example, the formula $\phi \equiv l_a \wedge (l_b \vee (l_c \wedge l_d))$ is in RCNF. The over-approximation comes from the fact that non-atomic parts of the formula are grouped together: the only possible non-trivial weakenings for ϕ are l_a and $l_b \vee (l_c \wedge l_d)$, and it is impossible to express $l_a \wedge (l_b \vee l_c)$ within the defined search space.

We may abuse the notation by treating ϕ in RCNF as a set of its conjuncts, and writing $l \in \phi$ for a lemma l which is an argument of the parent conjunction of ϕ, or $\phi_1 \subseteq \phi_2$ to indicate that all lemmas in ϕ_1 are contained in ϕ_2, or $\|\phi\|$ for the number of lemmas in ϕ. For ϕ in RCNF we define a set of all possible *weakenings* as conjunctions over all sets of lemmas contained in ϕ. We use an existing, optimal counter-example based algorithm in order to find the *strongest* weakening of ϕ with respect to τ in the next section.

A trivially correct conversion to a relaxed conjunctive normal is to convert an input formula ϕ to a conjunction $\bigwedge\{\phi\}$. However, this conversion is not very interesting, as it gives rise to a very small set of weakenings: ϕ and $\top$. Consequently, with such a conversion, if ϕ is not inductive with respect to the transition of interest, no non-trivial weakening can be found. On the other extreme, ϕ can be converted to CNF explicitly using associativity and distributivity laws, giving rise to a very large set of possible weakenings. However, the output of such a conversion is exponentially large.

We present an algorithm which convert ϕ into a polynomially-sized conjunction of lemmas. Our conversion algorithm applies the following rules recursively until a fixpoint is reached:

- *Flattening:* all nested conjunctions are flattened. E.g. $a \wedge (b \wedge c)$ is converted to $a \wedge b \wedge c$.
- *Factorization:* when processing a disjunction over multiple conjunctions we find and extract a common factor. E.g. $(a \wedge b) \vee (b \wedge c)$ is converted to $b \wedge (a \vee c)$
- *Explicit expansion with size limit:* a disjunction $\bigvee L$, where each $l \in L$ is a conjunction, is converted to a conjunction over disjunctions over all elements in the cross product over L. E.g. $(a \wedge b) \vee (c \wedge d)$ is rewritten to $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$.

  Applying such an expansion results in an exponential blow-up, but we only perform it if the resulting formula size is smaller than a fixed constant, and we limit the expansion depth to one.

6.2.1 Eliminating Existentially Quantified Variables

The formulas resulting form large block encoding (Section 2.9) may have intermediate (neither input nor output), existentially bound variables. In general, existential quantifier elimination (with e.g. Fourier-Motzkin) is exponential. However, for many cases such as simple deterministic assignments, existential quantifier elimination is easy: e.g. $\exists t.\, x' = t + 3 \wedge t = x + 2$ can be trivially replaced by $x' = x + 5$ using substitution.

We use a two-step method to remove the quantified variables: we run a best-effort pattern-matching approach, removing the bound variables which can be eliminated in polynomial time, and in the second step we drop all the lemmas which still contain the existentially bound variables. The resulting formula is an over-approximation of the original one.

6.3 Formula Slicing: Overall Algorithm

We develop the *formula slicing* algorithm in order to apply the inductive weakening approach for generating inductive invariants in large, potentially non-reducible programs with nested loops.

"Classical" Houdini-based algorithms consist of two steps: *candidate* lemmas generation, followed by counterexample-to-induction-based filtering. However, in our case candidate lemmas representing postconditions depend on previous filtering steps, and careful consideration is required in order to generate *unique* candidate lemmas which do not depend on the chosen iteration order.

6.3.1 Abstract Reachability Tree

In order to solve this problem we use an algorithm for abstract reachability tree generation, as given in Section 4.4.1.

The transfer relation for the formula slicing is given in Algorithm 6.2. In order to generate a successor for an element (n_a, d, b), and an edge (n_a, τ, n_b) we first traverse the chain of backpointers up the tree. If we can find a "neighbour" element s where $s|_1 = n_a$[1] by following the backpointers, we weaken s until inductiveness (line 4) relative to the new incoming transition τ, and return that as a postcondition. Such an operation effectively performs widening [CC77a] to enforce convergence. If no such neighbour exists, we convert $\exists \mathbf{x}.\, [\![d]\!]\,(\mathbf{x}) \wedge \tau(\mathbf{x} \cup \mathbf{x}')$ to RCNF form (line 6), and this becomes a new element of the abstract domain.

Observe that our approach for generating initial candidate invariants ensures monotonicity and reproducibility, even in the case of a non-reducible CFA. As a downside, tree representation may lead to the exponential state-space explosion (as a single node in a CFA may correspond to many nodes in an ART). However, from our experience in the evaluation (Section 6.6), with a good iteration order (stabilizing inner components first [Bou93]) this problem does not occur in practice.

[1] In the implementation, the *neighbour* is defined by a combination of a callstack, a CFA node and a loopstack.

Algorithm 6.2 Formula Slicing: Postcondition Computation.

function POST(edge $e \equiv (n_a, \tau, n_b)$, state $t \equiv (n_a, d, b)$)
  neighbour $s \leftarrow$ FINDNEIGHBOUR(b, n_0)
  if $s \neq \emptyset$ **then**
    ▷ Abstract postcondition of d under τ in weakenings of s (Section 6.1.4).
    $e \leftarrow$ WEAKEN$(d, \tau \wedge n_b, s)$
  else
    ▷ Convert the current invariant candidate to RCNF.
    $e \leftarrow$ TORCNF$([\![d]\!] \wedge \tau)$
  end if
  return (n_b, e, t)
end function

6.3.2 Example Formula Slicing Run

Consider running formula slicing on the program in Figure 6.3, which contains two nested loops. The corresponding edge encoding is given in Equation 6.3:

$$\begin{aligned}
\tau_1 \equiv& x' = 0 \wedge y' = 0 \wedge (p' = 1 \wedge s' \vee p' = 2 \wedge \neg s') \\
\tau_2 \equiv& x' = x + 1 \wedge c' = 100 \\
\tau_3 \equiv& (p \neq 1 \wedge p \neq 2 \implies c' = 0) \wedge y' = y + 1 \wedge p' = p \\
\tau_4 \equiv& x' = x \wedge y' = y \wedge p' = p \wedge c' = c
\end{aligned} \tag{6.3}$$

```
1  int p, c, s=input(), x = 0, y = 0;
2  p = s ? 1 : 2;
3  while (input()) { // A
4      x++;
5      c = 100;
6      while (input()) { // B
7          if (p != 1 && p != 2) {
8              c = 0;
9          }
10         y++;
11     }
12     assert(c == 100);
13 }
14 assert((s && p == 1) || (!s && p == 2));
```

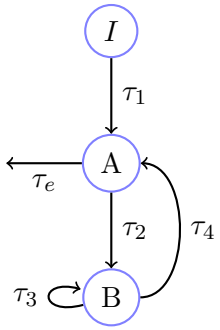


FIGURE 6.3: Example Program with Nested Loops: Listing and CFA.

Similarly to Equation 2.3, we can check candidate invariants $A(\mathbf{x}), B(\mathbf{x})$ for inductiveness by posing an SMT query shown in Equation 6.4. The constraint in Equation 6.4 is unsatisfiable iff $\{A : A(\mathbf{x}), B : B(\mathbf{x})\}$ is an inductive invariant (Section 2.5.3).

$$\exists \mathbf{x} \cup \mathbf{x}'. \bigvee \begin{aligned}
&\tau_1(\mathbf{x}') \wedge \neg A(\mathbf{x}') \\
&A(\mathbf{x}) \wedge \tau_2(\mathbf{x} \cup \mathbf{x}') \wedge \neg B(\mathbf{x}') \\
&B(\mathbf{x}) \wedge \tau_3(\mathbf{x} \cup \mathbf{x}') \wedge \neg B(\mathbf{x}') \\
&B(\mathbf{x}) \wedge \tau_4(\mathbf{x} \cup \mathbf{x}') \wedge \neg A(\mathbf{x}')
\end{aligned} \tag{6.4}$$

Equation 6.4 is unsatisfiable iff *all* of the disjunction arguments are unsatisfiable, and hence the checking can be split into multiple steps, one per analyzed edge. Each postcondition computation (Algorithm 6.2) either generates an initial seed invariant candidate, or picks one argument of Equation 6.4, and weakens the right hand side until the constraint becomes unsatisfiable. Run of the formula slicing algorithm on the example is given below:

- Traversing τ_1, we get the initial candidate invariant:

$$I(A) \leftarrow \bigwedge \{x = 0, y = 0, s \implies p = 1, \neg s \implies p = 2\}$$

- Traversing τ_2, the candidate invariant for B becomes:

$$I(B) \leftarrow \bigwedge \{x = 1, y = 0, s \implies p = 1, \neg s \implies p = 2, c = 100\}$$

- After traversing τ_3, we weaken the candidate invariant $I(B)$ by dropping the lemma $y = 0$ which gives rise to the counterexample to induction (y gets incremented). The result is:

$$I(B) \leftarrow \bigwedge \{x = 1, s \implies p = 1, \neg s \implies p = 2, c = 100\}$$

  which is inductive under τ_3.

- The edge τ_4 is an identity, and the postcondition computation results in lemmas $x = 0$ and $y = 0$ dropped from $I(A)$, resulting in:

$$I(A) \leftarrow \bigwedge \{y = 0, s \implies p = 1, \neg s \implies p = 2\}$$

- After traversing τ_2, we obtain the weakening of $I(A)$ by dropping the lemma $x = 1$ from $I(B)$, resulting in:

$$I(B) \leftarrow \bigwedge \{s \implies p = 1, \neg s \implies p = 2, c = 100\}$$

- Finally, the iteration converges, as all further postconditions are already covered by existing invariant candidates. Observe that the computed invariant is sufficient for proving the asserted property.

6.4 Extensions

Syntactic Weakening Algorithm A syntactic-based approach is possible as a faster and less precise alternative which does not require SMT queries. For an input formula $\phi(\mathbf{x})$ in RCNF, and a transition $\tau(\mathbf{x} \cup \mathbf{x}')$, syntactic weakening returns a subset of lemmas in ϕ, which are not *syntactically modified* by τ: that is, none of the variables are modified or have their address taken. For example, the lemma $x > 0$ is not syntactically modified by the transition $y' = y + 1 \wedge x \geq 1$, but it is modified by $x' = x + 1$.

Non-Nested Loop Handling When performing the inductive weakening (Algorithm 6.2, line 4) on the edge (N, τ, N) we annotate and weaken the candidate invariants on both sides (without modifications described in Section 6.1.4), and we cache the fact that the resulting weakening is inductive under τ.

Liveness-Based Filtering We precompute live variables, and the candidate lemmas (Algorithm 6.2, line 6) which only contain dead variables are discarded.

Extending the Set of Weakenings In a sense, every inductive invariant which could be possibly found by a static analyzer is a *weakening* of the initial condition. Our set of considered weakenings is relatively small, as it only includes the weakenings obtained by dropping lemmas from the RCNF form. This set can be extended using the following approaches:

- Replacing the assignment $a = b$ in formulas with a set of constraints $a \leq b \wedge a \geq b$. This gives us a richer set of weakenings, as each assignment may be weakened to an inequality (as compared to the previous approach).

6.5 Implementation

We have developed the Slicer tool, which runs the formula slicing algorithm on an input C program. Slicer performs inductive weakenings using the Z3 [MB08] SMT solver, and best-effort quantifier elimination using the `qe-light` Z3 tactic. Our tool can analyze a verification task by finding an inductive invariant and reporting `true` if the found invariant *separates* the initial state from the error property, and `unknown` otherwise. Additional usage details are described further in Chapter 7.

6.6 Experiments and Evaluation

We have evaluated the formula slicing algorithm on the "Device Drivers" category from the International Competition on Software Verification (SV-COMP) [Bey16]. The dataset consists of 2120 verification tasks, of which 1857 are designated as *correct* (the error property is unreachable), and the rest admit a counter-example. All the experiments were performed on Intel Xeon E5-2650 with 2.6 GHz, and limits of 8 GB RAM, 2 cores, and 600 seconds CPU time per program. We compare the following three approaches:

- Slicer-CEX (rev `21098`): formula slicing using counterexample-based weakening (Section 6.1.3).
- Slicer-Syntactic (rev `21098`): formula slicing using syntactic weakening (Section 6.4).
- Predicate Analysis: predicate abstraction with interpolants [McM06], as implemented inside CPAchecker. We have chosen this approach for comparison as it represents state-of-the-art in model checking, and was found especially suitable for analyzing device drivers.
- PAGAI [HMM12] (git hash `e44910`): abstract interpretation-based tool, which implements the path focusing [MG11] approach.

In Table 6.1 we show overall precision and performance of the four compared approaches. As formula slicing is inherently over-approximating, it is not capable of finding counterexamples to safety, and we only compare the number of produced safety proofs.

From the data in the table we can see that predicate analysis produces the most correct proofs. This is expected since it can generate new predicates, and it is *driven* by the target

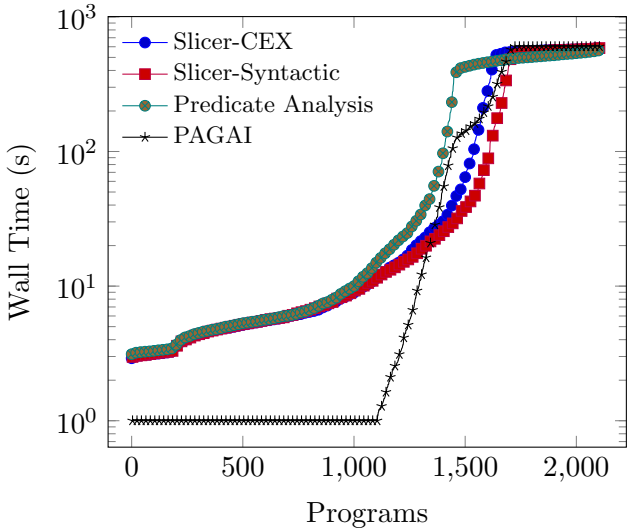


FIGURE 6.4: Quantile plot showing performance of the compared approaches. Shows analysis time for each benchmark, where the data series are sorted by time separately for each tool. For readability, the dot is drawn for every 20^{th} program, and the time is rounded up to one second.

Tool	# proofs	# incorrect	# timeouts	# memory outs
Slicer-CEX	1253	0	475	0
Slicer-Syntactic	1166	0	407	0
Predicate Analysis	1301	0	657	0
PAGAI	1214	3	409	240

TABLE 6.1: Evaluation results. The "# incorrect" column shows the number of safety proofs the tool has produced where the analyzed program admitted a counterexample.

property. However, formula slicing and abstract interpretation have much less timeouts, and they do not require target property annotation, making them more suitable for use in domains where a single error property is not available (advanced compiler optimizations, multi-property verification, and boosting another analysis by providing an inductive invariant). The programs verified by different approaches are also different, and formula slicing verifies 22 programs predicate analysis could not.

The performance of the four analyzed approaches is shown in the quantile plot in Figure 6.4. The plot shows that predicate analysis is considerably more time consuming than other analyzed approaches. Initially, PAGAI is much faster than other tools, but around 15 seconds it gets overtaken by both slicing approaches. Though the graph seems to indicate that PAGAI overtakes slicing again around 100 seconds, in fact the bend is due to out of memory errors. The flattening around 900 seconds for all tools corresponds to the grace period before the hard timeout.

The quantile plot also shows that the time taken to perform inductive weakening does not dominate the overall analysis time for formula slicing. This can be seen from the small timing difference between the syntactic and counterexample-based approaches, as the syntactic approach does not require querying the SMT solver for weakening.

Finally, we present data on the number of SMT calls required for computing inductive weakenings in Figure 6.5. The distribution shows that the overwhelming majority of weakenings can be found within just a few SMT queries.

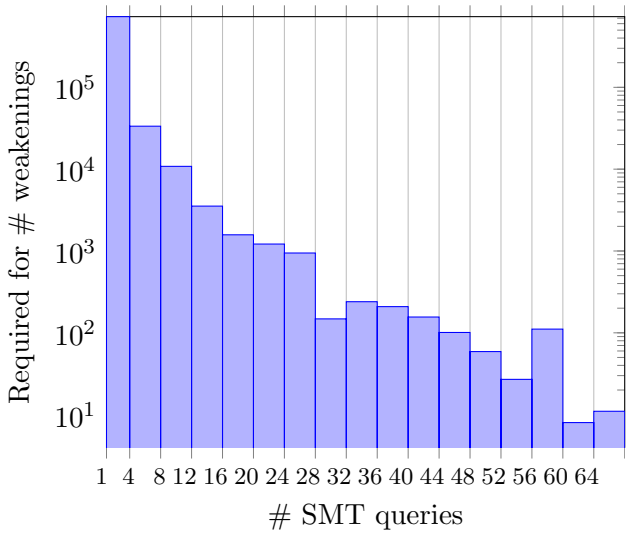


FIGURE 6.5: Distribution of the number of iterations of inductive weakening (Section 6.1.3) required for convergence across all benchmarks. Horizontal axis represents the number of SMT calls required for convergence of each weakening, and vertical axis represents the count of the number of such weakenings.

6.7 Complexity of Finding a Non-Trivial Inductive Weakening Over Literals

As we have mentioned in Section 6.2, a more expressive space of weakenings over formulas is to consider replacing any subset of literals with $\top$ after a NNF conversion. In this section we show that it leads to a number of undesirable properties, including the absence of *strongest* inductive weakening (Example 6.1), and Σ_2^p complexity for finding any non-trivial inductive weakening (Theorem 6.1).

Example 6.1 (No Strongest Inductive Weakening)**.** Consider a program over four Boolean variables a, b, c, d and the transition relation $\tau \equiv a \wedge b \wedge c \wedge d \wedge \neg a' \wedge b' \wedge \neg c' \wedge d'$ (the only possible transition is from $a \wedge b \wedge c \wedge d$ to $\neg a \wedge b \wedge \neg c \wedge d$). Consider finding the weakening of $\phi \equiv (a \wedge b) \vee (c \wedge d)$, Both the $\{a\}$-weakening $(b \vee (c \wedge d))$ and the $\{c\}$-weakening $((a \wedge b) \vee d)$ are inductive, but their intersection $(a \wedge b) \vee (b \wedge d) \vee (c \wedge d)$ (obviously inductive) is not a weakening of ϕ and there is no inductive weakening stronger than either of these.

Theorem 6.1 (Σ_2^p-completeness)**.** The problem of deciding, given quantifier-free SMT formulas $\phi(\mathbf{x})$ and $\tau(\mathbf{x} \cup \mathbf{x}')$, whether there exists a non-trivial ($\not\equiv \top$) weakening of ϕ that is inductive with respect to τ is Σ_2^p-complete.

Belonging to Σ_2^p. Let S be some subset of literals of ϕ. Let $\hat{\phi}$ be the weakening of ϕ where all literals in S are replaced with $\top$. Checking that $\hat{\phi}$ is inductive with respect to τ is in co-NP, therefore the problem of finding a non-trivial $\hat{\phi}$ is in Σ_2^p □

We show completeness by constructing from an arbitrary closed $\exists^*\forall^*$ formula ψ a loop τ and a precondition I such that the existence of a non-trivial ($\not\equiv \top$) weakening of the precondition is equivalent to the truth of ψ. Without loss of generality, let ψ have m Boolean variables $x_0, \ldots, x_{m-1}$ bound by the existential quantifier and n Boolean variables $y_0, \ldots, y_{n-1}$ bound by

```
bitvector x = ⊥;
boolean o = ⊥;
while(nondet()) {
    // Non-deterministic choice.
    bitvector y = nondet();
    if (not G(x,y)) {
        if (x == ⊤) {
            // Set the overflow
            // bit.
            o = ⊤;
            x = nondet();
        } else {
            // Increment a given
            // bitvector.
            x = succ(x);
        }
    }
}
```

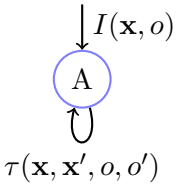


FIGURE 6.6: Counter Program and Transition System

the universal one:

$$\begin{aligned}\psi \equiv &\exists x_0, \ldots, x_{m-1}. \\ &\forall y_0, \ldots, y_{n-1}. G(x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1})\end{aligned} \tag{6.5}$$

Let us denote the bitvector $(x_0, \ldots, x_{m-1})$ as $\mathbf{x}$ and the bitvector $(y_0, \ldots, y_{n-1})$ as $\mathbf{y}$. Let $enc : \mathbb{B}^m \to [0, 2^m - 1]$ denote the function for standard integer encoding of the $\mathbf{x}$ bitvector, x_0 being the lowest-order bit and x_{m-1} the highest-order one. Let $succ : \mathbb{B}^m \setminus \{\top^m\} \to \mathbb{B}^m$ be the successor function such that $enc(succ(\mathbf{x})) = 1 + enc(\mathbf{x})$, which is only defined for non-overflowing values.

Now we define the transition system over the set of bolean variables $\mathbf{x}$ and the overflow bit o. Let the initial state $I(\mathbf{x}, o)$ be $\mathbf{x} = \bot \wedge o = \bot$, and let the transition relation $\tau(\mathbf{x}, \mathbf{x}', o, o')$ to be:

$$\begin{aligned}&(\neg(\forall \mathbf{y}. G(\mathbf{x}, \mathbf{y})) \wedge \\ &((\mathbf{x} \neq \top \wedge \mathbf{x}' = succ(\mathbf{x}) \wedge o' = o) \vee (\mathbf{x} = \top \wedge o' = \top))) \\ &\bigvee (\mathbf{x}' = \mathbf{x} \wedge o' = o)\end{aligned} \tag{6.6}$$

In plain terms, the transition relation may increment $\mathbf{x}$ as long as it is not overflowing and the guard can be falsified for some $\mathbf{y}$, and $\mathbf{x}$ is forced to stay constant on overflow or when it reaches some $\hat{\mathbf{x}}$ such that $\forall \mathbf{y}. G(\hat{\mathbf{x}}, \mathbf{y})$. Initialization and transition relation for the transition system, and the corresponding program are shown in Figure 6.6.

Lemma 6.1. There exists a non-trivial ($\not\equiv \top$) inductive invariant for the program in Figure 6.6 if and only if ψ (Equation 6.5) is satisfiable.

Observe that τ can be satisfied for all possible values of $\mathbf{x}$ by a suitable choice of $\mathbf{x}'$. Let $f(\mathbf{x})$ be the largest (under enc) possible value of $\mathbf{x}'$ which satisfies $\tau(\mathbf{x}, \mathbf{x}', o, o')$.

Proof. Sufficient Condition. Assume ψ is satisfiable for some $\hat{\mathbf{x}}$. Then $\hat{\mathbf{x}}$ is a fixed point under f

(as it satisfies G for all values of $\mathbf{y}$). Consider the set of values defined by $R \equiv \neg o \wedge enc(\mathbf{x}) \leq \hat{\mathbf{x}}\}$. It is inductive, since the largest value in R set maps to itself under f, and all other values map to the "next" (under enc) value in R. It is also non-trivial, since the bit o is defined not to be $\top$. □

Proof. Necessary Condition. Assume there exists a non-trivial inductive invariant for the program in Figure 6.6. At every transition, $\mathbf{x}$ either stays constant or is incremented by 1. Since we have assumed the existence of a non-trivial inductive invariant, there exists $\hat{\mathbf{x}}$ such that it is a fixpoint under f and $enc(\hat{\mathbf{x}}) \leq 2^m - 1$ (otherwise the entire state space is reachable, and the only possible inductive invariant is $\top$). This is only possible if $\forall \mathbf{y}.\, G(\hat{\mathbf{x}}, \mathbf{y})$ (otherwise $\hat{x}$ may be incremented). But this is exactly the condition for ψ being satisfiable. □

Corollary 6.1. For every non-trivial inductive invariant of the program in Figure 6.6 there exists some $\hat{\mathbf{x}}$ such that $\{\mathbf{x} \mid enc(\mathbf{x}) < enc(\hat{\mathbf{x}})\}$ is inductive. Furthermore, the reachable state space is exactly all $\mathbf{x}$ smaller (under enc) than $\hat{\mathbf{x}}$, and $\{\mathbf{x} \mid \mathbf{x} \neq \hat{\mathbf{x}}\}$ is inductive (as the states larger than $\hat{\mathbf{x}}$ are not reachable).

Now consider finding inductive (with respect to τ Figure 6.6) weakenings of the following formula ϕ:

$$\phi \equiv \bigvee (x_i \wedge \neg x_i) \tag{6.7}$$

Each x_i represents i'th bit of $\mathbf{x}$. Observe that for any $\hat{\mathbf{x}} \in [0, 2^m - 1]$, we can weaken ϕ to be equivalent to $\mathbf{x} \neq \hat{\mathbf{x}}$, by making a suitable weakening choice for every i'th bit of $\hat{\mathbf{x}}$ (if the i-th bit in $\hat{\mathbf{x}}$ is $\bot$ we replace $\neg x_i$ by $\top$, if it is $\top$ we replace x_i by $\top$).

From Corollary 6.1 we know that for every non-trivial inductive invariant there exists $\hat{\mathbf{x}}$, s.t. the set of all $\mathbf{x}$ not equal to $\hat{\mathbf{x}}$ is inductive. Thus if a non-trivial inductive invariant exists, there exists a non-trivial inductive weakening of ϕ. In Lemma 6.1 we have shown that deciding the existence of a non-trivial inductive invariant is as hard as deciding the satisfiability of an arbitrary $\exists^*\forall^*$ formula ψ, thus deciding an existence of a non-trivial inductive weakening is as hard as well.

Σ_2^p-completeness. Membership in Σ_2^p is proved in Lemma 6.1. Reduction from the Σ_2^p-complete problem is done from deciding the truth of $\exists^*\forall^*$ propositional formulas [Sto76, Th. 4.1]. Transforming G into τ can be done within a logarithmic working space. □

Relationship to Template Abstraction Complexity Lahiri and Qadeer [LQ09] consider the problem of *template abstraction*: given a precondition, a postcondition, a transition relation and a formula $\phi(C, X)$, C and X being sets of Boolean variables, check whether an appropriate choice of C makes ϕ an inductive invariant. They show this problem to be Σ_2^p-complete as well. Our class of problems is a strict subset of theirs (our weakening problems can be immediately translated into template abstraction problems, but not all template abstraction problems correspond to weakenings), but we still show completeness.

6.8 Conclusion and Future Work

We have proposed a "formula slicing" algorithm for efficiently finding potentially disjunctive inductive invariants in programs, which performs abstract interpretation in the space of weakenings over the formulas representing the "initial" state for each node. We have demonstrated

that it could verify many programs other approaches could not, and that the cost of running the algorithm in practice is surprisingly cheap.

The motivation for our approach is addressing the limitation of abstract interpretation which forces it to perform abstraction after each analysis step, which often results in a very rough over-approximation. Thus we believe our method is well-suited for augmenting numeric abstract interpretation.

6.8.1 Future Work

As with any new inductive invariant generation technique, a possible future work is investigating whether formula slicing can be used for increasing the performance and precision of other program analysis techniques, such as k-induction, predicate abstraction or property-directed reachability. An obvious extension is feeding the generated invariants to an analysis running policy iteration (Chapter 3).

Furthermore, the inductive weakening approach could also be used for the generalization of the k-induction algorithm over multiple properties. If we check a set of properties P for inductiveness under the loop transition τ, and not all properties are inductive, the weakening approach can find the largest inductive subset.

The conversion to the RCNF form is currently done syntactically, but it can be also done using the counterexamples to induction.

Part III

Engineering Contributions

Chapter 7

Implementation

7.1 Introduction

In the course of this thesis we have developed two novel tools, LPI (`http://lpi.metaworld.me`), which performs the local policy iteration described in Chapter 3, and Slicer (`http://slicer.metaworld.me`), which performs the formula slicing as described in Chapter 6. Both tools compute an inductive invariant for an input C program, and are implemented as CPAs inside the CPAchecker [BK11] framework.

In this chapter we describe various features of those tools, architecture, strength and weaknesses, usage instructions, and describe the various extensions.

7.2 Software Architecture

Simplified CPAchecker architecture is shown in Figure 7.1. The end user specifies a program to be analyzed and a safety property of interest, which is converted into a CFA (Definition 2.3) using Eclipse CDT [Fou] parser. The CFA is subsequently analyzed using a fixed point CPA algorithm (Algorithm 2.2), parameterized by a configurable program analysis (Section 2.10) provided by the developer. In turn, the analysis module often relies on the C to Formula package which converts a sequence of C statements into an SMT formula (Section 2.3), and on the JavaSMT (Chapter 8) engine for dispatching the formulas to the solver.

Our main technical contributions include LPI (Chapter 3) and Slicer (Chapter 6) which can be run by the CPAAlgorithm either in standalone mode, or combined with other analyses. Additionally, we were heavily involved in the creation of the JavaSMT library, described further in Chapter 8, and have contributed a very large number of patches across the entire CPAchecker codebase.

7.3 Installation Instructions

Contributed tools are present in the CPAchecker source code, which makes the installation from source remarkably simple, assuming the client machine has Java 8 and GIT installed:

```
> git clone https://github.com/sosy-lab/cpachecker.git
> cd cpachecker
> ant
```

The call to `ant` would fetch all the required dependencies, including the solver binaries. More detailed installation instruction are available at the website: `http://cpachecker.sosy-lab.org`.

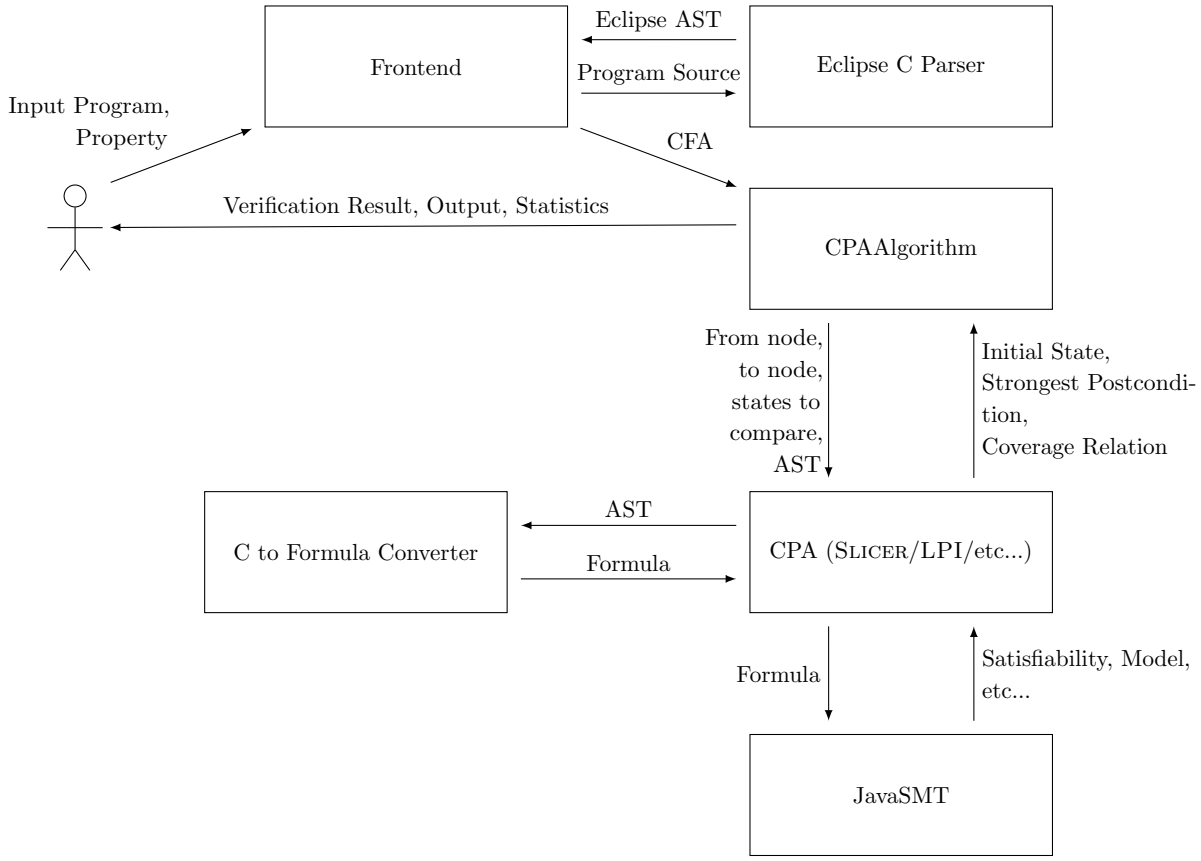


Figure 7.1: CPAchecker Architecture

7.4 Usage Instructions

```
1  #include<assert.h>
2  extern int __VERIFIER_nondet_int();
3  extern int __VERIFIER_assume(int condition);
4  int main() {
5      int sum = 0;
6      int bound = __VERIFIER_nondet_int();
7      __VERIFIER_assume(bound >= 0);
8
9      for (int i=0; i<bound; i++) {
10         sum++;
11     }
12     assert(sum == bound);
13 }
```

Figure 7.2: Sample C Program

Consider using LPI on a program shown in Figure 7.2. Note the following features of the example: external function with no body `__VERIFIER_nondet_int()` is used to model non-deterministic input and the function `__VERIFIER_assume(int condition)` is used to restrict the input space. The default LPI configuration is called `-policy` and it can be launched as follows from the CPAchecker directory:

```
> ./scripts/cpa.sh -preprocess -policy program.c
```

After running this file, the user get the log output from the tool, followed by the message that

the program was verified successfully: that is, the program analysis module was able to prove that no specification violations are reachable.

7.4.1 Configuration Options

When we have called CPAchecker we have used two command line options: `-preprocess` and `-policy`. The `-preprocess` command line switch tells CPAchecker to pre-run the C preprocessor on the program, which is necessary for parsing `#include` calls. Note that we have not explicitly specified the specification against which we are verifying: by default the specification looks for assertion violations. The `-policy` switch selects the *configuration* to be used for analysis, and is simply a shortcut to select one of the files in the `config` directory. The LPI tool is shipped with the following configurations:

- `-policy` standard LPI configuration, synthesizes *octagonal* templates.
- `-policy-intervals` a faster configuration, which only uses *interval* templates.
- `-policy-refinement` a configuration which uses the template refinement procedure (Chapter 4): the set of templates is continuously increased until the property can be proven.
- `-policy-k-induction` An analysis which runs k-induction, as described in Section 7.7.2, and uses LPI for the invariant generation. The `-policy-refinement` configuration is used for invariant generation.
- `-policy-summaries` An analysis performing summary generation with no inlining, capable of dealing with recursive programs, as described in Chapter 5. As of writing only available on the `summaries` branch.

The following configuration options are available for the Slicer tool:

- `-formula-slicing` generate inductive invariants from preconditions using SMT solver.
- `-formula-slicing-k-induction` The configuration which gives the invariant above to k-induction.

The usage of more detailed options is documented in the file `doc/ConfigurationOptions.txt`, shipped with CPAchecker. The options specific to LPI are grouped under the key `cpa.lpi` and the options specific to Slicer are grouped under `cpa.formula_slicing`.

7.4.2 Looking at the Output Further

After the run of the tool the `output` folder is generated which contains various output artifact describing the obtained results. If the client wishes to examine the inductive invariant embedded into the program as `assume` statements, the following command can be used:

```
> ./scripts/cpa.sh -preprocess -policy -setprop cinvariants.export=true program.c
```

That generates the file `output/inv-program.c` with the `assume` call specifying the inductive invariant embedded into the loop body shown in Figure 7.3.

Additionally, either to inspect the path to the property violation, or to look at the graphical representation of the invariant, the file `output/ARG.dot` might be useful (or for large files, often `output/ARGSimplified.dot`). This is explained in more detail in Section 7.6.

```c
#include<assert.h>
extern int __VERIFIER_nondet_int();
extern int __VERIFIER_assume(int condition);
int main() {
  int sum = 0;
  int bound = __VERIFIER_nondet_int();
  __VERIFIER_assume(bound >= 0);

  for (int i=0; i<bound; i++) {
    __VERIFIER_assume(i >= 0 && i == sum && i < bound);
    sum++;
  }
  assert(sum == bound);
  return 1;
}
```

FIGURE 7.3: Sample Program for Inductive Invariant Generation

7.5 CPA Formulation

In this section we give precise configurable program analysis definition for both SLICER and LPI. For both analyses, in order to avoid losing precision due to intermediate abstractions, we do not express the invariant as an abstract state at every node: instead the transfer relation operates on formulas and we only perform over-approximation at certain *abstraction points* (which correspond to loop heads in a well-structured CFA). This approach is inspired by adjustable-block encoding [BKW10], which performs the same operation for predicate abstraction.

Thus we introduce two lattices for each analysis: *abstracted states* (not to be confused with *abstract states* in general: both intermediate and abstracted states are *abstract*) for states associated with abstraction points (which can only express abstract states in the corresponding lattice) and *intermediate states* for all others (which can express regions using decidable SMT formulas).

Intermediate states represent reachable state-spaces using formulas directly, together with the meta-information to record the parent(s)[1] abstracted state.

Definition 7.1 (Intermediate State)**.** An *intermediate state* is a tuple (a_0, ϕ), where a_0 is a parent abstracted state, and $\phi(\mathbf{x} \cup \mathbf{x}')$ is a formula over a set of input variables $\mathbf{x}$ and output variables $\mathbf{x}'$.

For two intermediate states (a_1, ϕ_1) and (a_2, ϕ_2) with a_1 identical to a_2 the *merge operator* returns the disjunction $(a_1, \phi_1 \vee \phi_2)$. Otherwise, the states are kept separate. The coverage on intermediate states is defined using syntactic comparison on formulas: $(a_1, \phi_1) \preceq (a_2, \phi_2)$ iff ϕ_1 is syntactically equivalent to ϕ_2 and $a_1 \preceq a_2$ under the defined ordering on abstracted states. Such a coverage check is an over-approximation, yet can be implemented efficiently.

The postcondition computation runs symbolic execution: the successor of an intermediate state $(a, \phi(\mathbf{x} \cup \mathbf{x}'))$ under the edge $(A, \tau(\mathbf{x} \cup \mathbf{x}'), B)$ is the intermediate state $(a, \phi'(\mathbf{x} \cup \mathbf{x}'))$ with $\phi'(\mathbf{x} \cup \mathbf{x}') \equiv \exists \hat{\mathbf{x}}.\, \phi(\mathbf{x}, \hat{\mathbf{x}}) \wedge \tau(\hat{\mathbf{x}}, \mathbf{x}')$. Postcondition is only computed for intermediate state: in order to compute a postcondition of the abstracted state a, it is first converted to an intermediate

[1] If the summary computation described in Chapter 5 is enabled, intermediate states associated with return nodes have *two* parents: a state associated with a callsite, and a state associated with a summary. In such a case, the definitions reads as below, yet all operations on the parent state are done element-wise.

state $(a, \top)$. If after the postcondition computation the successor node is a loop head, then *abstraction* is performed on the resulting state. We now proceed to describe abstracted state, join operator over them, and the abstraction operator for both LPI and Slicer.

7.5.1 Abstraction for LPI

Definition 7.2 (LPI Abstracted State)**.** An LPI abstracted state is a tuple (i, A) where i is an intermediate state, and A is LPI abstracted state (Definition 3.1). Effectively to support adjustable block encoding we extend the abstract state definition with a meta-information recording the generating intermediate state.

The abstraction operator was defined in Algorithm 3.2, line 7, and the join in Algorithm 3.3. The partial order on abstracted states is defined by component-wise comparison of bounds associated with respective templates (Section 3.3.1).

7.5.2 Abstraction for Slicer

Definition 7.3 (Slicer Abstracted State)**.** A Slicer abstracted state is either an empty set $\emptyset$ denoting $\top$, or a tuple (i, ϕ), where i is an intermediate state (generating backpointer), and ϕ is a set of lemmas in RCNF form over $\mathbf{x}$.

The abstraction operation converts ϕ to a set of lemmas in RCNF form, and the merge operation performs weakening of the old state with respect to the new one. The comparison on two Slicer abstracted state is given by containment relation on the set of lemmas, using syntactical comparison on individual formulas.

7.6 Abstract Reachability Graph Generation

Both LPI and Slicer can be seen as traditional static analyses: all abstracted states are joined, and some form of widening (either value determination or inductive weakening) is used to enforce convergence. This approach has the following downsides:

- Combination with analyses which require *splitting* states (such an explicit value analysis or predicate abstraction with interpolants) inside `CompositeCPA` (Section 2.10.1) is problematic, due to the fact that LPI/Slicer can not split states, as it depends on the join step to enforce convergence, and an analysis which depends on splitting can not join, as that would severely affect the precision.
- It is not possible to generate the abstract reachability tree, which visualizes the analysis progression.
- It is not possible to get an abstract path for the property violation, which greatly enhances user's experience, and can be additionally used for interpolation.

We handle these issues using our *join emulation* approach, which produces an abstract reachability tree, as described in Section 4.4.1. Such an algorithm overcomes the disadvantages outlined above, and we lay the groundwork for combination with CPAs which rely on keeping the states separate.

```
1  extern int __VERIFIER_nondet_int();
2  extern int __VERIFIER_assume(int condition);
3  extern void __VERIFIER_assert(int condition);
4  int main() {
5    int sum = 0;
6    int bound = __VERIFIER_nondet_int();
7    for (int i=0; i<bound; i++) {
8      sum++;
9    }
10   assert(sum == bound);
11 }
```

FIGURE 7.4: Sample Program for ARG Generation

Example 7.1 (Using ARG for Verification Feedback)**.** Consider proving the assertion for the program shown in Figure 7.4. When launching LPI with a default configuration we get a message that the property could not be proven. In order to investigate why this is the case an obtain the abstract path to an error we can consult the generated ARG (by default generated in `output/ARG.dot`), shown in Figure 7.6. By visualizing the flow to the error, and checking a possible concretization of values at the error location, it can be quickly seen that an error is in the program: we have not ensured that `bound` has to be positive.

After modifying the program and adding a statement `__VERIFIER_assert(bound >= 0);` below an assignment to `bound`, LPI can verify the new task successfully. Again, we can visualize the produced proof by consulting the ARG, which is shown in Figure 7.5.

7.7 Extensions

Finding Non-Termination We have implemented an extension which allows to use our analyses for having an over-approximating check for the non-terminating behavior. If after the analysis was finished the set of reachable states does not contain any *exiting* state (`exit` call, `return` from main, assertion violation, etc) we report that the program is not terminating for any input.

Iteration Order In our experiments, we have found performance to depend on the iteration order. Experimentally, we have determined a good iteration order to be the recursive iteration strategy using the weak topological ordering [Bou93].

7.7.1 Combination with Other Configurable Program Analyses

One of the main advantages of the CPACHECKER architecture is the ability to run multiple "sibling" analyses together, using a `CompositeCPA` (Section 2.10.1). For our tools we have found the combinations with the following analyses to be helpful:

Location Analysis (`LocationCPA`, not written by us, enabled by default) for generality, even keeping track of location in CPACHECKER is a separate configurable program analysis.

Callstack Analysis (`CallstackCPA`, not written by us, enabled by default) unless summary generation is enabled (Chapter 5), the analysis with a `CallstackCPA` performs *dynamic inlining*:

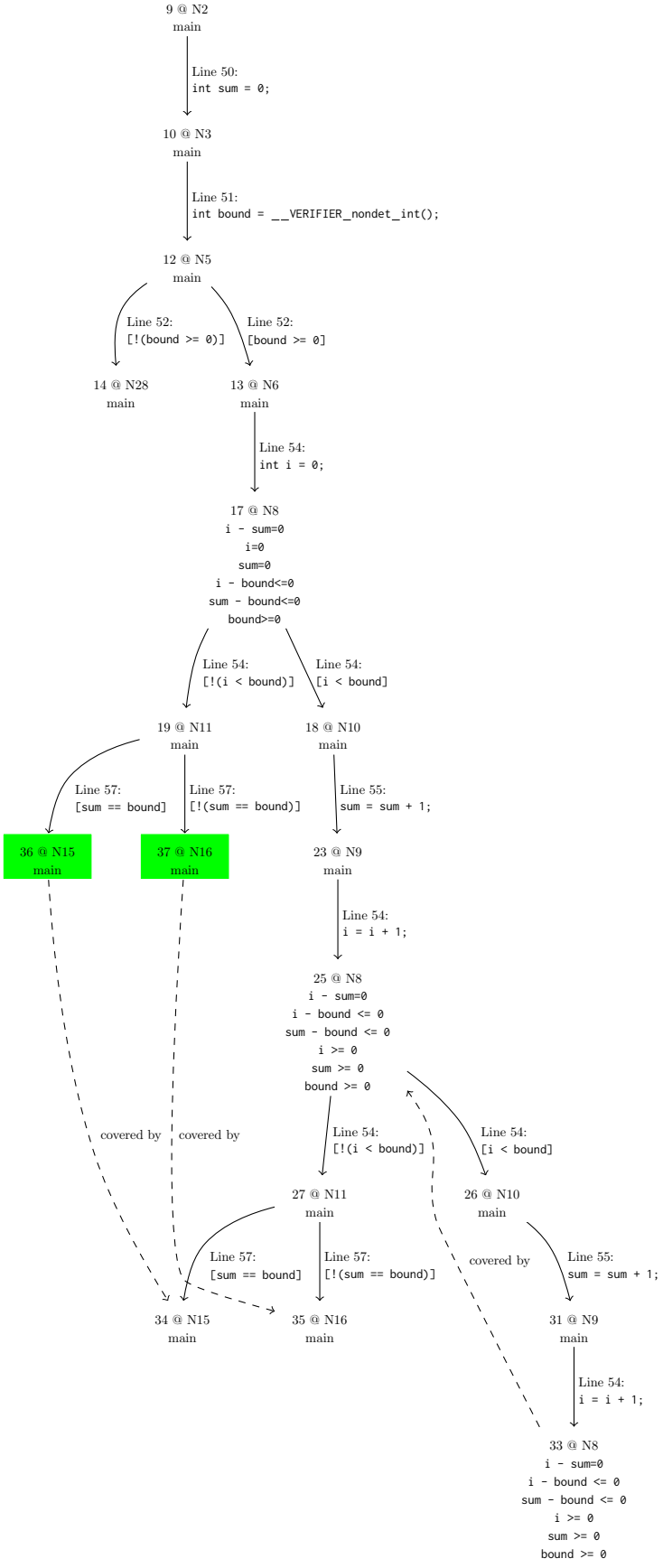


FIGURE 7.5: ARG For a Successfully Verified Program

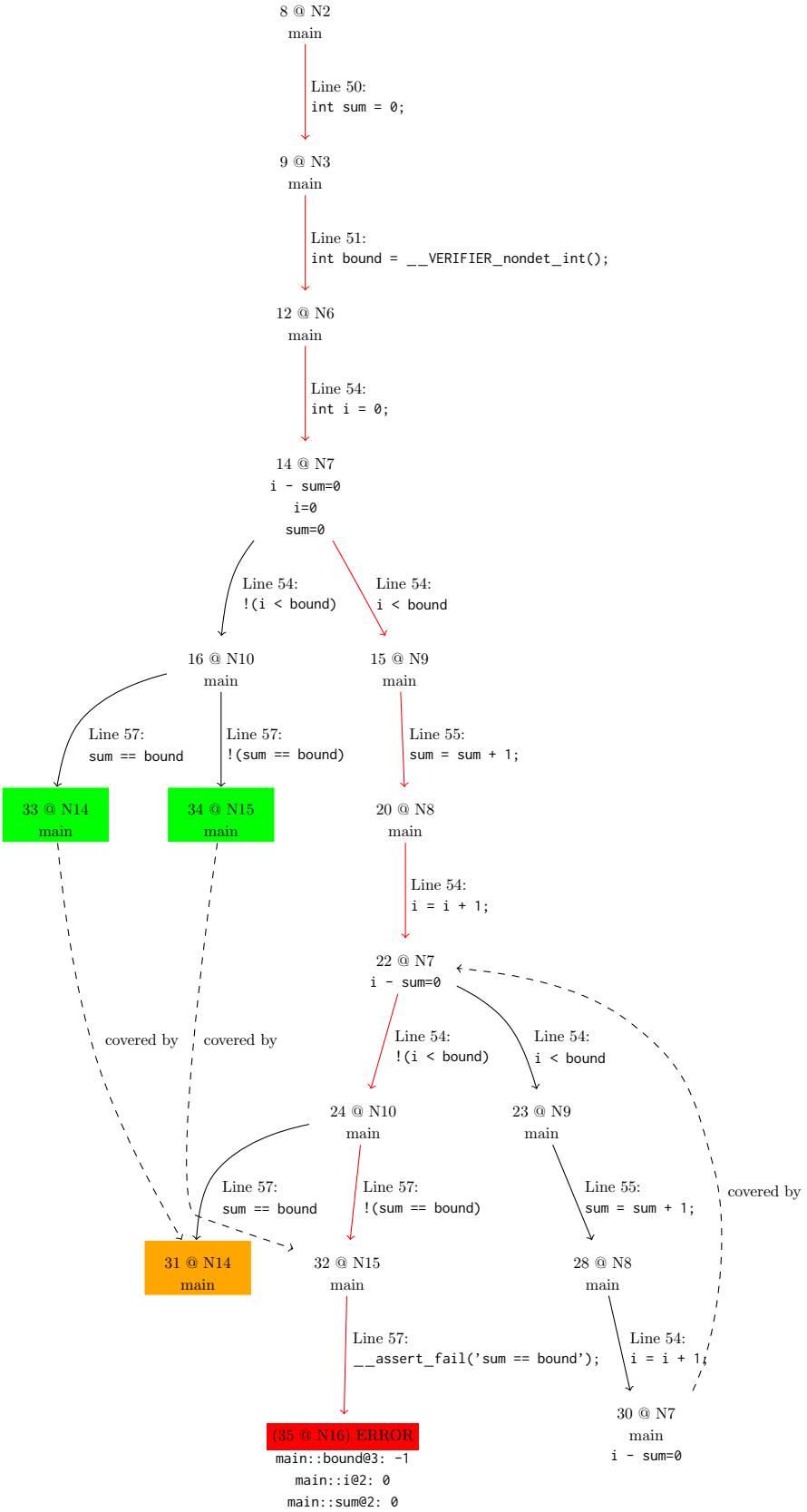


FIGURE 7.6: ARG Demonstrating an Error Path

procedures are inlined as they are explored (which can be often exponentially more performant than inlining all procedure calls upfront).

Loopstack Analysis (`LoopstackCPA`, not written by us, enabled by default) similarly to the callstack analysis, loopstack CPA can perform dynamic loop unwinding, which unwinds the loop until a certain criteria is met.

Function Pointer Analysis (`FunctionPointerCPA`, not written by us, enabled by default) Simple abstract interpretation-based analysis: each function pointer in the abstract domain is represented by a possible location it can be aliased to (undecidability of the aliasing problem does not arise as we use over-approximation). At each function pointer call the abstract value is constructed and all possible aliasing locations are explored. Note that the abstract domain is not *relational* and does not track the information on the *conditions* under which a function pointer may be aliased.

Congruence Analysis (`CongruenceCPA`, written by us, enabled by default) A congruence analysis which tracks whether a variable is even or odd (a more general congruence analysis may be used, but we did not find the need for it on our examples). During the abstraction step, the congruence information is conjoined to the formula being maximized, and the bounds from "sibling" CPAs are used for the congruence analysis.

Basic Backwards Reachability (`TargetReachabilityCPA`, written by us, not enabled by default) We apply a basic reachability check from the error nodes: we do not traverse the nodes from which the error location can not be *syntactically* reached (there is no backwards path from the error location to those nodes). This enhancement can be seen as a combination of forward invariant-generating analysis with a very simple backwards location-based analyzer.

Value Analysis (`ValueCPA`, not written by us, not enabled by default) For some examples we have found a combination with explicit value analysis [BL13] to be useful: while LPI can efficiently reason about variables involved in arithmetic operations, the explicit value analysis can track variables which can assume only a small number of discrete values.

7.7.2 Combination with k-Induction

The basic backwards reachability, described in Section 7.7.1, restricts the search space to the backwards reachable nodes. However, the backwards search is purely location-based and does not take the property into account. Thus it can be extended to k-induction with invariants [KT11] which checks whether the negation of the error property, supported by the invariants, is inductive.

As the output of both Slicer and LPI tools is an inductive invariant, they can be naturally augmented with k-induction. The invariant produced by our tools is fed to the k-induction [BDW15] procedure, already present in CPAchecker. For a given value of k, k-induction performs two checks: whether the error state is reachable from the initial one in k steps (forward reachability), and whether the negation of the error property is k-inductive, subject to the strengthening by the invariant produced by the analysis. Invariant generation (including continuous refinement) runs asynchronously to the k-induction procedure, and they are both continuously refined (number of templates is increasing, and so is the value of k).

We have used k-induction as it is a natural fit to our invariant generation procedure due to support for continuously refined invariants. LPI and Slicer improve the precision of pure k-induction, as the inductiveness check may fail due to counterexamples-to-induction which are not reachable in the selected abstract domain.

Moreover, from our experience, when using an invariant-generating method for verification there is no point in *not* using it together with k-induction, as it strictly increases the precision (by combining forward and backwards reachability analyses) often at a rather small time cost. Additionally, it allows the tool to perform bugfinding using bounded model checking with depth k, which can also benefit from the produced invariant.

7.8 Conclusion

During the course of this thesis, we have implemented a number of different analysis within the CPAchecker framework. Although using an existing framework has certain limitations, the overall experience was far outweighed by a number of advantages, including much shorter iteration time (once the framework is learned it is possible to write a new analysis in a very short timeframe), and a potentially larger impact (users are more likely to use an existing tool with a new configuration, rather than switching to a new tool entirely).

7.8.1 Software Project and Contributors

The Slicer and LPI code was written by George Karpenkov. CPAchecker is mainly developed by the Software Systems Lab at the University of Passau/Ludwig-Maximilian University of Munich. The k-induction module was developed by Matthias Dangl. All the code mentioned in this thesis is distributed under the Apache 2.0 license.

7.8.2 Future Work

Parallelization The approach for generating an ART described in Section 7.6 has an additional advantage, as it allows the *parallelization* of described analysis, both for Slicer and LPI. The computations happening in different branches, where one state is not a successor of another, can not possibly influence each other, and thus can be safely parallelized.

We have not implemented such an extension, as it is a significant implementation task requiring a large engineering investment, including changing the core algorithms of CPAchecker, and can only yield a constant-time speed up. Moreover, underlying SMT solvers only support parallelization with use of multiple contexts, which might require many copying steps for formulas.

Chapter 8

JavaSMT Library

8.1 Introduction

The tools described in this thesis heavily rely on the capabilities of SMT solvers, as do many other approaches for software verification and automated bug finding. This chapter presents a new library which eases the communication with SMT solvers for programs written in the Java language. A large part of this chapter was published [KBF16] in "Verified Software: Techniques, Tools and Evaluation" conference in 2016.

The SMT-LIB [BFT15] initiative defines a common interface language for SMT solvers, much like SQL standardizes the interface to a relational database. However, from the perspective of a tool developer, using the textual SMT-LIB communication channel is often suboptimal. Firstly, it does not expose all features that modern solvers offer: interpolation [1], multiple independent solvers, formula introspection, and optimization modulo theories are not included in SMT-LIB 2.0. It is also not possible to conditionally *store* formulas for future reuse and remove them when they are no longer needed.

Secondly, such a textual communication can be very inefficient, because all queries to the solver have to be serialized to strings, and all solver output has to be parsed. For a tool that poses a large number of trivial queries (such as in PDR [BM08]), parsing and serialization can become a bottleneck.

However, when using a solver API directly, users face the design problem of "solver lock-in", which makes it difficult to switch to a different solver without rewriting a large chunk of the application.

We propose JavaSMT, a library that exposes a common API layer across multiple backend solvers. It is written in Java and is available as open source under the Apache 2.0 License on GitHub at the URL `https://github.com/sosy-lab/java-smt`. JavaSMT communicates with solvers using their API, and imposes only a minimal amount of overhead. For solvers implemented in Java the exposed API is used directly, and for the solvers in other languages we integrated JNI bindings.

Chapter Outline We start by describing the library features in Section 8.2. In Section 8.3 we describe the project architecture, and in Section 8.4 we state the memory handling strategies used for formulas. Finally, in Section 8.5 we present implementation of the Houdini algorithm as a case study presenting the library features, followed by related work outline in Section 8.6 and conclusion in Section 8.7.

[1] A proposal draft [CH12] exists since 2012.

Table 8.1: Theories and features supported by different SMT solvers

	MathSAT	OptiMathSAT	Z3	SMTInterpol	Princess
Integer	+	+	+	+	+
Rational	+	+	+	+	-
Array	+	+	+	+	+
Bitvector	+	+	+	-	-
Float	+	+	-	-	-
Unsat Core	+	+	+	+	-
Partial Models	-	-	+	-	+
Assumptions	+	+	+	+	+
Quantifiers	-	-	+	-	+
Interpolation (Tree/Sequential)	+	+	+	+	+
Optimization	-	+	+	-	-
Incremental Solving	+	+	+	+	+
SMT-LIB2	+	+	+	+	+

8.2 Features

JavaSMT currently provides access to five different SMT solvers: MathSAT [Cim+13], OptiMathSAT [ST15], Z3 [MB08], SMTInterpol [CHN12], and Princess [Rüm12]. Table 8.1 lists the theories and features that are supported by these solvers.

8.2.1 Formula Representation

To keep the memory overhead low, JavaSMT does not store an own internal representation of the formulas, but keeps only one single pointer to each formula in the solver's memory, possibly with an additional pointer to the current solver context. Consequently, the memory footprint of JavaSMT is proportional to a small constant multiplied by the number of formulas that the client application needs a reference to, *regardless* of the size of the constructed formulas.

This choice ensures high performance, but obstructs transferring formulas between different contexts for different operations, such as checking satisfiability with Z3 and performing interpolation with SMTInterpol. For such inter-solver translations we use SMT-LIB serialization.

8.2.2 Type Safety

Using and enforcing types is beneficial for a software library, because it guarantees the absence of errors that are caused by incorrect type usage *at compile time* and can increase the level of trust in the software. Improving such confidence is particularly important for tools for software verification, because the verdict of such tools is only reliable if all components operate correctly ("who verifies the software verifier").

JavaSMT uses the Java type system to differentiate between the different sorts of formulas (e.g., `BooleanFormula` and `IntegerFormula`) and guarantees that all operations respect the formula type. The typed interface avoids incorrect operations (such as adding integers to

booleans), which would not pass the compiler. Type safety also extends to model evaluation: for example, evaluating an `IntegerFormula` is guaranteed at compile time to return a `BigInteger`.

8.2.3 Formula Introspection

In many applications, formula introspection is a required feature. For instance, an analysis might wish to re-encode expensive non-linear operations as uninterpreted functions, or find and rename all variables used in the formula.

In our experience with formula introspection and transformation code in CPAchecker, we have discovered that writing *correct and robust* formula-traversing code can be very challenging both for the client and for the library, due to:

- cases missed by the client, e.g., an unexpected `XOR`,
- incorrect assumptions by the client, such as assuming that the input formula has no quantifiers,
- not performing memoization for recursive traversals, resulting in exponential blow-up on formulas represented as directed acyclic graphs, or
- performing recursive traversal using recursion is not optimal, because it can result in stack-overflow exceptions on large formulas.

In order to decrease the likelihood of such bugs, we use the Visitor design pattern [Gam+95] for formula traversal and transformation. We expose two different kinds of visitors, `FormulaVisitor`, supporting any sorts, and the `BooleanVisitor`. The boolean visitor requires implementations for boolean primitives that can occur in the formulas (equality, implication, etc.) and matches all other formulas as atoms. It is useful for transformation of the boolean structure of the formula, such as a negated normal form conversion. The `FormulaVisitor` does not explicitly require matching each possible function, but provides an enumeration consisting of most common function declarations (addition, subtraction, comparison, etc.) and can be used to recursively traverse the entire formula, e.g., in order to find all used variables. Each visitor can be used in three different modes of operations: traversal, where only the root formula is visited, recursive traversal, where each sub-term is visited exactly once, and recursive transformation, where a visitor is supposed to create a new sub-term for each sub-term visited.

Our experience shows that a visitor-based approach leads to a considerably safer code as compared to direct formula manipulation.

8.2.4 Handling Interruptions

SAT-solving is a canonical NP-complete problem, and extending the scope to support SMT makes it even harder. The problems posed to the solvers often have very high complexity, and a solver often becomes the most time-consuming component of the client application. With such a complexity the ability to support interrupts becomes critical, as it might be undesirable to kill the client application because the solver computation has diverged.

In order to solve this problem, JavaSMT uses a `ShutdownNotifier` tool which allows it to handle interrupts gracefully across all solvers. Any client component can request an interrupt on any thread, and all solvers promptly terminate their computations.

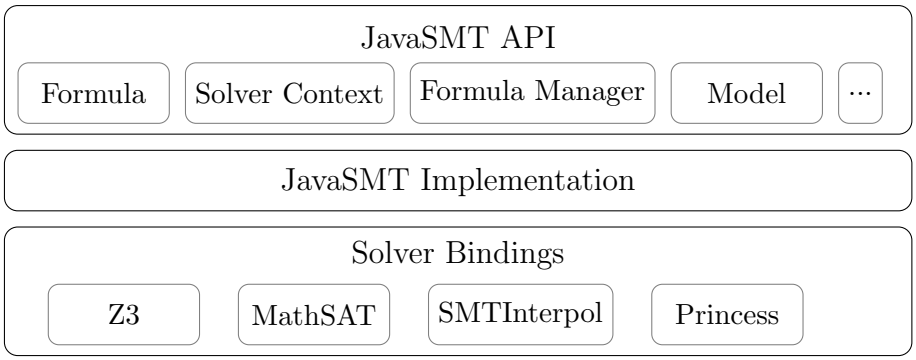


FIGURE 8.1: JAVASMT Architecture

8.2.5 Multithreading Support

JAVASMT provides two different mechanisms for multithreading support. Firstly, as mentioned in Section 8.2.4, any thread is allowed to interrupt the computations on any thread, including the computations performed in native code. Additionally, the *solving* can proceed in parallel even for the same solver, provided the computation is performed on different solver *contexts*. JAVASMT provides the translation API which can be used to exchange the information between different threads.

8.3 Project Architecture

The overall structure of the library is shown in Figure 8.1. An interaction with the JAVASMT library starts with a `SolverContextFactory`, which is used to create a `SolverContext` object, encapsulating a context for a particular solver. All further interaction is performed through the `SolverContext` class, which exposes the features outlined in Section 8.2. Instances of `SolverContext` are *not* thread-safe, and should be accessed only from a single thread. However, separate contexts are independent from each other and can be safely used from different threads.

An interface to every represented solver is implemented as a separate package with an entry class that implements the `SolverContext` API.

8.4 Memory Management

Different SMT solvers resort to different strategies for memory management. The solvers running in managed environments (e.g., SMTINTERPOL and PRINCESS running on JVM) use the available garbage collector, while solvers exposing a C API have to offer the memory management facilities to the user. The underlying problem is that for a library that exposes its API through the native non-managed language, it is *impossible* to know whether a previously returned object is still referenced by the client application, or whether it can be deleted.

MATHSAT and OPTIMATHSAT expose a "manual" garbage-collection interface, which removes all formulas except those that are specifically requested to be kept. This requires an application to keep track of created objects that can still be referenced.

Z3 uses a reference-counting approach, where an object is considered unreachable whenever its reference count reaches zero. While this interface has an automatic memory management in C++ (incrementing references in constructors, and decrementing in destructors) using it in an *efficient* and correct way is surprisingly difficult from Java.

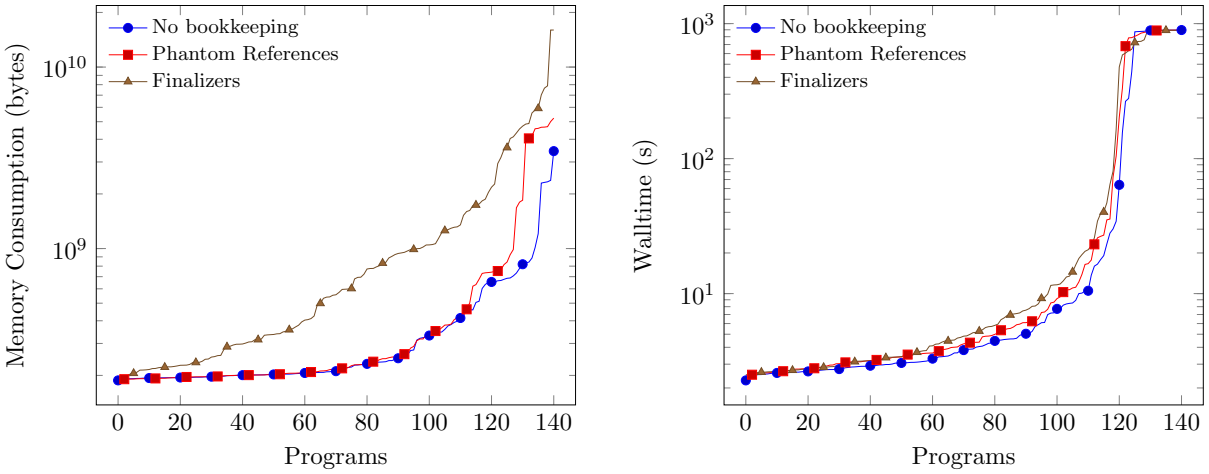


FIGURE 8.2: Resource usage comparison across different memory management strategies for Z3

The official Z3 Java API is using Java finalizers to decrement the references, explicitly performing locking on the queue of references that need to be decremented. Unfortunately, finalizers are known to have a very severe memory and performance penalty [Blo08]. Thus we have developed our own Z3 JNI bindings with a memory strategy based on using `PhantomReference` and `ReferenceQueue`, provided by the JDK to get a more fine-grained control over the garbage collection.

We present the performance evaluation of three different memory managing strategies for Z3: (1) using the official Z3 API, which relies on finalizers, (2) using our phantom reference-based implementation, and (3) not closing resources at all. We have chosen a benchmark setup that runs a program analysis with local policy iteration [KMW16] on the SV-COMP [Bey16] data set. Obtained results are shown in Figure 8.2. Unsurprisingly, the approach using finalizers has worst performance by far, with performance penalty often eclipsing the analysis time, and a very large memory consumption. The no-GC approaches minimizes both memory and time consumption. We attribute the high performance of no-GC approach to the hash-consing used in Z3, which results in no additional memory consumption for ASTs that were previously already constructed.

8.5 Case Study: Inductive Formula Weakening

To give a tour of the library, we present a small but usable implementation of the inductive-invariant synthesis algorithm called HOUDINI [FL01]. In order to provide the context, we include a brief background that explains the motivation and how the algorithm works.

8.5.1 Implementation Task

The HOUDINI algorithm finds a maximal inductive subset of a given a set L of *candidate* lemmas. It repeatedly checks $\bigwedge L$ for inductiveness, and updates L to exclude the lemmas that give rise to counterexamples-to-induction. At the end the algorithm terminates with an inductive subset $L_I \subseteq L$.

Counterexamples-to-induction are derived from the *model* returned by an SMT solver in response to an inductiveness checking query (such a model exists iff the conjunction of lemmas is not inductive). Given a model $\mathcal{M}$, the HOUDINI algorithm filters out all lemmas $l \in L$ for

```java
public class HoudiniApp {
  private final FormulaManager fmgr;
  private final BooleanFormulaManager bfmgr;
  private final SolverContext context;

  public HoudiniApp(String[] args) throws Exception {
    Configuration config = Configuration.fromCmdLineArguments(args);
    LogManager logger = new BasicLogManager(config);
    ShutdownNotifier notifier = ShutdownManager.create().getNotifier();

    context = SolverContextFactory.createSolverContext(
        config, logger, notifier);
    fmgr = context.getFormulaManager();
    bfmgr = context.getFormulaManager().getBooleanFormulaManager();
  }
}
```

FIGURE 8.3: JAVASMT initialization

```java
private BooleanFormula prime(BooleanFormula input) {
  return fmgr.transformRecursively(
      new FormulaTransformationVisitor<Formula>() {

    @Override
    public Formula visitFreeVariable(Formula f, String name) {
      return fmgr.makeVariable(
          fmgr.getFormulaType(f), name + "'");
    }

  }, input);
}
```

FIGURE 8.4: Transforming formulas with JAVASMT

which $\mathcal{M} \models \neg l(X')$ holds. After such filtering is applied in a fixed-point manner, a (possibly empty) inductive subset remains.

8.5.2 Implementation

Initialization: To initialize JAVASMT, we recommend to either pass the required classes using dependency injection, or to initialize them in a constructor, as shown in Figure 8.3. This code snippet generates a configuration from passed command-line arguments (configuration can choose a solver, and tweak any of its options), generates a logger instance, and initializes the solver context.

Formula Transformation: The HOUDINI algorithm gets a set of lemmas as input. However, for checking inductiveness we need *primed* versions of these lemmas, which we obtain by renaming all free variables using a transformation visitor as shown in Figure 8.4.

Instead of directly removing asserted lemmas from the solver, we use annotation with auxiliary *selector* variables. Each lemma l_i is converted to $l_i \vee s_i$, where s_i is a fresh boolean variable. After such an annotation, the lemma l_i can be relaxed by asserting an assumption s_i. The code for input-lemma annotation is shown in Figure 8.5. Finally, the main HOUDINI loop, which performs lemma filtering until inductiveness, is shown in Figure 8.6.

```java
public List<BooleanFormula> houdini(
    List<BooleanFormula> lemmas, BooleanFormula transition)
    throws Exception {
  List<BooleanFormula> annotated = new ArrayList<>();
  List<BooleanFormula> annotatedPrimes = new ArrayList<>();
  Map<Integer, BooleanFormula> indexed = new HashMap<>();

  for (int i = 0; i < lemmas.size(); i++) {
    BooleanFormula lemma = lemmas.get(i);
    BooleanFormula primed = prime(lemma);

    annotated.add(bfmgr.or(getSelectorVar(i), lemma));
    annotatedPrimes.add(bfmgr.or(getSelectorVar(i), primed));
    indexed.put(i, lemma);
  }

  // ... Continuted Later ...
}

private BooleanFormula getSelectorVar(int idx) {
  return bfmgr.makeVariable("SEL_" + idx);
}
```

FIGURE 8.5: Annotating formulas with JAVASMT

8.6 Related Work

The package JSMTLIB [Cok13] is a solver-agnostic library for Java which uses SMT-LIB for communication with the solvers, and thus has the associated restrictions outlined in Sect. 8.1, including costly serialization overhead and a limitation to the features offered by SMT-LIB. In contrast, our work presents a solver-independent library for Java which connects directly to the solvers API.

The newly published JDART [Luc+16] tool bundles a JCONSTRAINTS library that offers a functionality similar to JAVASMT. However, JAVASMT has more features, communicates with solvers using their API, and provides an efficient memory-management strategy (JCONSTRAINTS uses the official Z3 Java API, which relies on finalizers). Additionally, our library provides several solvers that can be installed automatically and one simple configuration option to switch between them.

For JCONSTRAINTS, the user has to manually include and configure all the solver's bindings and binaries. We have learned that these steps are complicated and error-prone, as the library might be used as part of a bigger software system. Thus, our solvers and their bindings do not require to setup any special environment.

The problem of creating such a library has also been tackled for Python in PYSMT [GM15]. In contrast to our work, PYSMT keeps the formula structure itself, while delegating the queries to the solvers. While this allows for creating formulas without any solvers installed, and for easier transfer of formulas between different contexts, it incurs a large memory overhead.

8.7 Conclusion

We have presented JAVASMT, a new library for efficient and safe communication with SMT solvers. The advantages of using such a library over communicating with SMT-LIB include

```
try (ProverEnvironment prover =
    context.newProverEnvironment(ProverOptions.GENERATE_MODELS)) {
  prover.addConstraint(transition);
  prover.addConstraint(bfmgr.and(annotated));
  prover.addConstraint(bfmgr.not(bfmgr.and(annotatedPrimes)));

  while (!prover.isUnsat()) {
    Model m = prover.getModel();
    for (int i = 0; i < annotatedPrimes.size(); i++) {
      BooleanFormula annotatedPrime = annotatedPrimes.get(i);
      if (!m.evaluate(annotatedPrime)) {
        prover.addConstraint(getSelectorVar(i));
        indexed.remove(i);
      }
    }
  }
}
return new ArrayList<>(indexed.values());
```

FIGURE 8.6: HOUDINI main loop with JAVASMT

performance, access to new features, and the ability to control which formulas remain in scope and which should be discarded. Some disadvantages exist as well — using JAVASMT means restricting to the supported solvers, and relying on JAVASMT developers to update the solvers in time. Our experience with using SMT solvers is that for applications posing a few large, monolithic queries communication using SMT-LIB is more optimal, while for tools that post many cheap, incremental queries, using the API via JAVASMT is the better solution.

New editions of SMT-LIB could make missing features like interpolation available (proposed draft already exists [CH12]), but giving the user control over memory management for formulas (Sect. 8.4), or allowing efficient communication without string serialization and parsing may be far outside of the scope of SMT-LIB initiative. So for users requiring such features, an intermediate-layer library is always beneficial.

8.7.1 Future Work

Currently we are collaborating with CVC4 developers to add CVC4 support to JAVASMT. Additionally, there are plans to add support for Horn clauses, and add different Horn clause solvers as backends.

Part IV

Conclusion

Chapter 9

Conclusion

In this thesis we have developed new algorithms based on policy iteration for efficient analysis of inter- and intraprocedural programs. Our contribution is both theoretical, largely formulated as new algorithms and their properties, and practical, provided as software artifacts.

9.1 Contributions Outline

In Chapter 3 we have presented extended background for max-policy iteration, and we have developed the new *local* policy iteration algorithm (LPI). The LPI algorithm formulates policy iteration as traditional abstract interpretation, by constructing a precise *widening* operator, which guarantees the convergence to the least inductive invariant after finitely many applications.

The algorithm operates within a *template constraints* abstract domain, where each abstract state represents a convex polyhedra which *shape* is pre-specified before the analysis. Thus in Chapter 4 we have addressed the *template synthesis* problem by developing multiple algorithms for generating template annotations, and we have performed extensive empirical comparison. Additionally, we have presented an algorithm for generating an *abstract reachability tree* representing a run of an arbitrary abstract interpretation, including LPI. Using abstract reachability trees establishes an ability to *refine* the abstract domain using *counterexample traces* in CEGAR spirit. We have shown how new templates can be generated from the results of an interpolation procedure, and stated the associated results on the algorithm completeness.

In Chapter 5 we have approached the problem of generating invariants using policy iteration for interprocedural, potentially recursive programs. We have developed a framework for analyzing such programs using intraprocedural abstract interpretation, and we have studied its parameterization under the LPI algorithm, which guarantees obtaining the least inductive invariant for a pre-specified summary structure.

Finally, in Chapter 6, we have formulated a new algorithm for augmenting abstract interpretation by generating potentially non-convex inductive invariants derived from symbolic execution and subsequent weakening.

The second part of our thesis was devoted to practical contributions: we have described our implementation in Chapter 7, and the JavaSMT library in Chapter 8.

9.2 Importance of Engineering

In Chapter 1 we have stated that formal methods are often critized for not providing a cost-effective amount of value to software engineers. Thus an important goal of this thesis was not just developing new approaches and algorithms, but also providing efficient implementations,

which could be potentially used by software engineers. We have aimed to provide robust, scalable, and well-engineered solutions, with plans for maintenance in the future.

With great help from CPAchecker [BK11] community, those goals were largely fulfilled: our approaches were implemented in the state-of-the-art program analysis framework, with a sizeable developer- and user-base. Additionally, as a result of our work, CPAchecker was improved in many aspects.

Furthermore, the JavaSMT library we have developed makes SMT solvers more accessible to the general community. As a result, it is already actively used by multiple researches, and additional effort to integrate more solvers is underway.

9.3 Future Work and Research Directions

This thesis addresses problems at the intersection of multiple research areas: traditional dataflow analysis, static program analysis with abstract interpretation, and logic-based model checking. In the spirit of configurable program verification this is another step towards removing the borders between the domains, and generating new results using the combinations of approaches.

Combinations with Other Analyses Formulating policy iteration as an LPI enables seamless combinations with other analyses, including lazy abstraction [McM06]. Intuitively, invariants resulting from both approaches should be able to complement each other during the analysis, resulting in a stronger precision than any of the analyses individually. We have only done preliminary experiments in this area, and detailed experimental study of such combinations (including formula slicing) remains an item for future work.

Using Weakest Precondition In this thesis we have focused exclusively on the *strongest postcondition* semantics, where the invariant is propagated forwards in the direction of the program control flow. Using weakest precondition semantics, which runs *backwards* with respect to the program might be a better approach for verifying properties of interest. Initial investigation has shown the approach to be challenging due to extra quantifiers appearing in combination with large block encoding.

Supporting Non-Convex Invariants A common theme we have found is that very often a convex invariant is inherently insufficient to verify a desired property. More work can be done on using abstract interpretation approach to generate non-convex invariants based on properties of programs which are to be verified.

A promising direction which can be used in conjunction with abstract interpretation is *splitting* states based on counterexample traces in the spirit of Sankaranarayanan et al. [San+06].

9.3.1 Towards Software Systems Verification

In this thesis we have based our experiments exclusively on the SV-COMP dataset. Applying an abstract interpretation-based tool to a large, annotated collection of programs has an advantage of knowing whether the analysis result is *useful* with respect to verifying the property of interest.

Yet on some level it is unsatisfying: while we have corrected a number of verification verdicts in the canonical benchmark source, most of the time, no *new* bugs can be found and no new verification verdicts can be issued on the known data.

Thus an important research avenue would be applying the developed approaches towards actual *programs* with the goal of verifying properties of interest, such as lack of overflows.

Such advances would enter the realm of commercial tools such as Coverity [Bes+10], yet might be necessary for demonstrating the applicability of new methods.

Bibliography

[ACI10] Corinne Ancourt, Fabien Coelho, and François Irigoin. "A Modular Static Analysis Approach to Affine Loop Invariants Detection". In: *NSAD*. Vol. 267. Elsevier, 2010, pp. 3–16 (cit. on p. 106).

[AGG10] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. "Coupling Policy Iteration with Semi-definite Relaxation to Compute Accurate Numerical Invariants in Static Analysis". In: *ESOP*. 2010, pp. 23–42 (cit. on p. 46).

[Alu+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-guided synthesis". In: *FMCAD*. 2013, pp. 1–8 (cit. on p. 84).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986 (cit. on pp. 25, 32, 39, 50).

[ASV12] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. "Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis". In: *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*. 2012, pp. 157–172 (cit. on p. 106).

[Bar+01] Howard Barringer, Saddek Bensalem, Klaus Havelund, Insup Lee, Grigore Rosu, and Oleg Sokolsky. *Runtime Verification*. 2001. URL: `http://runtime-verification.org/` (cit. on p. 24).

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *CAV*. 2011, pp. 171–177 (cit. on p. 35).

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "Boosting k-Induction with Continuously-Refined Invariants". In: *CAV*. 2015, pp. 622–640 (cit. on p. 147).

[Bes+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World". In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75 (cit. on p. 161).

[Bey+04] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "Generating Tests from Counterexamples". In: *Proc. ICSE*. IEEE, 2004, pp. 326–335 (cit. on pp. 27, 36).

[Bey+07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The Software Model Checker Blast". In: *Int. J. Softw. Tools Technol. Transfer* 9.5-6 (2007), pp. 505–525 (cit. on p. 88).

[Bey+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. “Software model checking via large-block encoding”. In: *FMCAD*. 2009, pp. 25–32 (cit. on pp. 50, 57).

[Bey15] Dirk Beyer. “Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015)”. In: *TACAS*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 401–416 (cit. on p. 79).

[Bey16] Dirk Beyer. “Reliable and Reproducible Competition Results with BenchExec and Witnesses”. In: *Proc. TACAS*. LNCS 9636. Springer, 2016, pp. 887–904 (cit. on pp. 119, 124, 131, 153).

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015 (cit. on p. 149).

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”. In: *CAV*. 2007, pp. 504–518 (cit. on pp. 19, 50, 51, 57).

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems”. In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21 (cit. on pp. 45, 47).

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. “Bounded model checking”. In: *Advances in Computers* 58 (2003), pp. 117–148 (cit. on p. 26).

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. “Symbolic Model Checking without BDDs”. In: *TACAS*. LNCS 1579. Springer, 1999, pp. 193–207 (cit. on p. 36).

[Bjø+15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. “Horn Clause Solvers for Program Verification”. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. 2015, pp. 24–51 (cit. on p. 106).

[BK11] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *CAV*. 2011, pp. 184–190 (cit. on pp. 52, 56, 79, 139, 160).

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. “Predicate Abstraction with Adjustable-Block Encoding”. In: *Proc. FMCAD*. FMCAD, 2010, pp. 189–197 (cit. on pp. 57, 142).

[BL13] Dirk Beyer and Stefan Löwe. “Explicit-State Software Model Checking Based on CEGAR and Interpolation”. In: *FASE*. 2013, pp. 146–162 (cit. on pp. 88, 147).

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A Static Analyzer for Large Safety-Critical Software”. In: *PLDI*. ACM, 2003, pp. 196–207 (cit. on p. 27).

[Blo08] Joshua Bloch. *Effective Java*. Prentice Hall, 2008 (cit. on p. 153).

[BM07] Aaron R. Bradley and Zohar Manna. “Checking Safety by Inductive Generalization of Counterexamples to Induction”. In: *FMCAD*. 2007, pp. 173–180 (cit. on pp. 124, 127).

[BM08] Aaron R. Bradley and Zohar Manna. “Property-directed incremental invariant generation”. In: *Formal Asp. Comput.* 20.4-5 (2008), pp. 379–405 (cit. on p. 149).

[BM75] J. W. de Bakker and Lambert G. L. T. Meertens. “On the Completeness of the Inductive Assertion Method”. In: *J. Comput. Syst. Sci.* 11.3 (1975), pp. 323–357 (cit. on p. 37).

[Boi98] Bernard Boigelot. “Symbolic Methods for Exploring Infinite State Spaces”. PhD thesis. Universite de Liege, 1998 (cit. on p. 36).

[Bol+13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. “A Formally-Verified C Compiler Supporting Floating-Point Arithmetic”. In: *ARITH, 21st IEEE International Symposium on Computer Arithmetic*. IEEE Computer Society Press, 2013, pp. 107–115 (cit. on p. 26).

[Bou93] François Bourdoncle. “Efficient chaotic iteration strategies with widenings”. In: *Formal Methods in Programming and Their Applications*. Vol. 735. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 128–141 (cit. on pp. 42, 128, 144).

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. “νZ - An Optimizing SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 194–199 (cit. on p. 56).

[BPR03] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. “Boolean and Cartesian abstraction for model checking C programs”. In: *STTT* 5.1 (2003), pp. 49–58 (cit. on pp. 48, 126).

[BR00] Thomas Ball and Sriram K. Rajamani. “Bebop: A Symbolic Model Checker for Boolean Programs”. In: *SPIN 00: SPIN Workshop*. Lecture Notes in Computer Science 1885. Springer-Verlag, 2000, pp. 113–130 (cit. on p. 106).

[Bra+15] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. “Safety Verification and Refutation by k-Invariants and k-Induction”. In: *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. 2015, pp. 145–161 (cit. on p. 57).

[Bra07] Aaron R. Bradley. “Safety Analysis of Systems”. PhD thesis. Stanford University, 2007 (cit. on p. 39).

[Bra11] Aaron R. Bradley. “SAT-based model checking without unrolling”. In: *Proc. VMCAI*. LNCS 6538. Springer, 2011, pp. 70–87 (cit. on p. 39).

[CC77a] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. 1977, pp. 238–252 (cit. on pp. 27, 28, 39, 44, 128).

[CC77b] Patrick Cousot and Radhia Cousot. “Static determination of dynamic properties of recursive procedures”. In: *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*. Ed. by E.J. Neuhold. North-Holland, 1977, pp. 237–277 (cit. on p. 106).

[CC92] Patrick Cousot and Radhia Cousot. “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation”. In: *Proc. PLILP*. LNCS 631. Springer, 1992, pp. 269–295 (cit. on p. 41).

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. “A Survey on Product Operators in Abstract Interpretation”. In: *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*. 2013, pp. 325–336 (cit. on p. 52).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI* (2008) (cit. on p. 35).

[CE82] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Proc. Logic of Programs 1981*. LNCS 131. Springer, 1982, pp. 52–71 (cit. on p. 26).

[CGS09] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. “Decision diagrams for linear arithmetic”. In: *FMCAD*. 2009, pp. 53–60 (cit. on p. 47).

[CH12] Jürgen Christ and Jochen Hoenicke. *Interpolation in SMTLIB 2.0*. 2012. URL: `https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf` (visited on 02/10/2016) (cit. on pp. 149, 156).

[CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. 1978, pp. 84–96 (cit. on pp. 45, 46, 95).

[CH85] Thierry Coquand and Gérard P. Huet. “Constructions: A Higher Order Proof System for Mechanizing Mathematics”. In: *EUROCAL ’85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 1: Invited Lectures*. 1985, pp. 151–184 (cit. on pp. 26, 27).

[Cha+08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. “A TLA+ Proof System”. In: *LPAR*. 2008 (cit. on p. 24).

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. “SMTInterpol: An Interpolating SMT Solver”. In: *Proc. SPIN*. LNCS 7385. Springer, 2012, pp. 248–254 (cit. on p. 150).

[Cim+13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. “The MathSAT5 SMT Solver”. In: *TACAS*. LNCS 7795. Springer, 2013, pp. 93–107 (cit. on p. 150).

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Proc. TACAS*. LNCS 2988. Springer, 2004, pp. 168–176 (cit. on p. 36).

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *CAV*. 2000, pp. 154–169 (cit. on pp. 47, 48, 83, 92).

[Cok13] David R. Cok. *The jSMTLIB User Guide*. 2013. URL: `http://smtlib.github.io/jSMTLIB/jSMTLIBUserGuide.pdf` (visited on 02/10/2016) (cit. on p. 155).

[Coo71] Stephen A. Cook. “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158 (cit. on p. 35).

[Cos+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. “A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs”. In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. 2005, pp. 462–475 (cit. on pp. 57, 80, 120).

[Cra57] William Craig. “Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem”. In: *J. Symb. Log.* 22.3 (1957), pp. 250–268 (cit. on p. 48).

[CSS03] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. “Linear Invariant Generation Using Non-linear Constraint Solving”. In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003*. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 420–432 (cit. on pp. 57, 85).

[CVE13] CVE. *CVE-2014-0160*. Available from MITRE, CVE-ID CVE-2014-0160. 2013. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (cit. on p. 24).

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently computing static single-assignment form and the program dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490 (cit. on p. 35).

[Dij69] Edsger W. Dijkstra. “Structured programming”. circulated privately. Aug. 1969. URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF (cit. on p. 24).

[DKS13] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. “Common Specification Language for Static and Dynamic Analysis of C Programs”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC ’13. ACM, 2013, pp. 1230–1235 (cit. on p. 24).

[DM06] Bruno Dutertre and Leonardo De Moura. “A Fast Linear-Arithmetic Solver for DPLL(T)”. In: (2006), pp. 81–94 (cit. on p. 35).

[Don+11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. “Software Verification Using k-Induction”. In: *SAS*. 2011, pp. 351–368 (cit. on p. 38).

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. “Byte-Precise Verification of Low-Level List Manipulation”. In: *Proc. SAS*. LNCS 7935. Springer, 2013, pp. 215–237 (cit. on p. 48).

[Ela96] Saber N. Elaydi. *An Introduction to Difference Equations*. Springer, 1996 (cit. on p. 86).

[FB14] Alexis Fouilhé and Sylvain Boulmé. “A Certifying Frontend for (Sub)polyhedral Abstract Domains”. In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. 2014, pp. 200–215 (cit. on p. 95).

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. “Annotation inference for modular checkers”. In: *Information Processing Letters* (2001) (cit. on pp. 124, 125).

[FL01] Cormac Flanagan and K. Rustan M. Leino. “Houdini, an Annotation Assistant for ESC/Java”. In: *FME*. 2001, pp. 500–517 (cit. on pp. 124, 153).

[Flo67] Robert W. Floyd. “Assigning meanings to programs”. In: *Mathematical Aspects of Computer Science*. AMS, 1967, pp. 19–32 (cit. on pp. 32, 36).

[Fou] The Eclipse Foundation. *Eclipse C Development Tooling*. URL: `http://eclipse.org/cdt/` (visited on 08/04/2016) (cit. on p. 139).

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995 (cit. on p. 151).

[Gau+07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. “Static Analysis by Policy Iteration on Relational Domains”. In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 237–252 (cit. on pp. 57, 69).

[GBM14] Arie Gurfinkel, Anton Belov, and João Marques-Silva. “Synthesizing Safe Bit-Precise Invariants”. In: *TACAS*. 2014, pp. 93–108 (cit. on p. 124).

[GKN15] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. “SeaHorn: A Framework for Verifying C Programs (Competition Contribution)”. In: *Proc. TACAS*. Springer, 2015 (cit. on p. 120).

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *Proc. PLDI*. ACM, 2005, pp. 213–223 (cit. on p. 35).

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. 2008 (cit. on p. 35).

[GM12] Thomas M. Gawlitza and David Monniaux. “Invariant Generation through Strategy Iteration in Succinctly Represented Control Flow Graphs”. In: *Logical Methods in Computer Science* 8.3 (2012) (cit. on pp. 56, 72).

[GM15] Marco Gario and Andrea Micheli. “PySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms”. In: *Proc. SMT*. 2015 (cit. on p. 155).

[GR06] Denis Gopan and Thomas W. Reps. “Lookahead Widening”. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. 2006, pp. 452–466 (cit. on p. 55).

[Gra91] Philippe Granger. “Static Analysis of Linear Congruence Equalities among Variables of a Program”. In: *TAPSOFT’91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP’91)*. 1991, pp. 169–192 (cit. on p. 47).

[GS07a] Thomas M. Gawlitza and Helmut Seidl. “Precise Fixpoint Computation Through Strategy Iteration”. In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 300–315 (cit. on p. 75).

[GS07b] Thomas M. Gawlitza and Helmut Seidl. “Precise Relational Invariants Through Strategy Iteration”. In: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007*. 2007, pp. 23–40 (cit. on p. 56).

[GS14] Thomas M. Gawlitza and Helmut Seidl. “Numerical invariants through convex relaxation and max-strategy iteration”. In: *Formal Methods in System Design* 44.2 (2014), pp. 101–148 (cit. on pp. 57, 59, 60, 67).

[GS97] Susanne Graf and Hassen Saïdi. “Construction of Abstract State Graphs with PVS”. In: *Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings*. 1997, pp. 72–83 (cit. on pp. 47, 124, 126).

[Gur16] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: `http://www.gurobi.com` (cit. on p. 58).

[Haw+14] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. “Ironclad apps: End-to-end security via automated full-system verification”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 165–181 (cit. on p. 24).

[Hay71] Patrick J. Hayes. *The Frame Problem and Related Problems in Artificial Intelligence*. Tech. rep. Stanford, CA, USA, 1971 (cit. on p. 35).

[Hen+04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. “Abstractions from proofs”. In: *Proc. POPL*. ACM, 2004, pp. 232–244 (cit. on p. 48).

[HG08] Klaus Havelund and Allen Goldberg. “Verified Software: Theories, Tools, Experiments”. In: ed. by Bertrand Meyer and Jim Woodcock. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Verify Your Runs, pp. 374–383 (cit. on p. 24).

[HH12] Nicolas Halbwachs and Julien Henry. “When the Decreasing Sequence Fails”. In: *SAS*. 2012, pp. 198–213 (cit. on p. 55).

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Nested interpolants”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, pp. 471–482 (cit. on p. 118).

[HHS11] John Hawkin, Robert Holte, and Duane Szafron. “Automated action abstraction of imperfect information extensive-form games”. In: *In Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2011 (cit. on p. 55).

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. “PAGAI: A Path Sensitive Static Analyser”. In: *Electr. Notes Theor. Comput. Sci.* 289 (2012) (cit. on pp. 50, 79, 131).

[Hoa71] C. A. R. Hoare. “Procedures and parameters: An axiomatic approach”. In: *Symposium on Semantics of Algorithmic Languages* (1971), pp. 102–116 (cit. on p. 104).

[How60] Ronald Howard. *Dynamic Programming and Markov Processes*. Wiley, 1960 (cit. on p. 55).

[How80] William A. Howard. “The formulas-as-types notion of construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press, 1980, pp. 479–490 (cit. on p. 26).

[IBM10] IBM. *IBM ILOG CPLEX Optimizer*. 2010. URL: http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ (cit. on p. 58).

[Kar16] George E. Karpenkov. “LPI: Software verification with local policy iteration (competition contribution)”. In: *TACAS*. 2016 (cit. on p. 29).

[KBF16] George E. Karpenkov, Dirk Beyer, and Karlheinz Friedberger. “JavaSMT: a unified interface for SMT solvers in Java”. In: *VSTTE*. 2016 (cit. on pp. 29, 149).

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Proc. POPL. Boston, Massachusetts: ACM, 1973, pp. 194–206 (cit. on pp. 27, 39).

[Kin76] James C. King. “Symbolic execution and program testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394 (cit. on pp. 26, 35).

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Bibl. Matematica. Amsterdam: North-Holland, 1952 (cit. on pp. 40, 42, 59).

[KM16] George E. Karpenkov and David Monniaux. “Formula Slicing: Inductive Invariants from Preconditions”. In: *HVC*. 2016 (cit. on pp. 28, 123).

[KMW16] George E. Karpenkov, David Monniaux, and Philipp Wendler. “Program Analysis with Local Policy Iteration”. In: *Proc. VMCAI*. LNCS 9583. Springer, 2016, pp. 127–146 (cit. on pp. 28, 55, 153).

[Kom+13] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. “Automatic Abstraction in SMT-Based Unbounded Software Model Checking”. In: *CAV*. 2013, pp. 846–862 (cit. on pp. 106, 120, 124).

[Kri63] Saul Kripke. “Semantical Considerations on Modal Logic”. In: *Acta Phil. Fennica* 16 (1963), pp. 83–94 (cit. on p. 26).

[Kro+08] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. “Loop Summarization Using Abstract Transformers”. In: *ATVA*. 2008, pp. 111–125 (cit. on p. 93).

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008 (cit. on pp. 38, 76).

[KT11] Temesghen Kahsai and Cesare Tinelli. “PKind: A parallel k-induction based model checker”. In: *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011*. 2011, pp. 55–62 (cit. on pp. 38, 147).

[Lec08] Thierry Lecomte. “Safe and Reliable Metro Platform Screen Doors Control/Command Systems”. In: *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*. 2008, pp. 430–434 (cit. on p. 23).

[Lei10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 2010, pp. 348–370 (cit. on p. 24).

[LQ09] Shuvendu K. Lahiri and Shaz Qadeer. “Complexity and Algorithms for Monomial and Clausal Predicate Abstraction”. In: *CADE*. 2009, pp. 214–229 (cit. on pp. 124, 135).

[Luc+16] Kasper Søe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. “JDart: A Dynamic Symbolic Analysis Framework”. In: *Proc. TACAS*. LNCS 9636. 2016 (cit. on p. 155).

[Man69] Zohar Manna. “The correctness of programs”. In: *Journal of Computer and Systems Sciences* 3(2) (1969), pp. 119–127 (cit. on p. 37).

[MB08] Leonardo M. de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proc. TACAS*. LNCS 4963. 2008, pp. 337–340 (cit. on pp. 35, 131, 150).

[McM03] Kenneth L. McMillan. “Interpolation and SAT-Based Model Checking”. In: *Proc. CAV*. LNCS 2725. Springer, 2003, pp. 1–13 (cit. on p. 48).

[McM05] Kenneth L. McMillan. “An interpolating theorem prover”. In: *Theor. Comput. Sci.* 345.1 (2005), pp. 101–121 (cit. on p. 88).

[McM06] Kenneth L. McMillan. “Lazy Abstraction with Interpolants”. In: *CAV*. 2006, pp. 123–136 (cit. on pp. 48, 88, 131, 160).

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44 (cit. on p. 24).

[MG11] David Monniaux and Laure Gonnord. “Using Bounded Model Checking to Focus Fixpoint Iterations”. In: *SAS*. 2011, pp. 369–385 (cit. on pp. 49, 50, 96, 131).

[Min06] Antoine Miné. “The octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100 (cit. on pp. 45, 46, 84, 87).

[Mon10] David Monniaux. “Automatic Modular Abstractions for Template Numerical Constraints”. In: *Logical Methods in Computer Science* (June 2010). arXiv: 1005.4844 (cit. on p. 57).

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991 (cit. on p. 25).

[MR05] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers”. In: *Proc. ESOP*. LNCS 3444. Springer, 2005, pp. 5–20 (cit. on p. 47).

[MS04] Markus Müller-Olm and Helmut Seidl. “Precise interprocedural analysis through linear algebra”. In: *POPL*. 2004, pp. 330–341 (cit. on p. 106).

[MS14] David Monniaux and Peter Schrammel. “Speeding Up Logico-Numerical Strategy Iteration”. In: *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014*. 2014, pp. 253–267 (cit. on p. 57).

[New+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73 (cit. on p. 23).

[New14] Chris Newcombe. "Why Amazon Chose TLA+". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. 2014, pp. 25–39 (cit. on p. 23).

[Niv56] Ivan Niven. *Irrational Numbers*. The Mathematical Association of America, 1956 (cit. on p. 86).

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999 (cit. on pp. 105, 106).

[Oh+14] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. "Selective context-sensitivity guided by impact pre-analysis". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, p. 49 (cit. on p. 84).

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *Proc. Symposium on Programming*. LNCS 137. Springer, 1982, pp. 337–351 (cit. on p. 26).

[RG13] Pierre Roux and Pierre-Loïc Garoche. "Integrating Policy Iterations in Abstract Interpreters". In: *ATVA*. 2013, pp. 240–254 (cit. on p. 57).

[RG14] Pierre Roux and Pierre-Loïc Garoche. "Computing Quadratic Invariants with Min- and Max-Policy Iterations: A Practical Comparison". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. 2014, pp. 563–578 (cit. on pp. 57, 80).

[RHS95] Thomas W. Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Data-Flow Analysis via Graph Reachability". In: *Proc. POPL*. ACM, 1995, pp. 49–61 (cit. on p. 106).

[Ric53] Henry G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: 74.2 (1953), pp. 358–366 (cit. on p. 25).

[RSY04] Thomas W. Reps, Mooly Sagiv, and Greta Yorsh. "Symbolic Implementation of the Best Transformer". In: *VMCAI*. 2004 (cit. on pp. 124, 125).

[RTC11] Special Committee of RTCA. *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. 2011 (cit. on p. 23).

[Rüm12] Philipp Rümmer. "E-Matching with Free Variables". In: *Proc. LPAR*. LNCS 7180. 2012, pp. 359–374 (cit. on p. 150).

[San+06] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. "Static Analysis in Disjunctive Numerical Domains". In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. 2006, pp. 3–17 (cit. on pp. 47, 160).

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 28–28 (cit. on p. 24).

[SGS14] Helmut Seidl, Thomas M. Gawlitza, and Martin S. Schwarz. “Parametric Strategy Iteration”. In: *6th International Symposium on Symbolic Computation in Software Science, SCSS 2014, Gammarth, La Marsa, Tunisia, December 7-8, 2014*. 2014, pp. 62–76 (cit. on p. 84).

[Sha79] Adi Shamir. “A Linear Time Algorithm for Finding Minimum Cutsets in Reducible Graphs”. In: *SIAM J. Comput.* 8.4 (1979), pp. 645–655 (cit. on pp. 50, 56).

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: Data Race Detection in Practice”. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA ’09. New York, New York, USA: ACM, 2009, pp. 62–71 (cit. on p. 24).

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *Proc. ESEC/FSE*. ACM, 2005, pp. 263–272 (cit. on p. 26).

[SMM12] Pavel Shved, Mikhail U. Mandrykin, and Vadim S. Mutilin. “Predicate Analysis with BLAST 2.7 - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012*. 2012, pp. 525–527 (cit. on p. 79).

[SP81] Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data dalow analysis”. In: *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981, pp. 189–233 (cit. on p. 104).

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. “Scalable Analysis of Linear Systems Using Mathematical Programming”. In: *VMCAI*. 2005, pp. 25–41 (cit. on pp. 38, 45, 46).

[ST15] Roberto Sebastiani and Patrick Trentin. “OptiMathSAT: A Tool for Optimization Modulo Theories.” In: *Proc. CAV*. LNCS 9206. 2015 (cit. on p. 150).

[Sto76] Larry J. Stockmeyer. “The polynomial-time hierarchy”. In: *Theoretical Computer Science* 3.1 (1976), pp. 1–22 (cit. on p. 135).

[Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications”. In: 5 (1955), pp. 285–309 (cit. on pp. 41, 59).

[Tur36] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265 (cit. on p. 25).

[Tur49] Alan M. Turing. “Checking a Large Routine”. In: *Report on a Conference on High Speed Automatic Calculating Machines*. Cambridge Univ. Math. Lab., 1949, pp. 67–69 (cit. on p. 36).

[VB96] Lieven Vandenberghe and Stephen Boyd. “Semidefinite Programming”. In: *SIAM Rev.* 38.1 (Mar. 1996), pp. 49–95 (cit. on p. 80).

[Zal] Michal Zalewski. *American fuzzy lop*. URL: `http://lcamtuf.coredump.cx/afl/` (visited on 08/04/2016) (cit. on p. 24).

[Zha+14] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. “Hybrid top-down and bottom-up interprocedural analysis”. In: *PLDI*. 2014, pp. 249–258 (cit. on p. 106).