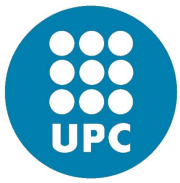


Universitat Politècnica de Catalunya
BarcelonaTech
Departament de Ciències de la Computació
PhD. in Computing

Automatic Program Analysis using Max-SMT

Daniel Larraz Hurtado

PhD. Thesis

A dissertation submitted to obtain the qualification of Doctor in Computer Science from the Universitat Politècnica de Catalunya

Advisor:
Dr. Albert Rubio Gimeno

Barcelona, June 2015

Abstract

This thesis addresses the development of techniques to build fully-automatic tools for analyzing sequential programs written in imperative languages like C or C++. In order to do the reasoning about programs, the approach taken in this thesis follows the constraint-based method used in program analysis. The idea of the constraint-based method is to consider a template for candidate invariant properties, e.g., linear conjunctions of inequalities. These templates involve both program variables as well as parameters whose values are initially unknown and have to be determined so as to ensure invariance. To this end, the conditions on inductive invariants are expressed by means of constraints (hence the name of the approach) on the unknowns. Any solution to these constraints then yields an invariant. In particular, if linear inequalities are taken as target invariants, conditions can be transformed into arithmetic constraints over the unknowns by means of Farkas' Lemma. In the general case, a Satisfiability Modulo Theories (SMT) problem over non-linear arithmetic is obtained, for which effective SMT solvers exist.

One of the novelties of this thesis is the presentation of an optimization version of the SMT problems generated by the constraint-based method in such a way that, even when they turn out to be unsatisfiable, some useful information can be obtained for refining the program analysis. In particular, we show in this work how our approach can be exploited for proving termination of (sequential) programs, disproving termination of non-deterministic programs, and do compositional safety verification. Besides, an extension of the constraint-based method to generate universally quantified array invariants is also presented.

Since the development of practical methods is a priority in this thesis,

all the techniques have been implemented and tested with examples coming from academic and industrial environments.

The main contributions of this thesis are summarized as follows:

- A new constraint-based method for the generation of universally quantified invariants of array programs. We also provide extensions of the approach for sorted arrays.

- A novel Max-SMT-based technique for proving termination. Thanks to expressing the generation of a ranking function as a Max-SMT optimization problem where constraints are assigned different weights, quasi-ranking functions –functions that almost satisfy all conditions for ensuring well-foundedness– are produced in a lack of ranking functions. Moreover, Max-SMT makes it easy to combine the process of building the termination argument with the usually necessary task of generating supporting invariants.

- A Max-SMT constraint-based approach for proving that programs do not terminate. The key notion of the approach is that of a quasi-invariant, which is a property such that if it holds at a location during execution once, then it continues to hold at that location from then onwards. Our technique considers for analysis strongly connected subgraphs of a program's control flow graph and thus produces more generic witnesses of non-termination than existing methods. Furthermore, it can handle programs with unbounded non-determinism.

- An automated compositional program verification technique for safety properties based on quasi-invariants. For a given program part (e.g., a single loop) and a postcondition, we show how to, using a Max-SMT solver, an inductive invariant together with a precondition can be synthesized so that the precondition ensures the validity of the invariant and that the invariant implies the postcondition. From this, we build a bottom-up program verification framework that propagates preconditions of small program parts as postconditions for preceding program parts. The method recovers from failures to prove validity of a precondition, using the obtained intermediate results to restrict the search space for further proof attempts.

Acknowledgements

I want to express here my gratitude to all those people that have made this thesis possible. First and foremost, I would like to thank my advisor Albert Rubio for his encouragement, patience, and guidance throughout these years. Besides, I am also indebted to Enric Rodríguez and Albert Oliveras for sharing with me their knowledge and inspired thoughts during all this time.

In addition to my closest collaborators, I want to extend my gratitude to Marc Brockschmidt and Kaustubh Nimkar for contributing their time and ideas to our respective joint works.

My grateful thanks to Cristina Borralleras and José Miguel Rivero for joining us in developing the VeryMax tool.

I would also like to thank the Jutge.org team for providing us with benchmarks, Byron Cook for his helpful comments, and for giving us access to T2 and their benchmarks, and all the anonymous reviewers for their valuable comments and suggestions.

Finally, I wish to thank my parents, Asun and Pedro, and my partner, Natalia, for their support and encouragement throughout these years.

This research has been supported by Spanish MINECO under the FPI grant BES-2011-044621 (project code TIN2010-21062-C02-01).

Contents

Chapter 1

Introduction

At the same time that the complexity of computer systems is growing rapidly, today's information society is becoming increasingly dependent on such systems. Consequently, exploring techniques to produce reliable software is an issue of increasing importance.

Although software engineering methodologies try to overcome human limitation for managing complexity, software development is still currently a time-consuming, costly, and error-prone activity. Because of this, the construction of computer-aided tools that assist in the design and the implementation of computer systems is an important challenge.

Formal specification provides a means of describing informal software requirements in a rigorous and high-level way reducing the ambiguity of the problem domain. They can be used not only to aid the design and the implementation of computer systems, but also to verify their correctness. In particular, the formal verification of a program consists in proving that its semantics, what the program executions actually do, satisfies its specification, what the program executions are intended to do. In spite of its usefulness, formal methods are not widely used to verify programs in the industrial software environment, and its application is usually restricted to safety-critical software. One of the reasons for its uncommon use is that, even in the cases where automatic proof checkers are available, programmers are often required to annote the source code with auxiliary information like loop invariants and ranking functions, which are necessary to prove the program correctness.

With the aim of reversing this situation, this thesis addresses the de-

velopment of techniques to build fully-automatic tools that are capable of: modeling program semantics from source code, discovering key properties of interest, and using them to verify specifications and prove termination of programs.

In order to do the reasoning about programs, the approach taken in this thesis follows the constraint-based method [Colón et al., 2003; Bradley et al., 2005] used in program analysis. The idea is to consider templates for candidate invariant properties, such as (conjunctions of) linear inequalities. These templates contain both the program variables $\mathcal{V}$ as well as template variables $\mathcal{X}$, whose values have to be determined to ensure the required properties. To this end, the conditions on inductive invariants are expressed by means of *constraints* of the form $\exists\mathcal{X}.\forall\mathcal{V}.\mathcal{F}(\mathcal{X},\mathcal{V})$. Any solution to these constraints then yields an invariant. In the case of linear arithmetic, Farkas' Lemma [Schrijver, 1998] is often used to handle the quantifier alternation in the generated constraints. Intuitively, it allows one to transform $\exists\forall$ problems encountered in invariant synthesis into $\exists$ problems[1]. In the general case, a Satisfiability Modulo Theories (SMT) problem [Biere et al., 2009] over non-linear arithmetic is obtained, for which effective SMT solvers exist [Borralleras et al., 2012; Jovanović and De Moura, 2012; Larraz et al., 2014b].

One of the novelties and main contributions of this thesis is the presentation of an optimization version of the SMT problems generated by the constraint-based method in such a way that, even when they turn out to be unsatisfiable, some useful information can be obtained for refining the program analysis. In particular, we show in this work how our approach can be exploited for proving termination of (sequential) programs [Larraz et al., 2013a], disproving termination of non-deterministic programs [Larraz et al., 2014a], and do compositional safety verification [Brockschmidt et al., 2015]. Besides, an extension of the constraint-based method to generate universally quantified array invariants [Larraz et al., 2013b] is also presented.

As the development of practical methods is a priority in this thesis, all the techniques have been implemented and tested with examples coming from academic and industrial environments[2]. The author of this thesis was the developer of a first prototype called CppInv, which applies the techniques

[1] The reader is referred to Section 3.2.2 for further information.
[2] See Sections 3.3.5, 4.3.6, 4.4.6, and 5.4 for more details.

described in Chapters 3 and 4 to programs written in a subset of C++. A new tool called VeryMax, that implements the compositional analysis framework explained in Chapter 5, is currently developed by a group of researchers (including the author of the thesis) as the continuation of the line of work started with CppInv.

1.1 Thesis Outline

The thesis document is organized as follows: Chapter 2 describes the theoretical frameworks used in this thesis to model programs, and to solve all program analysis problems. Firstly, the chapter introduces the Satisfiability Modulo Theories (SMT) problem, and its optimization version, the Max-SMT problem, which compose the bases of the solving technology employed in this work. They are based on the known *propositional satisfiability* (SAT) problem, which consists in determining whether a propositional formula is *satisfiable*, i.e., if it has a *model*: an assignment of Boolean values to variables that satisfies the formula.

On the other hand, the *Satisfiability Modulo Theories* (SMT) problem, one of the extensions of SAT, consists in deciding the satisfiability of a given quantifier-free first-order formula with respect to a background theory. In this setting, a model is an assignment of values from the theory to variables that satisfies the formula. In this thesis, we will use the theories of *quantifier-free linear arithmetic*, where literals are linear inequalities, and the more general theory of *quantifier-free non-linear arithmetic*, where literals are polynomial inequalities.

Another extension of SAT is *Max-SAT* [Biere et al., 2009], which generalizes SAT to finding an assignment such that the number of satisfied clauses in a given formula F is maximized. This problem in turn can be generalized to the *weighted partial Max-SAT* problem, where some clauses in F are *soft clauses* with an assigned weight, and the others are *hard clauses*. Here, we look for a model of the hard clauses that minimizes the sum of the weights of the satisfied soft clauses.

The last extension presented is *Max-SMT* [Nieuwenhuis and Oliveras, 2006; Larraz et al., 2014b], which combines Max-SAT and SMT, and is derived from SMT analogously to how Max-SAT is derived from SAT. So in a *(weighted partial) Max-SMT* problem a formula is of the form $H_1 \wedge \ldots \wedge$

$H_n \wedge [S_1, \omega_1] \wedge \ldots \wedge [S_m, \omega_m]$, where the hard clauses H_i and the soft clauses S_j (with weight ω_j) are disjunctions of literals over a background theory, and the aim is to find a model of the hard clauses that minimizes the sum of the weights of the satisfied soft clauses.

Finally, in Chapter 2 *transition systems* are introduced as the programming model used through this thesis to represent programs, and two simple applications of the use of SMT solvers in program analysis are described.

Chapter 3 starts presenting the invariant inference problem, and reviewing the main approaches to the problem. Discovering invariants, assertions over the program variables that remain true whenever the location is reached, is crucial for program verification. However, it is a tedious and, sometimes, difficult task for the programmer. For this reason, the design of techniques to discover automatically invariants attracted researchers attention from the beginning of the field. Despite this, it has not been until recently, with the lastest technological advances in constraint solving and theorem proving, that practical methods have been applied to programs of interest.

Chapter 3 continues describing the constraint-based method [Colón et al., 2003] to generate invariants over scalar program variables in detail. Then, our extension of the method for generating universally quantified array invariants is presented. Unlike other techniques, our technique does not require extra predicates nor assertions. It does not need the user to provide a template either, but it can take advantage of hints by partially instantiating the global template considered here. This work corresponds to the following paper [Larraz et al., 2013b].

In Chapter 4 we show how Max-SMT can be used in constraint-based program termination proving originated in [Bradley et al., 2005]. Thanks to expressing the generation of a ranking function as a Max-SMT optimization problem where constraints are assigned different weights, *quasi-ranking functions* –functions that almost satisfy all conditions for ensuring well-foundedness– are produced in a lack of ranking functions. By means of trace partitioning, this allows our method to progress in the termination analysis where other approaches would get stuck. Moreover, Max-SMT makes it easy to combine the process of building the termination argument with the usually necessary task of generating supporting invariants. The technique was presented in [Larraz et al., 2013a].

We then show how Max-SMT-based invariant generation can also be exploited for proving non-termination of programs. The construction of the proof of non-termination is guided by the generation of *quasi-invariants* – properties such that if they hold at a location during execution once, then they will continue to hold at that location from then onwards. The check that quasi-invariants can indeed be reached is then performed separately. Our technique considers for analysis strongly connected subgraphs of a program's control flow graph and thus produces more generic witnesses of non-termination than existing methods. Furthermore, it can handle programs with unbounded non-determinism and is more likely to converge than previous approaches. This method can be found in [Larraz et al., 2014a].

In Chapter 5 we present an automated compositional program verification technique for safety properties based on *quasi-invariants*. For a given program part (e.g., a single loop) and a postcondition φ, we show how to synthesize an inductive invariant together with a precondition such that the precondition ensures validity of the invariant and the invariant implies φ. From this, we build a bottom-up program verification framework that propagates preconditions of small program parts as postconditions for preceding program parts. The method recovers from failures to prove the validity of a precondition, using the obtained intermediate results to restrict the search space for further proof attempts.

As only small program parts need to be handled at a time, our method is scalable and distributable. The derived conditions can be viewed as implicit contracts between different parts of the program, and thus enable an incremental program analysis. The techniques presented in this chapter have been included in [Brockschmidt et al., 2015].

Chapter 2

Preliminaries

This chapter is devoted to the theoretical frameworks used in this thesis to model programs, and to solve all program analysis problems. In Section 2.1 we present the Satisfiability Modulo Theories (SMT) problem, and its optimization version, the Max-SMT problem, which compose the bases of the solving technology employed in this work. Later, in Section 2.2 we introduce transition systems, the programming model used through this thesis to represent programs. Finally, in order to illustrate the use of the concepts introduced in the chapter, in Section 2.3 we present two simple applications of the use of SMT solvers in program analysis.

2.1 SAT and SMT solving

Many of the problems arising in applications of automatic reasoning can be formulated as the problem of checking the satisfiability of a formula in a certain logic. For many logics of interest, this problem is undecidable. For satisfiability of first-order formulas, for example, several efficient *semi-decision procedures* exist, which can prove the unsatisfiability of a formula in finite time, but may run forever when their input formula is satisfiable. For other cases (e.g., certain logics with built-in integers) not even semi-decision procedures can exist.

2.1.1 Propositional satisfiability

Probably the easiest satisfiability problem one can think of is that of propositional satisfiability (SAT). In this case, the atomic formulas are nothing but syntactic symbols, which are combined by means of boolean connectives to construct the formula. A boolean formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses, each clause being a disjunction of literals, and each literal being a propositional variable or its negation. Since any formula can be converted into CNF format in a *satisfiability preserving* polynomial-time process [Tseitin, 1983; Plaisted and Greenbaum, 1986], it is usually assumed that all formulas are in CNF format.

Example 2.1. Consider the following boolean formula $(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_1 \wedge (x_2 \vee \neg x_3)$ in CNF format. It is satisfiable: the assignment $\{x_1, x_2, \neg x_3\}$ (i.e., setting x_1 and x_2 to true, and x_3 to false) is a model of it. If the clause $\neg x_1 \vee x_3$ is added, the conjunction becomes *unsatisfiable*. A complete *SAT solver* is a tool that, given a set of clauses, either finds a model for it or reports unsatisfiability.

Most state-of-the-art SAT solvers [Moskewicz et al., 2001; Goldberg and Novikov, 2002; Eén and Sörensson, 2004; Ryan, 2004; Biere, 2008] use *Conflict-driven Clause Learning*, and are originally based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam, 1960; Davis et al., 1962] (see, e.g., [Nieuwenhuis et al., 2006] for details and more references).

Despite the NP-completeness of the problem, a lot of work has been done in developing algorithms that can handle larger and larger practical problems. The past few years have seen enormous progress in the development of SAT solvers (see `www.satlive.org`) and there is no doubt that they have now grown out of academic curiosity to become a viable industrial strength reasoning and deduction engine for production tools. More and more problems are efficiently encoded into propositional logic, which increases the interest in developing such efficient SAT solvers.

There exist several optimization versions of the SAT problem. In *Max-SAT*, the aim is to find a model that maximizes the number of satisfied clauses. In *Partial Max-SAT* the input consists of two sets of clauses, the *hard* ones and *soft* ones, and the problem is to find a model for the hard clauses that maximizes the number of satisfied soft clauses. In *Weighted*

(Partial) Max-SAT each soft clause has a *weight* and the aim is to minimize the sum of the weights of the falsified soft clauses.

A very naive approach to (unweighted) Max-SAT solving consists, first of all, in extending each original clause C_i to obtain clause $C_i \vee b_i$, where b_i is a fresh variable. Then, any model of the formula F_k, consisting of all the extended clauses plus a CNF encoding of the cardinality constraint $\sum b_i \leq k$, is an assignment that falsifies at most k original clauses. Hence, the goal is to find k such that F_k is satisfiable but F_{k-1} is not. Such a k can be found with different methods: using binary search, or starting with $k = 0$ and increasing it while F_k is unsatisfiable, or starting with k equal to the number of clauses and decreasing it until F_k is unsatisfiable.

Another less naive way to tackle Max-SAT is via branch-and-bound techniques. Roughly speaking, once an assignment falsifying k clauses has been found, these techniques look for an assignment falsifying less than k clauses. As soon as one can infer that the current partial assignment cannot be extended to one with the aforementioned property, it is discarded and another assignment is tried. The key point is to develop powerful but efficient pruning techniques that allow one to discard partial assignments as soon as possible; some examples are [Alsinet et al., 2008; Heras et al., 2008; Lin et al., 2008; Pipatsrisawat et al., 2008].

Example 2.2. Consider the following set of clauses $\{x_1 \vee b_1,\ \neg x_1 \vee x_3 \vee b_2,\ x_2 \vee \neg x_3 \vee b_3,\ \neg x_1 \vee \neg x_2 \vee \neg x_3\}$ where a clause without a b_i variable is considered *hard*, otherwise *soft*. The assignment $\{x_1, \neg x_2, x_3, \neg b_1, \neg b_2, b_3\}$ is an optimal solution. Now suppose clauses are weighted: $[b_1, 1]$, $[b_2, 2]$, and $[b_3, 3]$. In that case the previous assignment is no more an optimal solution, the only one is the assignment $\{\neg x_1, x_2, x_3, b_1, \neg b_2, \neg b_3\}$ with weight 1.

2.1.2 Satisfiability Modulo Theories

The SAT problem can be considered an instance of a more general problem, namely, the *Satisfiability Modulo Theories* (SMT) problem [Biere et al., 2009], which consists in deciding the satifisfiability of arbitrary boolean formulas whose atoms belong to a certain theory.

The richer the theory is, the easier it is to express the desired properties. For example, when reasoning about software, it helps to work with logics including standard data types, such as arrays, lists, bit-vectors or

trees. Arithmetic operators are also really helpful, and hence the possibility of using Presburger arithmetic, linear and non-linear arithmetic, or other fragments is highly desirable.

Example 2.3. Consider the following SMT formula over linear arithmetic $(y = x + 1 \wedge y \leq z \wedge z < x)$. Clearly, the formula is unsatisfiable, that is, there is no real numbers x, y and z that satisfy the formula.

During the last years many successively more sophisticated techniques for deciding satisfiability modulo theories have been developed, most of which can be classified as being *eager* or *lazy*.

In the eager approaches the input formula is translated, in a single satisfiability-preserving step, into a propositional formula, which is checked by a SAT solver for satisfiability. The lazy approaches [Armando et al., 2000; De Moura et al., 2002; Audemard et al., 2002; Barrett et al., 2002; Flanagan et al., 2003] instead abstract each atom of the input formula by a distinct propositional variable, use a SAT solver to find a propositional model of the formula, and then check that model against the theory. Models that are incompatible with the theory are discarded from later consideration by adding a proper *lemma* to the original formula. This process is repeated until a model compatible with the theory is found or all possible propositional models have been explored.

The eager approach allows one to use existing SAT solvers *as-is* and leverage their performance and capacity improvements over time. On the other hand, the loss of the high-level semantics of the underlying theories means that the SAT solver has to work a lot harder than necessary to discover *obvious* facts like, for instance, the commutative property of reals.

There also exist optimization versions of the SMT problem. The problem of Max-SMT merges Max-SAT and SMT, and is defined from SMT analogously to how Max-SAT is derived from SAT. E.g., the *Weighted Max-SMT* problem consists in, given a weighted formula, to find an assignment that minimizes the sum of the weights of the falsified clauses in the background theory. Henceforth, we will refer to the Weighted Max-SMT problem plainly as the Max-SMT problem.

2.1.3 Barcelogic SMT/Max-SMT solver

All the methods developed for this thesis have been implemented, some of them in a tool called CppInv (Chapters 3 and 4), and other ones in a tool called VeryMax (Chapter 5). These tools parse programs written in a subset of C++, abstract their program semantics and discover linear program properties that holds at some program locations. In order to infer such properties, the inference problem is encoded into an SMT/Max-SMT problem over non-linear arithmetic, and then sent to the Barcelogic solver [Bofill et al., 2008].

Solving non-linear arithmetic constraint over the integers is undecidable. The situation is not much better when considering the reals since, although the problem is decidable as it was shown in [Tarski, 1953], using the related algorithms in practice is unfeasible due to their complexity.

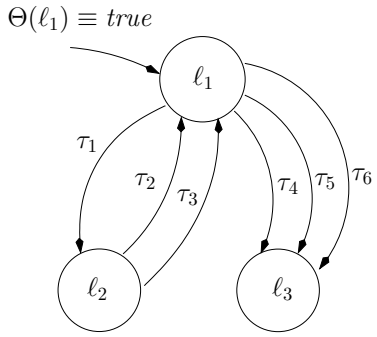
Therefore, all methods used in practice for both integer or real solution domains are incomplete and are focused on either proving satisfiability or proving unsatisfiability. In this thesis we are particularly interested on the former because each found solution represents a new discovered property. That is the reason we choose the Barcelogic solver. Barcelogic has proved to be very effective in finding solutions [Borralleras et al., 2012]; e.g., it won the division of quantifier-free non-linear integer arithmetic (QF NIA) in the 2009 edition of the SMT-COMP competition (`www.smtcomp.org/2009`), and since then (as of 2013) no other competing solver in this division had solved as many problems.

The Max-SMT(NA) solver for mixed non-linear arithmetic in Barcelogic [Larraz et al., 2014b] improves and extends the techniques presented in [Borralleras et al., 2012] for solving SMT(NIA) problems. This is achieved by allowing integer and real variables in the underlying linear arithmetic solver, and wrapping this solver with a branch-and-bound scheme for optimization [Nieuwenhuis and Oliveras, 2006].

2.2 Modelling programs

SAT/SMT solvers are useful tools for program analysis. But, in order to reason about programs, it is necessary to model program semantics using an abstraction based on some logic. Transitions systems are a convenient framework to describe imperative programs.

```
int main() {
    int x, y, z;
ℓ1:  while (x + 1 = y && y ≤ z) {
    ℓ2:  if (z < x) x++;
         else z--;
    }
    ℓ3:
}
```



$$
\begin{array}{llll}
\rho_{\tau_1}: & x+1=y,\ y\le z, & x'=x, & y'=y,\ z'=z \\
\rho_{\tau_2}: & z<x, & x'=x+1, & y'=y,\ z'=z \\
\rho_{\tau_3}: & z\ge x, & x'=x, & y'=y,\ z'=z-1 \\
\rho_{\tau_4}: & x+1<y, & x'=x, & y'=y,\ z'=z \\
\rho_{\tau_5}: & x+1>y, & x'=x, & y'=y,\ z'=z \\
\rho_{\tau_6}: & y>z, & x'=x, & y'=y,\ z'=z
\end{array}
$$

Figure 2.4. Program and its transition system.

2.2.1 Transition Systems

A transition system $\mathcal{S} = (\mathcal{V}, \mathcal{L}, \Theta, \mathcal{T})$ consists of a tuple of *variables* $\mathcal{V}$, a set of *locations* $\mathcal{L}$, a map Θ from locations to formulas characterizing the initial values of the variables, and a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a triple (ℓ, ℓ', ρ), where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and ρ is the *transition relation*: a formula over the program variables $\mathcal{V}$ and their primed versions $\mathcal{V}'$, which represent the values of the variables after the transition. In general, to every formula φ over the program variables $\mathcal{V}$ we associate a formula φ' which is the result of replacing every variable x_i in φ by its corresponding primed version x_i'.

The logic chosen to model the transition relations depends on the kind of properties one wants to analyze. From now on, we assume that variables take *integer* values and programs are *linear*, i.e., the initial conditions Θ and transition relations ρ are described as conjunctions of linear inequalities.

See Fig. 2.4 for an example of a program together with a corresponding representation as a transition system.

2.2.2 States and Executions

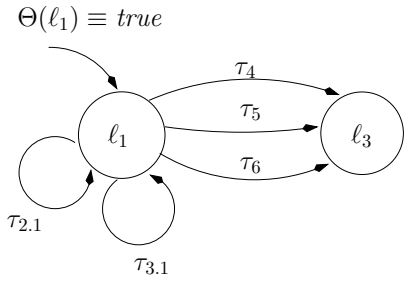
A *state* is a pair (ℓ, σ) consisting of a location $\ell \in \mathcal{L}$ and an assignment σ of a value to each of the variables in $\mathcal{V}$. To ease the reading, we will sometimes refer to (ℓ, σ) as the state σ at location ℓ, and we may omit the location if it is clear from the context. A state (ℓ, σ) is *initial* if $\sigma \models \Theta(\ell)$. We denote an *evaluation step* with transition $\tau = (\ell, \ell', \rho)$ by $(\ell, \sigma) \to_\tau (\ell', \sigma')$, where the assignments σ, σ' satisfy the transition relation ρ. We use $\to_{\mathcal{S}}$ if we do not care about the executed transition, and $\to_{\mathcal{S}}^*$ to denote the transitive-reflexive closure of $\to_{\mathcal{S}}$. A *computation* is a sequence of states (ℓ_0, σ_0), (ℓ_1, σ_1), ... such that $\sigma_0 \models \Theta(\ell_0)$, and for each pair of consecutive states there exists $\tau_i \in \mathcal{T}$ satisfying $(\ell_i, \sigma_i) \to_{\tau_i} (\ell_{i+1}, \sigma_{i+1})$. A state (ℓ, σ) is *reachable* if there exists a computation ending at (ℓ, σ). A transition system is *terminating* if all its computations are finite, and *non-terminating* otherwise.

A transition $\tau = (\ell, \ell', \rho)$ is *disabled* if it can never be executed, i.e., if for all reachable state (ℓ, σ), there does not exist any σ' such that $(\sigma, \sigma') \models \rho$. A transition τ is called *finitely executable* if in any computation, τ is only executed a finite number of times (in particular, if τ is disabled). Otherwise, i.e., if there exists a computation where τ is executed infinitely, we say that τ is *infinitely executable*.

2.2.3 Locations and Cutsets

When modeling a program as a transition system, it is necessary to select a set of locations that are associated to points in the original program. For instance, in the example of Fig. 2.4, location ℓ_1 maps to the beginning of the `while` loop, location ℓ_2 is associated with the beginning of the `if` statement and location ℓ_3 maps to the end of the main program. But that is not the only way to build the set of locations. In many cases, intermediate locations like ℓ_2 can be ignored merging the incoming transitions with the transitions that exit the location (see Fig. 2.6). That can be the case, for instance, if one is only interested in properties that hold at the beginning of the loop. In particular, in many applications it is important to construct a *cutset*, a set of locations (called *cutpoints*) such that every cyclic path of the program contains a location within the set.

The number of cutpoints has a strong influence on the complexity of the program analyses. As will be seen in later chapters, the size of the generated



$$
\begin{array}{lllllll}
\rho_{\tau_{2.1}}: & x+1=y, & y \leq z, & z < x, & x' = x+1, & y' = y, & z' = z \\
\rho_{\tau_{3.1}}: & x+1=y, & y \leq z, & z \geq x, & x' = x, & y' = y, & z' = z-1 \\
\rho_{\tau_4}: & x+1<y, & & & x' = x, & y' = y, & z' = z \\
\rho_{\tau_5}: & x+1>y, & & & x' = x, & y' = y, & z' = z \\
\rho_{\tau_6}: & y>z, & & & x' = x, & y' = y, & z' = z
\end{array}
$$

Figure 2.6. Simplified transition system for program of Fig. 2.4.

SMT problems depends on the number of template properties associated to each cutpoint. Therefore, it seems desirable to look for cutsets of small cardinality. Although the problem of finding a minimum cutset for an arbitrary directed graph is NP-complete [Karp, 1972], a linear time algorithm is known [Shamir, 1979] for *practical* flowcharts of programs. Nevertheless, one also have to take into account that reducing the number of locations can entail, in general, an increment of the number of transitions. Therefore, in practice, the choice of a set of locations depends on selecting the appropriate threshold for the resulting number of locations and transitions.

2.3 SMT and program analysis

In order to illustrate the use of the concepts introduced in this chapter, and the close relationship between them, in what follows we present two simple applications of the use of SMT solvers in program analysis.

The first example of application is the detection of an *unfeasible transition*, which allows one to simplify a transition system for subsequent analyses, and detect simple cases of unreachable locations.

Example 2.5. Consider the transition system of Fig. 2.6 which models program of Fig. 2.4 using only ℓ_1 and ℓ_3 as locations.

Note that there are two transitions ($\tau_{2.1}$ and $\tau_{3.1}$) that cycle back to the entry of the `while` loop (location ℓ_1), but only the one that passes through

the `else` branch of the conditional statement ($\tau_{3.1}$) is executable. That happens because the conditions of the `while` and the `if` statements are incompatible. In order to detect it we can send two queries to the solver about the satisfiability of the SMT formulas that models the transitions (its corresponding transition relations $\rho_{2.1}$ and $\rho_{3.1}$). Since the transition relation for the `if` branch ($\tau_{2.1}$) is unsatisfiable (see example 2.3), we can conclude that the transition will never be executed and, therefore, the x increment is unreachable.

Another application is to check whether an assertion at some location of the program holds whenever the control flow reaches the location. Let $\rho_{\tau_1}(\mathcal{V}, \mathcal{V}'), \ldots, \rho_{\tau_m}(\mathcal{V}, \mathcal{V}')$ be the transition relations associated with each of the transitions $\tau_1, \ldots, \tau_m$ that reaches a location where an assertion $\varphi(\mathcal{V})$ is claimed to be hold. Then, it must be fulfilled that *for all* values of the program variables $\mathcal{V}$ and $\mathcal{V}'$, the formula $\rho_{\tau_1}(\mathcal{V}, \mathcal{V}') \vee \ldots \vee \rho_{\tau_m}(\mathcal{V}, \mathcal{V}') \rightarrow \varphi(\mathcal{V}')$ is satisfied. Since an SMT solver only can check if there are *some* values that satisfy a formula, it is necessary to transform the original problem into an equivalent one that can be solved by the SMT tool. Checking that a formula is satisfied for all values of the variables is equivalent to check that there is *no* values that satisfies the negation of the formula, i.e. $(\rho_{\tau_1}(\mathcal{V}, \mathcal{V}') \vee \ldots \vee \rho_{\tau_m}(\mathcal{V}, \mathcal{V}')) \wedge \neg\varphi(\mathcal{V}')$ is unsatisfiable.

Example 2.7. Consider again the program of Fig. 2.4. The assertion $x \leq z$ holds at the end of the loop for all values of x and z that satisfy the loop condition independently the transition taken to return ($\tau_{2.1}$ or $\tau_{3.1}$). As there is only one feasible transition that cycle back (see example 2.5), we can check the assertion always holds asking the solver if the SMT formula $(x + 1 = y \wedge y \leq z \wedge z \geq x \wedge x' = x \wedge y' = y \wedge z' = z - 1) \wedge (x' > z')$ is unsatisfiable.

The applications described previously have one common feature, they consist in *checking* some *known* property reducing the problem to SMT queries. In the literature, this kind of use of SAT/SMT solvers can be found in the derivation of counterexamples, from the models found by the solver, that explain faults [Prasad et al., 2005; Cadar et al., 2008; McMillan, 2003b; Xie and Aiken, 2005; Beyer et al., 2004; Majumdar and Xu, 2007; Godefroid et al., 2008; Jackson and Vaziri, 2000]. In contrast, the *inference* of *unknown* properties that fulfill some conditions has no straightforward encoding.

Chapter 3

Invariant Inference

In the late 60s, Hoare [Hoare, 1969] proposed an axiomatization based on mathematical basis of all the constructs of a *simple* imperative programming language as a mean to prove partial correctness of programs, where termination needs to be proved separately. Similar ideas for flowcharts, instead of text programs, have been published previously by Floyd [Floyd, 1967].

In Hoare logic, every piece of code is described using a precondition and a postcondition, which are assertions about the values of program variables before and after its execution. In this context, the intended function of a program is not specified by making assertions that ascribe to particular values of each variable, but asserting general properties of the values and the relationships holding between them. The kind of properties and assertions which are looked for are called invariants, from which *loop* invariants are a special and fundamental class of them. Specifically, an *invariant* at some program location is an assertion over the program variables that remains true whenever the location is reached.

Discovering invariants is crucial for program verification, but also a tedious and, sometimes, difficult task for the programmer. For this reason, heuristic methods for mechanically deriving invariants attracted researchers attention soon [German and Wegbreit, 1975; Wegbreit, 1974; Katz and Manna, 1973; Hegbreitt, 1973; Elspas et al., 1972]. Most of these methods are associated to the verification process and, therefore, are dependent of the output specification of the program. They try to find an *inductive* invariant, an assertion that holds the first time the location is reached and is preserved under every cycle back to the location, strengthening the post-

condition until the assertion fulfills the inductive conditions. In [Katz and Manna, 1976], invariants that are only dependent of the program code are generated using ad-hoc recursive equations.

All these works were focused on assertions about numerical relationships over program variables. That is because of their role in some many algorithms and its importance as fundamental types within programming languages. Likewise, this thesis is also centered on numerical invariants. In particular, it tackles the generation of universally quantified loop invariants over array and scalar variables. But before presenting our inference technique, the most important approaches for automatic invariant generation are reviewed.

3.1 Background: approaches

3.1.1 Abstract Interpretation

Abstract interpretation is a foundational framework for specifying program property inference as iterative approximations over a suitable domain (a lattice of facts in which the invariants are expected to lie) [Cousot and Cousot, 1977b]. The main idea behind this approach is to perform an approximate symbolic execution of the program until an assertion that remains unchanged is reached. However, in order to guarantee termination, the method introduces imprecision by use of a domain-specific extrapolation operator called *widening*. A complementary *narrowing* [Cousot and Cousot, 1992] operator is then used to improve the precision of the solution. Several widening heuristics [Wang et al., 2007; Gulavani et al., 2008; Gopan and Reps, 2007, 2006] have been developed to tailor specific classes of programs.

The set of numerical abstract domains studied includes the interval domain [Cousot and Cousot, 1977a] (that discovers variable bounds $\bigwedge_i x_i \in [a_i, b_i]$), the linear equalities domain [Karr, 1976] ($\bigwedge_j \sum_i \alpha_{ij} x_i = \beta_j$), the octagon domain [Miné, 2006] ($\bigwedge_j \pm x \pm y \leq \beta_j$), its generalization to more than two variables the octahedra domain [Clarisó and Cortadella, 2004], the polyhedron domain [Cousot and Halbwachs, 1978] ($\bigwedge_j \sum_i \alpha_{ij} x_i \leq \beta_j$), and the congruence domain [Granger, 1991] ($\bigwedge_i x_i \in a_i \mathbb{Z} + b_i$), where a_i, b_i, α_{ij} and β_j are integers.

Although some of them are clearly more expressive than the others, e.g., the polyhedron domain compared with the octagon domain, in practice the

preference between them also depends on its space and temporal computational requirements. For instance, the octagon domain is usually preferred to the polyhedron domain because the former has a memory and worst-case time cost polynomical, whereas the latter has a memory and time cost that is unbounded in theory and exponential in practice [Miné, 2006].

Besides conjunctions of numerical domains, there exist *disjunctive completions* of domains [Cousot and Cousot, 1979] including powerset extensions over linear inequalities [Giacobazzi and Ranzato, 1998; Gulavani and Rajamani, 2006].

Regarding the synthesis of quantified invariants for programs with arrays, in [Gopan et al., 2005] the index domain of arrays is partitioned into several symbolic intervals I, and then each subarray $A[I]$ is associated to a summary auxiliary variable A_I. Although assignments to individual array elements can thus be handled precisely, in order to discover relations among the contents at different indices, hints must be manually provided. This shortcoming is overcome in [Halbwachs and Péron, 2008], where additionally relational abstract properties of summary variables and shift variables are introduced to discover invariants of the form $\forall\alpha : \alpha \in I : \psi(A_1[\alpha + k_1], ..., A_m[\alpha + k_m], \overline{x})$, where $k_1, \ldots, k_m \in \mathbb{Z}$, A_i are array variables, and $\overline{x}$ are scalar variables.

3.1.2 Predicate Abstraction

Predicate abstraction [Graf and Saïdi, 1997] can be seen as an instance of abstract interpretation. It differs from standard abstract interpretation because the abstraction is parametrized by, and specific to a program. The process consists in selecting a set of predefined predicates, typically provided manually by the user or computed heuristically from the program code and the assertions to be proved, and then generate an invariant built only over those predicates.

For programming language researchers, predicate abstraction was popularized by the model checking community, and in particular the SLAM [Ball and Rajamani, 2002, 2000] and BLAST [Beyer et al., 2007a; Henzinger et al., 2002, 2004] model checkers.

Although predicate abstraction was initially used to compute quantifier-free invariants, strategies to discover universally quantified invariants [Flanagan and Qadeer, 2002; Lahiri and Bryant, 2007; Jhala and McMillan, 2007]

and disjunctions of universally quantified invariants in the context of shape analysis [Podelski and Wies, 2005] have been proposed.

3.1.3 Computational algebra

The computational algebra approach leverages recurrence solving and algebraic techniques to discover invariant properties. In [Kovács, 2008], construction of invariant equalities over numeric scalar is presented. Later, this method is generalized to the construction of invariant inequalities using a combination of quantifier elimination techniques together with a program instrumentation using an auxiliary loop counter variable [Henzinger et al., 2008]. For a restricted class of loops that do not contain any branching statements and under non-deterministic treatment of the loop condition, recurrence solving over the loop body is used in [Henzinger et al., 2010b] to compute universally quantified array invariants. Some of the previous limitations are eliminated in a subsequent work [Henzinger et al., 2010a] where loops with restricted branching control-flow are supported.

3.1.4 First-order theorem proving

First-order theorem solvers are general tools to perform deductive reasoning provided that one is able two express the axiomatization of the theories of interest. In particular, two kind of theorem provers have been used for invariant inference.

On the one hand, interpolating provers have been used to generate inductive invariants for proving properties of sequential circuits [McMillan, 2003b] and sequential programs [McMillan, 2006], as well as abstraction refinement [Henzinger et al., 2004] and universally quantified invariants [McMillan, 2008]. The method consists in over-approximating image computation based on interpolation.

On the other hand, saturation theorem provers have been used to generate invariants with alternations of quantifiers for loop programs without nesting [Kovács and Voronkov, 2009; Hoder et al., 2011]. In this approach, one describes first the loop dynamics by means of first-order formulas, possibly using additional symbols denoting array updates or loop counters, and then a saturation theorem prover eliminates auxiliary symbols and reports the consequences without these symbols, which are the invariants.

3.1.5 Constraint solving

A constraint-based method for generating linear invariants was presented in [Colón et al., 2003]. The method reduces the problem of linear invariant generation to a non-linear constraint solving problem. In [Beyer et al., 2007b], a constraint-based algorithm for the synthesis of invariants expressed in the combined theory of linear integer arithmetic (LIA) and uninterpreted function symbols (EUF) is presented. By means of the reduction of the array property fragment to EUF+LIA, it is claimed that the techniques can be extended for the generation of universally quantified invariants for arrays. However, the technique has some limitations, namely, only properties where indices occurs in array accesses of the program can be generated.

The first of the goals of this thesis is to extend the language of array invariants that can be discovered using the constraint-based approach and achieve it without requiring extra predicates or assertions, only extracting properties from the semantics of the source code.

3.2 Scalar invariant generation

This section review in detail the constraint-based method for the generation of linear invariants over scalar program variables, which establishes the basis of the work done about generation of universally quantified array invariants.

3.2.1 Problem definition

We assume that every program is modeled with a transition system with transition relations over integer linear arithmetic (see Section 2.2) and we want to find an *invariant map* μ that assigns an invariant $\mu(\ell)$ to each of the locations ℓ. The main idea behind the technique explained in [Colón et al., 2003] is to represent a linear invariant $c_1x_1 + \cdots + c_nx_n + d \leq 0$ over the program variables x_i in terms of unknown coefficients $c_1, \ldots, c_n, d$ and generate constraints on the coefficients such that any solution corresponds to an inductive invariant.

Theorem 3.1. Let μ be a map from locations to properties such that:

- For every location $\ell \in \mathcal{L}$: $\Theta(\ell) \models \mu(\ell)$
- For every transition $\tau = (\ell, \ell', \rho) \in \mathcal{T}$: $\mu(\ell) \wedge \rho \models \mu(\ell')'$.

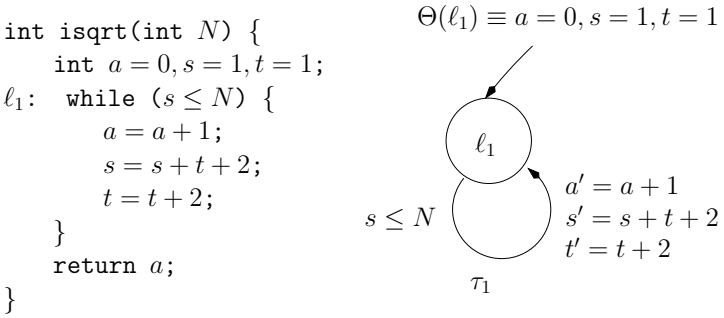


Figure 3.3. Program computing the integer square root of a natural number, and its transition system.

Then μ is an invariant map, and we say μ and its associated properties are *inductive*. We refer to the first condition of the Theorem as the *initiation* condition and to the second one as the *consecution* condition.

Example 3.2. Consider the program of Fig. 3.3 for computing the floor of the square root of a natural number N. An inductive invariant for location ℓ_1 is $-2a + t - 1 \leq 0$ since:

- $(a = 0 \wedge s = 1 \wedge t = 1) \models -2a + t - 1 \leq 0$
- $(-2a + t - 1 \leq 0) \wedge (s \leq N \wedge a' = a + 1 \wedge s' = s + t + 2 \wedge t' = t + 2) \models -2a' + t' - 1 \leq 0$

3.2.2 SMT encoding of the problem

Note that, given two formulas $\mathcal{F}$ and $\mathcal{G}$ with free variables $\overline{x}$, saying that $\mathcal{F}(\overline{x}) \models \mathcal{G}(\overline{x})$ holds is the same that saying $\forall \overline{x}(\mathcal{F}(\overline{x}) \rightarrow \mathcal{G}(\overline{x}))$ holds. As it showed in Section 2.3, using SMT solvers to check if an *known* inequality holds whenever a location is reached is straightforward. However, in order to discover some coefficients such that an inequality is inductive invariant, it is necessary to transform an $\exists\forall$ problem into an $\exists$ problem which can be handled directly by an SMT solver.

It is a well-known fact that given a conjunction of equalities and inequalities over a set of real variables[1], we can construct a logical consequence adding a multiple of one or more inequalities. The formalization of this

[1]Recall that an equality $A = B$ is equivalent to the conjunction of two inequalities $A \leq B$ and $B \leq A$.

elementary mathematical concept is the following result from polyhedral geometry:

Theorem 3.4 (Farkas' Lemma)**.** [Schrijver, 1998] Consider the following system of linear inequalities over real-valued variables $x_1, \ldots, x_n$:

$$S : \begin{bmatrix} a_{11}x_1 + \cdots + a_{1n}x_n + b_1 \leq 0 \\ \vdots \qquad\qquad \vdots \quad \vdots \leq 0 \\ a_{m1}x_1 + \cdots + a_{mn}x_n + b_m \leq 0 \end{bmatrix}$$

When S is satisfiable, it entails a given linear inequality

$$\psi : c_1x_1 + \cdots + c_nx_n + d \leq 0$$

if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_m$, such that

$$c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \quad \ldots \quad , c_n = \sum_{i=1}^{m} \lambda_i a_{in},\ d = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

Furthermore, S is unsatisfiable if and only if the inequality $1 \leq 0$ can be derived as shown above.

Although Farkas' lemma is applied to a conjunction of inequalities, an equality can be represented using two inequalities whose coefficients are the same with the sign changed. Therefore, regarding lambda values, an equality can be treated like an inequality where the lambda multiplier has no restriction, i.e. it can be positive, negative or zero.

Example 3.5. Consider the program of Fig. 3.3. Suppose we want to find an inductive invariant $\mu(\ell_1) : c_1a + c_2s + c_3t + d \leq 0$ at ℓ_1 where c_1, c_2, c_3, d are unknown coefficients. Thus, our problem is to find coefficients such that $\Theta(\ell_1) \models \mu(\ell_1)$ and $\mu(\ell_1) \wedge \rho_{\tau_1} \models \mu(\ell_1)'$. As it was shown in Example 3.2, for $c_1 = -2$, $c_2 = 0$, $c_3 = 1$, and $d = -1$, $\mu(\ell_1)$ is an inductive invariant. Therefore, according to Farkas' Lemma, there should exist $\lambda_0, \ldots, \lambda_3$ and $\lambda'_0, \ldots, \lambda'_5$ such that:

- $\lambda_0, \lambda'_0, \lambda'_1, \lambda'_2 \geq 0$
- $\lambda_1(a = 0) + \lambda_2(s - 1 = 0) + \lambda_3(t - 1 = 0) \equiv c_1a + c_2s + c_3t + d + \lambda_0 \leq 0$
- $\lambda'_1(c_1a + c_2s + c_3t + d \leq 0) + \lambda'_2(s - N \leq 0) + \lambda'_3(a' - a - 1 = 0) + \lambda'_4(s' - s - t - 2 = 0) + \lambda'_5(t' - t - 2 = 0) \equiv c_1a' + c_2s' + c_3t' + d + \lambda'_0 \leq 0$

that rewritten as equations like in Farkas' lemma are the following:

- $\lambda_0, \lambda'_0, \lambda'_1, \lambda'_2 \geq 0$
- $c_1 = \lambda_1 = \lambda'_3$, $c_2 = \lambda_2 = \lambda'_4$, $c_3 = \lambda_3 = \lambda'_5$
- $d = -\lambda_0 + \lambda_2 - \lambda_3 = \lambda'_1 d - \lambda'_3 - 2\lambda'_4 - 2\lambda'_5$
- $\lambda'_1 c_1 - \lambda'_3 = 0$, $\lambda'_1 c_2 + \lambda_2 - \lambda_4 = 0$
- $\lambda'_1 c_3 - \lambda'_4 - \lambda'_5 = 0$, $-\lambda'_2 = 0$

In fact, $\lambda_0 = 0$, $\lambda_1 = -2$, $\lambda_2 = 0$, $\lambda_3 = 1$, $\lambda'_0 = 0$, $\lambda'_1 = 1$, $\lambda'_2 = 0$, $\lambda'_3 = -2$, $\lambda'_4 = 0$, $\lambda'_5 = 1$ is a solution for the inductive invariant $-2a+t-1 \leq 0$. In general, for every $c_1, \ldots, c_3, d$ such that there also exist multipliers $\lambda_0, \ldots, \lambda_3$ and $\lambda'_0, \ldots, \lambda'_5$ which satisfy the encoding of the inductive conditions using Farkas' lemma, then $\mu(\ell_1)$ is an inductive invariant.

As it can be observed in Example 3.5, the problem that we obtain after encoding the inductive conditions is a satisfiability problem in propositional logic over *non-linear* arithmetic (note that λ'_1 multiply the unknown coefficients $c_1, \ldots, c_3, d$). The nonlinearity does not come from Farkas' lemma *per se*, but from the existence of an inequality in the system S which have *unknown* coefficients a_{ji} and b_j. Moreover, if one is interested in linear invariants with integer coefficients, as some unknowns are integer (the invariant coefficients) and some are real (the multipliers $\lambda_0, \lambda_1, \ldots, \lambda_m$), an SMT problem in mixed arithmetic is obtained.

Since Farkas' Lemma applies to reals, one may lose some inductive invariants, namely those that only hold using the fact that the program variables are integers. In order to perform (partial) integer reasoning, the following variation of Farkas' Lemma, based on the Gomory-Chvátal cutting plane rule [Robinson and Voronkov, 2001], is employed in practice:

Lemma 3.6. Let $Ax + b \leq 0$ ($A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$) be a system of linear inequalities over integer variables $x^T = (x_1, \ldots, x_n)$, and $c^T x + d \leq 0$ ($c \in \mathbb{Z}^n, d \in \mathbb{Z}$) be a linear inequality. If there is $\lambda \in \mathbb{R}^m$, such that $\lambda \geq 0$, $c^T = \lambda^T A$, $\lambda^T b > d - 1$, then $Ax + b \leq 0$ entails $c^T x + d \leq 0$.

Proof. If $Ax + b \leq 0$ is unsatisfiable, then the result follows trivially. So let us consider $x \in \mathbb{Z}^n$ such that $Ax+b \leq 0$. As $\lambda \geq 0$, we have $\lambda^T Ax+\lambda^T b \leq 0$. Since $c^T = \lambda^T A$, and $\lambda^T b > d - 1$, we have $c^T x = \lambda^T Ax \leq -\lambda^T b < 1 - d$. But $x \in \mathbb{Z}^n$ and $c \in \mathbb{Z}^n$. Hence, $c^T x + 1 \leq 1 - d$, i.e., $c^T x + d \leq 0$. □

3.3 Array invariant generation

Arrays are among the oldest and most important data structures, and are used by almost every program. In order to verify the correctness of programs manipulating arrays, usually one has to take into account invariant relationships among values stored in arrays and scalar variables. However, due to the unbounded nature of arrays, invariant generation for these programs is a challenging problem.

This section presents a novel method for generating universally quantified loop invariants over array and scalar variables, and it is one of the first contributions of this thesis. The method builds upon the so-called constraint-based approach and is able to generate automatically a quite general family of properties that allows handling a wide variety of programs. Namely, let $\overline{v} = (v_1, \ldots, v_n)$ and $\overline{a} = (A_1, \ldots, A_m)$ be, respectively, the scalar and the array variables of a program. Given an integer $k > 0$, the method generates invariants of the form:

$$\forall \alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i [d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0 \, ,$$

where $\mathcal{C}, \mathcal{E}_{ij}, \mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v}$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$ for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, k\}$.

Example 3.7. Consider the program showed on the left side of Fig. 3.8. An array A is filled with zeros from the middle outwards, moving alternatively to the left and to the right. An inductive loop invariant for this program is $P \ \equiv \ \forall \alpha \ : \ 0 \leq \alpha < r - l - 1 \ : \ A[\alpha + l + 1] = 0$.

3.3.1 Modelling programs with arrays

Henceforth, we will consider programs that consist of unnested loops and linear assignments, conditions and array accesses. We will model programs by means of transition systems (cf. Section 2.2), where the tuple of variables $\mathcal{V}$ contains the scalar variables $\overline{v}$, and the array variables $\overline{a}$. The *size* of an array $A \in \overline{a}$ is denoted by $|A|$ and the *domain* of its indices is $\{0 \ldots |A| - 1\}$ (i.e., indices start at 0, as in C-like languages). We assume that arrays can only be indexed by expressions built over scalar variables. Hence, by means of the read/write semantics of arrays, we can describe transition relations as array equalities (possibly guarded by conjunctions of equalities

```
int main() {
  int l=4, r=5, A[8];
  while (l>=0 and r<8)
    if (???)
      A[r++]=0;
    else
      A[l--]=0;
}
```

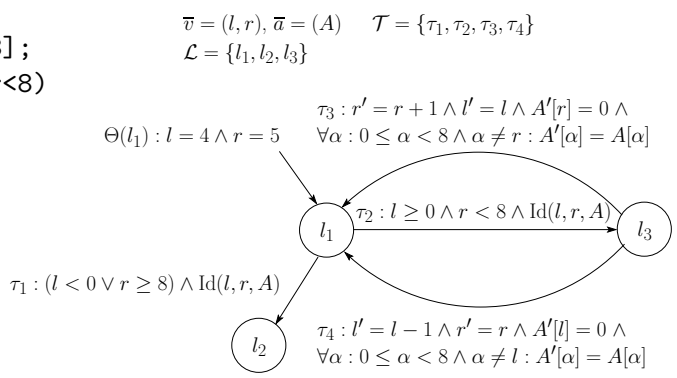


Figure 3.8. Program and its transition system. Predicate $\mathrm{Id}(u_1, \ldots, u_k)$ is short for $u_1 = u_1' \wedge \cdots \wedge u_k = u_k'$, i.e., indicates those variables that remain identical after a transition.

and disequalities between scalar expressions) and quantified information of the form $\forall \alpha : 0 \leq \alpha \leq |A| - 1 \wedge P(\alpha) : A'[\alpha] = A[\alpha]$, where P does not depend on array variables. Selected a location ℓ for which an array invariant is searched, we will discard every intermediate location ℓ' placed in the middle of a return path from ℓ to itself, merging the corresponding transitions as it was explained in Section 2.2.3.

Example 3.9. On the right side of Fig. 3.8 is showed the transition system associated with the program considered in Example 3.7. Since we want to find an array invariant at location ℓ_1, we ignore location ℓ_3 merging transitions τ_2 and τ_3 into a new transition $\tau_{3.1}$ and transitions τ_2 and τ_4 into a new transition $\tau_{4.1}$, both of them connecting now location ℓ_1 with itself. The resulting transition relations are:

$$\begin{aligned}
\rho_{\tau_{3.1}} : \;& l \geq 0 \;\wedge\; r < 8 \;\wedge\; r' = r + 1 \;\wedge\; l' = l \;\wedge\; A'[r] = 0 \wedge \\
& \forall \alpha \;:\; 0 \leq \alpha < 8 \wedge \alpha \neq r \;:\; A'[\alpha] = A[\alpha] \\
\rho_{\tau_{4.1}} : \;& l \geq 0 \;\wedge\; r < 8 \;\wedge\; l' = l - 1 \;\wedge\; r' = r \;\wedge\; A'[l] = 0 \wedge \\
& \forall \alpha \;:\; 0 \leq \alpha < 8 \wedge \alpha \neq l \;:\; A'[\alpha] = A[\alpha] \,.
\end{aligned}$$

Note the use of quantified information to describe which array elements remain with the same value in contrast to which ones have changed.

3.3.2 Illustration of the method

Similarly as it was explained in Section 3.2 for invariants over scalar variables, we tackle the generation of array invariants of the form described, by

finding expressions $\mathcal{C}, \mathcal{E}_{ij}, \mathcal{B}$ and coefficients a_{ij}, d_{ij}, b_α such that the inductive conditions (recall Theorem 3.1) are satisfied. For do that, we encode the conditions into SMT problems using Farkas' lemma and leverage the solver to find solutions for the unknowns.

Example 3.10. Suppose we want to find an array invariant $\mu(\ell_1)$ of the form $\forall\alpha : 0 \leq \alpha < \mathcal{C}(\overline{v}) : aA[d\alpha + \mathcal{E}(\overline{v})] + \mathcal{B}(\overline{v}) + b_\alpha\alpha \leq 0$ for the program of Example 3.9, like the invariant P of Example 3.7. In that case, we should find $\mathcal{C}, \mathcal{E}, \mathcal{B}, a, d, b_\alpha$ such that $\Theta(\ell_1) \models \mu(\ell_1)$, $\mu(\ell_1) \wedge \rho_{\tau_{3.1}} \models \mu(\ell_1)'$ and $\mu(\ell_1) \wedge \rho_{\tau_{4.1}} \models \mu(\ell_1)'$.

A way to ensure that a universally quantified array invariant satisfy the initiation condition is to force that the domain of the universally quantified variable is empty, what it could be achieved imposing constraints over $\mathcal{C}(\overline{v})$. Although at first this restriction could seem a limitation, in fact, it is very common that the starting point of the index variables that manipulate arrays fulfils this condition. That is also true even when the array invariant already holds for some initial range.

Example 3.11. Following Example 3.10, note that the solution $\mathcal{C}(\overline{v}) = r - l - 1$ fulfills that the initial conditions $\Theta(\ell_1)$ entail the property $P \equiv \forall\alpha : 0 \leq \alpha < r - l - 1 : A[\alpha + l + 1] = 0$. In particular, we need to see that $l = 4 \wedge r = 5 \models P$. This is trivial, since $l = 4$ and $r = 5$ imply that $r - l - 1$ is 0, i.e., the domain of the universally quantified variable α in P is empty.

The next step is to ensure that the consecution condition holds, i.e., the property is preserved after any transition that cycles back. If the domain of the universally quantified variable have changed after the transition, it is necessary to check that every array element currently indexed fulfills the property. Here we can find two cases, namely, the array element have been considered in previous iterations, or it is the first time that occurs. In the former case, a sufficient condition to prove that the property is preserved is to ensure that the array element have not been modified. In the later case, it is necessary to ensure the property holds for the new array element.

Example 3.12. Consider transition $\tau_{3.1}$ of the transition system of Example 3.9. We have to check that:

$$P \ \wedge \ l \geq 0 \ \wedge \ r < 8 \ \wedge \ r' = r + 1 \ \wedge \ l' = l \ \wedge \ A'[r] = 0$$
$$\wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq r \ : \ A'[\alpha] = A[\alpha] \ \models \ P' .$$

Now notice that the expression $r' - l' - 1$, which determines the domain of α in P', also has the property that $r' - l' - 1 = (r + 1) - l - 1 = (r - l - 1) + 1$. This means that, after $\tau_{3.1}$, the domain of α has exactly one new element, $\alpha = r - l - 1$. First, let us see that, after the transition, property $A'[\alpha + l' + 1] = A'[\alpha + l + 1] = 0$ holds for the other values of α, i.e., $\alpha \in \{0, \ldots, r - l - 2\}$. Indeed this is the case: since $\forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq r \ : \ A'[\alpha] = A[\alpha]$, all positions of A' except for the r-th remain the same. But $A'[r] = A'[(r - l - 1) + l' + 1]$ precisely corresponds to $\alpha = r - l - 1$. Hence from P we have that $A'[\alpha + l' + 1] = 0$ for all $\alpha \in \{0, \ldots, r - l - 2\}$. Now we only need to check $A'[\alpha + l' + 1] = 0$ for $\alpha = r - l - 1$, which follows from the premise $A'[r] = 0$. In conclusion, P' holds.

As regards $\tau_{4.1}$ we have to check that:

$$P \ \wedge \ l \geq 0 \ \wedge \ r < 8 \ \wedge \ l' = l - 1 \ \wedge \ r' = r \ \wedge \ A'[l] = 0$$
$$\wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq l \ : \ A'[\alpha] = A[\alpha] \ \models \ P' .$$

Again, the expression $r' - l' - 1$ also satisfies that $r' - l' - 1 = r - (l - 1) - 1 = (r - l - 1) + 1$. Hence the domain of α has exactly one new element. But unlike in the previous case, l changes its value. To prove P' from P, it is convenient to rewrite P so that array accesses are expressed in terms of $A[\alpha + l' + 1]$. By making a shift, P is equivalent to $\forall \alpha \ : \ 1 \leq \alpha < r' - l' - 1 \ : \ A[\alpha + l' + 1] = 0$. Again, since $\forall \alpha \ : \ 0 \leq \alpha < 8 \ \wedge \ \alpha \neq l \ : \ A'[\alpha] = A[\alpha]$, all positions of A' except for the l-th remain the same. But $A'[l] = A'[l' + 1]$ precisely corresponds to $\alpha = 0$. Therefore $A'[\alpha + l' + 1] = 0$ for all $\alpha \in \{1, \ldots, r' - l' - 2\}$. Further, as $A'[l] = 0$, we have that $A'[\alpha + l' + 1] = 0$ for $\alpha = 0$. Thus P' holds.

Apart from proving that P is invariant, we may also want to check that the array accesses that occur in it are correct. As regards initiation transitions, since the domain of α at the beginning is empty, there is nothing to check. Regarding consecution transitions, for example for $\tau_{3.1}$ we have to see that

$$l \geq 0 \wedge r < 8 \wedge r' = r + 1 \wedge l' = l \rightarrow \forall \alpha : 0 \leq \alpha < r' - l' - 1 : \alpha + l' + 1 \geq 0 \wedge \alpha + l' + 1 < 8,$$

where for the sake of simplicity we have ignored the array variable. Now, given that array accesses are linear functions in α, it is sufficient to check

correctness for $\alpha = 0$ and $\alpha = r' - l' - 2$, i.e., that the above premises entail $l' + 1 \geq 0 \ \wedge \ l' + 1 < 8 \ \wedge \ r' - 1 \geq 0 \ \wedge \ r' - 1 < 8$. Let us assume that we have already looked for linear inequality invariants over scalar variables (e.g., with the technique explained in Section 3.2), and have found that $l \leq r - 1$ is a loop invariant. Adding this invariant to the transition relation suffices to prove the above implication. A similar argument applies for $\tau_{4.1}$.

In general, our invariant generation method guarantees that the array accesses occurring in the synthesized invariants are correct. As in the example, this is achieved by ensuring that the accesses of the extreme values of universally quantified variables are correct. Since this often requires arithmetic properties of the scalar variables of the program, in practice it is convenient that, prior to the application of our array invariant generation techniques, a linear relationship analysis for the scalar variables has already been carried out.

3.3.3 Formal description of the method

Let $\overline{a} = (A_1, \ldots, A_m)$ be the tuple of array variables. Given a positive integer $k > 0$, our method generates invariants of the form

$$\forall \alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i [d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0 \, ,$$

where $\mathcal{C}$, $\mathcal{E}_{ij}$ and $\mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v} = (v_1, \ldots, v_n)$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$, for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$.

This template covers a quite general family of properties. See Section 3.3.5 for a sample of diverse programs for which we can successfully produce useful invariants and which cannot be handled by already existing techniques.

The invariant generation process at the cutpoint of the unnested loop under consideration is split into three steps, in order to make the approach computationally feasible:

1. Expressions $\mathcal{C}$ are generated such that the domain $\{0 \ldots \mathcal{C}-1\}$ is empty after every initiation transition reaching the cutpoint, and $\mathcal{C}$ does not change or is increased by one after every consecution transition. This guarantees that any property universally quantified with this domain holds after all initiation transitions and the domain includes at most

one more element after every consecution transition. We avoid the synthesis of different expressions that under the known information define the same domain. In the running example, we generate $\mathcal{C}(l, r) = r - l - 1$.

2. For every expression $\mathcal{C}$ obtained in the previous step and for every array A_i, linear expressions $d_i\alpha + \mathcal{E}_i$ over the scalar variables are generated such that: (i) $A_i[d_i\alpha + \mathcal{E}_i]$ is a correct access for all α in $\{0 \dots \mathcal{C} - 1\}$; (ii) none of the already considered positions in the quantified property is changed after any execution of the consecution transitions; and (iii), after every consecution transition, either $\mathcal{E}_i$ does not change or its value is $\mathcal{E}_i - d_i$. Namely, if $\mathcal{C}$ does not change, then $\mathcal{E}'_i = \mathcal{E}_i$ ensures that the invariant is preserved. Otherwise, the invariant has to be extended for a new value of α. If $\mathcal{E}_i$ does not change, from the previous condition for all $\alpha \in \{0, \dots, \mathcal{C} - 1\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i\alpha + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = \mathcal{C}$. Otherwise, if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for all $\alpha \in \{1, \dots, \mathcal{C}\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i(\alpha - 1) + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = 0$.

   In the running example, we generate $d_{11} = 1$ and $\mathcal{E}_{11} = l + 1$.

3. For the selected $\mathcal{C}$ we choose k expressions $\mathcal{E}_{ij}$ for every array A_i among the generated $\mathcal{E}_i$, such that for each consecution transition either all selected $\mathcal{E}_{ij}$ remain the same after the transition, or all have as new value $\mathcal{E}_{ij} - d_{ij}$ after the transition. Then, in order to generate invariant properties we just need to find integer coefficients a_{ij} and b_α and an expression $\mathcal{B}$ such that, depending on the case, either the property is fulfilled when $\alpha = \mathcal{C}$ at the end of all consecution transitions that increase $\mathcal{C}$ or it is fulfilled when $\alpha = 0$ at the end of all consecution transitions that increase $\mathcal{C}$. Further, $\mathcal{B}$ and b_α have to fulfill that the quantified property is maintained for $\alpha \in \{0 \dots \mathcal{C} - 1\}$, assuming that the contents of the already accessed positions are not modified.

   For instance, in the running example for $k = 1$ we generate $a_{11} = 1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall\alpha : 0 \leq \alpha < r - l - 1 : A[\alpha + l + 1] \leq 0$; and $a_{11} = -1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall\alpha : 0 \leq \alpha < r - l - 1 : -A[\alpha + l + 1] \leq 0$.

Next we formalize all these conditions, which ensure that every solution to the last phase provides an invariant, and show how to encode them as SMT problems.

While for scalar linear templates the conditions of Theorem 3.1 can be directly transformed into constraints over the parameters (recall the technique described in Section 3.2), this is no longer the case for our template of array invariants. To this end we particularize Theorem 3.1 in a form that is suitable for the constraint-based invariant generation method. The proof of this specialized theorem, given in detail below, mimics the proof of invariance of the running example given at the beginning of this section.

Let $\tau_1^I \dots \tau_p^I$ be the initiation transitions to our cutpoint and $\tau_1^C \dots \tau_q^C$ the consecution transitions going back to the cutpoint.

Theorem 3.13. Let $\mathcal{C}$, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$. If

1. Every initiation transition τ_r^I with transition relation $\rho_{\tau_r^I}$ satisfies $\rho_{\tau_r^I} \Rightarrow \mathcal{C}' = 0$.

2. For all consecution transitions τ_s^C with transition relation $\rho_{\tau_s^C}$, we have $\rho_{\tau_s^C} \Rightarrow (\mathcal{C}' = \mathcal{C} \ \vee \ \mathcal{C}' = \mathcal{C} + 1)$.

3. For all consecution transitions τ_s^C, all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$, we have $\rho_{\tau_s^C} \ \wedge \ \mathcal{C}' > 0 \Rightarrow 0 \leq \mathcal{E}'_{ij} \leq |A_i| - 1 \ \wedge \ 0 \leq d_{ij}(\mathcal{C}' - 1) + \mathcal{E}'_{ij} \leq |A_i| - 1$.

4. For all consecution transitions τ_s^C we have either

   (a) $\rho_{\tau_s^C} \ \wedge \ \mathcal{C}' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$, or

   (b) $\rho_{\tau_s^C} \Rightarrow \mathcal{C}' = \mathcal{C} + 1 \ \wedge \ \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$.

5. For all consecution transitions τ_s^C, we have $\rho_{\tau_s^C} \Rightarrow \forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] = A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$.

6. For all consecution transitions τ_s^C, we have

   - $\rho_{\tau_s^C} \ \wedge \ \mathcal{C}' = \mathcal{C} + 1 \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\mathcal{C} + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \mathcal{C} \leq 0$, if case 4a applies.

- $\rho_{\tau_s^C} \Rightarrow \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}'_{ij}] + \mathcal{B}' \leq 0$, if case 4b applies.

7. For all consecution transitions τ_s^C, we have

   - $\rho_{\tau_s^C} \wedge 0 \leq \alpha \leq \mathcal{C} - 1 \wedge x + \mathcal{B} + b_\alpha \alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha \alpha \leq 0$ for some fresh universally quantified variable x, if case 4a applies.
   - $\rho_{\tau_s^C} \wedge 0 \leq \alpha \leq \mathcal{C} - 1 \wedge x + \mathcal{B} + b_\alpha \alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$ for some fresh universally quantified variable x, if case 4b applies.

Then $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ is invariant.

Proof. Following Theorem 3.1, we show that the property holds after each initiation transition, and that it is maintained after each consecution transition.

The first condition easily holds by applying 1, since we have that $\rho_{\tau_r^I} \Rightarrow \mathcal{C}' = 0$ for every initiation transition τ_r^I, which implies $\forall \alpha : 0 \leq \alpha \leq \mathcal{C}' - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution transitions τ_s^C, we have $\rho_{\tau_s^C} \wedge \forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ implies $\forall \alpha : 0 \leq \alpha \leq \mathcal{C}' - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$.

By condition 2, we have $\rho_{\tau_s^C} \Rightarrow (\mathcal{C}' = \mathcal{C} \vee \mathcal{C}' = \mathcal{C} + 1)$, and by condition 4 either $\rho_{\tau_s^C} \wedge \mathcal{C}' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$, or $\rho_{\tau_s^C} \Rightarrow \mathcal{C}' = \mathcal{C} + 1 \wedge \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$. We distinguish three cases:

1. $\mathcal{C}' = \mathcal{C}$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. By condition 5, we can replace A'_i by A_i in the given domain, and hence we have to show that $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Then, since the array part coincides with the one of the assumption, we can replace it in both places by some fresh variable x. Now it suffices to show that, assuming $x + \mathcal{B} + b_\alpha \alpha \leq 0$, we have $x + \mathcal{B}' + b_\alpha \alpha \leq 0$ for all value of x, which follows from the premises and condition 7.

2. $\mathcal{C}' = \mathcal{C} + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. By conditions 1 and 2 we have

$0 \leq \mathcal{C}$, and hence $\mathcal{C} = \mathcal{C}' - 1$ belongs to the domain $\{0 \dots \mathcal{C}\}$ and $\mathcal{C}' > 0$. Then, by condition 3, we have that $0 \leq d_{ij}\mathcal{C} + \mathcal{E}_{ij} \leq |A_i| - 1 = |A'_i| - 1$ for all i and j. Therefore, we can extract the case $\alpha = \mathcal{C}$ from the quantifier obtaining $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\mathcal{C} + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \mathcal{C} \leq 0$. The first part holds as before by the premises and conditions 5 and 7, and the second part holds by the premises and condition 6.

3. $\mathcal{C}' = \mathcal{C} + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Since, by conditions 1 and 2, we have $0 \leq \mathcal{C}$, we have that $\mathcal{C}$ belongs to the domain $\{0 \dots \mathcal{C}\}$. By condition 3, we have $0 \leq \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij} \leq |A'_i| - 1$. Therefore, we can extract the case $\alpha = 0$ from the quantifier obtaining $\forall \alpha : 1 \leq \alpha \leq \mathcal{C} : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' \leq 0$. For the first one, replacing α by $\alpha + 1$ we have $\forall \alpha : 1 \leq \alpha + 1 \leq \mathcal{C} : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}(\alpha + 1) + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, or equivalently $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, which holds by applying conditions 5 and 7 as before. The second part holds again by the premises and condition 6, using the fact that $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$.

□

As we have described, our invariant generation method consists of three phases. The first phase looks for expressions $\mathcal{C}$ satisfying conditions 1 and 2. The second one provides, for every generated $\mathcal{C}$ and for every array A_i, expressions $\mathcal{E}_i$ with their corresponding integers d_i that fulfill conditions 3, 4 and 5. Note that, to satisfy condition 4, we need to record for each expression and transition whether we have $\mathcal{E}'_i = \mathcal{E}_i$ or $\mathcal{E}'_i = \mathcal{E}_i - d_i$, so as to ensure that all expressions $\mathcal{E}_{ij}$ have the same behavior. Finally, in the third phase we have to find coefficients a_{ij} and b_α and an expression $\mathcal{B}$ fulfilling conditions 6 and 7.

Solutions to all three phases are obtained by encoding the conditions of Theorem 3.13 into SMT problems in non-linear arithmetic thanks to Farkas' Lemma. Note that, because of array updates, transition relations may not be conjunctions of literals (i.e., atomic predicates or negations of atomic predicates). As in practice the guarded array information is useless until the last phase, in the first two phases we use the unconditional part of a

transition relation ρ, i.e., the part of ρ that is a conjunction of literals, denoted by $U(\rho)$.

Encoding Phase 1 Let $\mathcal{C}$ be $c_1 v_1 + \ldots + c_n v_n + c_{n+1}$, where $\overline{v}$ are the scalar variables and $\overline{c}$ are the integer unknowns. Then conditions 1 and 2 can be expressed as:

$$\exists \overline{c} \forall \overline{v}, \overline{v'} \bigwedge_{r=1}^{p} (U(\rho_{\tau_r^I}) \Rightarrow \mathcal{C}' = 0) \wedge \bigwedge_{s=1}^{q} (U(\rho_{\tau_s^C}) \Rightarrow \mathcal{C}' = \mathcal{C} \ \vee \ \mathcal{C}' = \mathcal{C} + 1).$$

We cannot apply Farkas' Lemma directly due to the disjunction in the conclusion of the second condition. To solve this, we move one of the two literals into the premise and negate it. As the literal becomes a disequality, it can be split into a disjunction of inequalities. Finally, thanks to the distributive law, Farkas' Lemma can be applied and an existentially quantified SMT problem in non-linear arithmetic is obtained. We also encode the condition that each newly generated $\mathcal{C}$ must be different from all previously generated expressions at the cutpoint, considering all already known scalar invariants.

Encoding Phase 2 Here, for each $\mathcal{C}$ obtained in the previous phase and for each array A_i, we generate expressions $\mathcal{E}_i$ and integers d_i that satisfy conditions 3 and 5, and also condition 4 as a single expression and not combined with the other expressions.

The encoding of condition 3 is direct using Farkas' Lemma. Now let us sketch the encoding of condition 4. Let $\mathcal{E}_i$ be $e_1 v_1 + \ldots + e_n v_n + e_{n+1}$, where $\overline{e}$ are integer unknowns. Then, as $\mathcal{E}_i$ is considered in isolation, we need

$$\exists \overline{e}, d_i \ \forall \overline{v}, \overline{v'} \ \textstyle\bigwedge_{s=1}^{q} \ \rho_{\tau_s^C} \Rightarrow ((\mathcal{C}' = \mathcal{C} + 1 \ \wedge \ \mathcal{E}_i' = \mathcal{E}_i - d_i) \ \vee \ \mathcal{C}' \leq 0 \ \vee \ \mathcal{E}_i' = \mathcal{E}_i) \, .$$

To apply Farkas' Lemma, we use a similar transformation as for condition 2. In addition, it is imposed that the newly generated expressions are different from the previous ones.

Regarding condition 5, the encoding is rather different. In this case, for every consecution transition τ_s^C, array A_i and expression $G \Rightarrow A_i'[W] = M$ in $\rho_{\tau_s^C}$, we ensure that

$$\forall \alpha \left(\rho_{\tau_s^C} \ \wedge \ 0 \leq \alpha \leq \mathcal{C} - 1 \ \wedge \ G \Rightarrow \ (W \neq d_i \alpha + \mathcal{E}_i \ \vee \ M = A_i[W]) \right) .$$

To avoid generating useless expressions, we add in the encoding a condition stating that if $\mathcal{E}'_i = \mathcal{E}_i$ then for every consecution transition where $\mathcal{C}$ is incremented, there is at least an access $A_i[W]$ in the transition such that $W = d_i(\mathcal{C}'-1)+\mathcal{E}'_i$. Otherwise, i.e., if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for every consecution transition where $\mathcal{C}$ is incremented, there is at least an access $A_i[W]$ in the transition such that $W = \mathcal{E}'_i$.

Encoding Phase 3 Condition 7 is straightforward. Regarding condition 6, the encoding does not need non-linear arithmetic, but requires to handle arrays:

$$\begin{array}{l}\exists \overline{a}, \overline{b}, b_\alpha \; \forall \overline{v}, \overline{v'}, \overline{A}, \overline{A'} \\ \bigwedge_{s=1}^{q} \quad (\rho_{\tau_s^C} \qquad\qquad\quad \Rightarrow \quad \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}'_{ij}] \; + \mathcal{B}' \; \leq \; 0) \qquad \wedge \\ \qquad\quad (\rho_{\tau_s^C} \; \wedge \; \mathcal{C}' = \mathcal{C} + 1 \quad \Rightarrow \quad \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{C} + \mathcal{E}'_{ij}] \; + \mathcal{B}' + b_\alpha \mathcal{C} \; \leq \; 0)\,.\end{array}$$

Here, the use of the guarded array information is crucial. However, since we want to apply Farkas' Lemma, array accesses have to be replaced by new universally quantified integer variables. In order to avoid losing too much information, we add the array read semantics after the replacement; i.e., if $A[i]$ and $A[j]$ have been respectively replaced by fresh variables z_i and z_j, then $i = j \Rightarrow z_i = z_j$ is added.

3.3.4 Extensions

Relaxations on Domains

Let us consider the following program:

```
int A[2*N], min, max, i;
if (A[0] < A[1]) { min = A[0]; max = A[1]; }
else             { min = A[1]; max = A[0]; }
for (i = 2; i < 2*N; i += 2) {
  int tmpmin, tmpmax;
  if (A[i] < A[i+1]) { tmpmin = A[ i ]; tmpmax = A[i+1]; }
  else               { tmpmin = A[i+1]; tmpmax = A[ i ]; }
  if (max < tmpmax) max = tmpmax;
  if (min > tmpmin) min = tmpmin;  }
```

It computes the minimum and the maximum of an even-length array simultaneously, using a number of comparisons which is 1.5 times its length. To prove correctness, the invariants $\forall \alpha : 0 \leq \alpha \leq i-1 : v[\alpha] \geq min$ and

$\forall\alpha : 0 \leq \alpha \leq i-1 : v[\alpha] \leq max$ are required. To discover them, two extensions of Theorem 3.13 are required:

- The domain of the universally quantified variable α cannot be forced to be initially empty. In this example, when the loop is entered, both invariants already hold for $\alpha = 0, 1$. This can be handled by applying our invariant generation method as described in Section 3.3.3, and for each computed invariant trying to extend the property for decreasing values of $\alpha = -1, -2$, etc. as much as possible. Finally, a shift of α is performed so that the domain of α begins at 0 and the invariant can be presented in the form of Section 3.3.3.

- The domain of the universally quantified variable α cannot be forced to increase at most by one at each loop iteration. For instance, in this example at each iteration the invariants hold for two new positions of the array. Thus, for a fixed parameter Δ, Condition 2 in Theorem 3.13 must be replaced by $\rho_{\tau_s^C} \Rightarrow (\mathcal{C}' = \mathcal{C} \vee \mathcal{C}' = \mathcal{C}+1 \vee \cdots \vee \mathcal{C}' = \mathcal{C}+\Delta)$. In this example, taking $\Delta = 2$ is required. Further, conditions 4b, 6 and 7 must also be extended accordingly in the natural way.

Sorted Arrays

The program below implements binary search: given a non-decreasingly sorted array A and a value x, it determines whether there is a position in A containing x:

```
assume(N > 0);
int A[N], l = 0, u = N-1;
while (l <= u) {
  int m = (l+u)/2;
  if      (A[m] < x) l = m+1;
  else if (A[m] > x) u = m-1;
  else break;  }
```

To prove that, on exiting due to $l > u$, the property $\forall\alpha : 0 \leq \alpha \leq N-1 : A[i] \neq x$ holds, one can use that $\forall\alpha : 0 \leq \alpha \leq l-1 : A[\alpha] < x$ and $\forall\alpha : u+1 \leq \alpha \leq N-1 : A[\alpha] > x$ are invariant. To synthesize them, the fact that A is sorted must be taken into account. The following theorem results from incorporating the property of sortedness into Theorem 3.13:

Theorem 3.14. Let $\mathcal{C}$, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$. If

1. For all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$ we have $b_\alpha \geq 0$, and $d_{ij} > 0 \Rightarrow a_{ij} \geq 0$, and $d_{ij} < 0 \Rightarrow a_{ij} \leq 0$.

2. Each initiation transition τ_r^I with transition relation $\rho_{\tau_r^I}$ fulfills $\rho_{\tau_r^I} \Rightarrow \mathcal{C}' = 0$.

3. Each initiation transition τ_r^I with transition relation $\rho_{\tau_r^I}$ fulfills $\rho_{\tau_r^I} \Rightarrow \forall \beta : 0 < \beta \leq |A_i'| - 1 : A_i'[\beta - 1] \leq A_i'[\beta]$ for all $i \in \{1 \dots m\}$.

4. Each consecution transition τ_s^C with transition relation $\rho_{\tau_s^C}$ fulfills $\rho_{\tau_s^C} \Rightarrow \mathcal{C}' \geq \mathcal{C}$.

5. For all consecution transitions τ_s^C all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$ we have
$\rho_{\tau_s^C} \wedge \mathcal{C}' > 0 \Rightarrow 0 \leq \mathcal{E}'_{ij} \leq |A_i| - 1 \wedge 0 \leq d_{ij}(\mathcal{C}' - 1) + \mathcal{E}'_{ij} \leq |A_i| - 1$.

6. For all consecution transitions τ_s^C we have one of the following:

   (a) $\rho_{\tau_s^C} \wedge \mathcal{C}' > 0 \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \leq \mathcal{E}_{ij}$ and
   $\rho_{\tau_s^C} \wedge \mathcal{C}' > 0 \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \geq \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$;

   (b) $\rho_{\tau_s^C} \Rightarrow \mathcal{C}' > \mathcal{C}$ and
   $\rho_{\tau_s^C} \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \leq \mathcal{E}_{ij} - (\mathcal{C}' - \mathcal{C})d_{ij}$ and
   $\rho_{\tau_s^C} \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \geq \mathcal{E}_{ij} - (\mathcal{C}' - \mathcal{C})d_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$.

7. For all consecution transitions τ_s^C, we have $\rho_{\tau_s^C} \Rightarrow \forall \beta : 0 \leq \beta \leq |A_i| - 1 : A_i'[\beta] = A_i[\beta]$ for all $i \in \{1 \dots m\}$.

8. For all consecution transitions τ_s^C, we have

   - $\rho_{\tau_s^C} \wedge \mathcal{C}' > \mathcal{C} \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(\mathcal{C}' - 1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(\mathcal{C}' - 1) \leq 0$, if case 6a applies.
   - $\rho_{\tau_s^C} \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(\mathcal{C}' - \mathcal{C} - 1) \leq 0$, if case 6b applies.

9. For all consecution transitions τ_s^C, we have

- $\rho_{\tau_s^C} \wedge 0 \leq \alpha \leq \mathcal{C} - 1 \wedge x + \mathcal{B} + b_\alpha \alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha \alpha \leq 0$ for some fresh universally quantified variable x, if case 6a applies.
- $\rho_{\tau_s^C} \wedge 0 \leq \alpha \leq \mathcal{C} - 1 \wedge x + \mathcal{B} + b_\alpha \alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + \mathcal{C}' - \mathcal{C}) \leq 0$ for some fresh universally quantified variable x, if case 6b applies.

Then $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ is invariant.

Proof. First of all, let us remark that arrays are always sorted in non-decreasing order, and that their contents are never changed. This follows by induction from conditions 3 and 7. Moreover, it can also be seen from conditions 2 and 4 that $\mathcal{C} \geq 0$ is an invariant property.

Now, we will show that the property in the statement of the theorem holds after every initiation transition reaching our cutpoint and that it is maintained after every consecution transition going back to the cutpoint.

The first condition easily holds applying 2, since we have that $\rho_{\tau_r^I} \Rightarrow \mathcal{C}' = 0$ for every initiation transition τ_r^I, which implies $\forall \alpha : 0 \leq \alpha \leq \mathcal{C}' - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution transitions τ_s^C, we have $\rho_{\tau_s^C} \wedge \forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ implies $\forall \alpha : 0 \leq \alpha \leq \mathcal{C}' - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$.

By condition 4, we have that $\rho_{\tau_s^C} \Rightarrow \mathcal{C}' \geq \mathcal{C}$. We distinguish three cases:

1. $\mathcal{C}' = \mathcal{C}$ and case 6a holds. If $\mathcal{C}' = 0$ there is nothing to prove. Otherwise $\mathcal{C}' > 0$, and by hypothesis we have that $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$. Together with $\rho_{\tau_s^C}$, this implies $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ by instantiating appropriately x in condition 9.

   Now, let us show that for all $i \in \{1 \ldots m\}$, for all $j \in \{1 \ldots k\}$ and for all $\alpha \in \{0 \ldots \mathcal{C} - 1\}$ we have $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$. Let us consider three subcases:

   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots \mathcal{C} - 1\}$ we have $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}\alpha + \mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.

- $a_{ij} < 0$. Then ${\mathcal{E}'}_{ij} \geq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots \mathcal{C}-1\}$ we have $d_{ij}\alpha+{\mathcal{E}'}_{ij} \geq d_{ij}\alpha+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+{\mathcal{E}'}_{ij}] \geq A_i[d_{ij}\alpha+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order (note that, by condition 5, we have that $0 \leq d_{ij}\alpha + {\mathcal{E}'}_{ij} \leq |A_i| - 1 = |A'_i| - 1$, so array accesses are within bounds). Therefore $a_{ij}A_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] \leq a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.
- $a_{ij} = 0$. Then the inequality trivially holds.

Altogether we have that $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] + \mathcal{B}' + b_\alpha\alpha \leq 0$. Now our goal easily follows, taking into account that $\mathcal{C}' = \mathcal{C}$ and that by condition 7 we can replace A_i by A'_i.

2. $\mathcal{C}' > \mathcal{C}$ and case 6a holds. Then $\mathcal{C}' > 0$, and following the same argument as in the previous case we get that $\forall \alpha : 0 \leq \alpha \leq \mathcal{C} - 1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+{\mathcal{E}'}_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.

   It only remains to prove that $\forall \alpha : \mathcal{C} \leq \alpha \leq \mathcal{C}'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+{\mathcal{E}'}_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ (note that, by condition 5, we have that $0 \leq {\mathcal{E}'}_{ij} \leq |A'_i| - 1$ and $0 \leq d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij} \leq |A'_i| - 1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{\mathcal{C} \ldots \mathcal{C}' - 1\}$ and let us show that $a_{ij}A'_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] \leq a_{ij}A'_i[d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}]$ for all $i \in \{1 \ldots m\}$ and for all $j \in \{1 \ldots k\}$. We distinguish three cases:

   - $d_{ij} > 0$. Then $\alpha \leq \mathcal{C}' - 1$ implies $d_{ij}\alpha \leq d_{ij}(\mathcal{C}' - 1)$, and hence $d_{ij}\alpha + {\mathcal{E}'}_{ij} \leq d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] \leq A'_i[d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} < 0$. Then $\alpha \leq \mathcal{C}' - 1$ implies $d_{ij}\alpha \geq d_{ij}(\mathcal{C}' - 1)$, and hence $d_{ij}\alpha + {\mathcal{E}'}_{ij} \geq d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] \geq A'_i[d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} = 0$. The goal trivially holds.

   Thus $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha + {\mathcal{E}'}_{ij}] + \mathcal{B}' \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}(\mathcal{C}' - 1) + {\mathcal{E}'}_{ij}]+\mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq \mathcal{C}'-1$ implies

$b_\alpha\alpha \leq b_\alpha(\mathcal{C}'-1)$. Therefore $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A'_i[d_{ij}(\mathcal{C}'-1)+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha(\mathcal{C}'-1) \leq 0$ by condition 8.

3. $\mathcal{C}' > \mathcal{C}$ and case 6b holds (notice that $\mathcal{C}' = \mathcal{C}$ and case 6b together are not possible). By hypothesis we have $\forall\alpha : 0 \leq \alpha \leq \mathcal{C}-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}+b_\alpha\alpha \leq 0$. Together with $\rho_{\tau_s^C}$, this implies $\forall\alpha : 0 \leq \alpha \leq \mathcal{C}-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ by instantiating appropriately x in condition 9. By shifting the universally quantified variable the previous formula can be rewritten as $\forall\alpha : \mathcal{C}'-\mathcal{C} \leq \alpha \leq \mathcal{C}'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha(\alpha-(\mathcal{C}'-\mathcal{C})) \leq 0$.

   Now, let us show that for all $i \in \{1\dots m\}$, for all $j \in \{1\dots k\}$ and for all $\alpha \in \{\mathcal{C}'-\mathcal{C}\dots\mathcal{C}'-1\}$ we have $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]$. Let us consider three subcases:

   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}-(\mathcal{C}'-\mathcal{C})d_{ij}$ by condition 6. Hence for all $\alpha \in \{\mathcal{C}'-\mathcal{C}\dots\mathcal{C}'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \leq d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]$.
   - $a_{ij} < 0$. Then $\mathcal{E}'_{ij} \geq \mathcal{E}_{ij}-(\mathcal{C}'-\mathcal{C})d_{ij}$ by condition 6. Hence for all $\alpha \in \{\mathcal{C}'-\mathcal{C}\dots\mathcal{C}'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \geq d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \geq A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(\mathcal{C}'-\mathcal{C}))+\mathcal{E}_{ij}]$.
   - $a_{ij} = 0$. Then the inequality trivially holds.

   Altogether we have that $\forall\alpha : \mathcal{C}'-\mathcal{C} \leq \alpha \leq \mathcal{C}'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.

   We are left proving that $\forall\alpha : 0 \leq \alpha \leq \mathcal{C}'-\mathcal{C}-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k}a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ (note that, by condition 5, we have that $0 \leq \mathcal{E}'_{ij} \leq |A'_i|-1$ and $0 \leq d_{ij}(\mathcal{C}'-1)+\mathcal{E}'_{ij} \leq |A'_i|-1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{0\dots\mathcal{C}'-\mathcal{C}-1\}$ and let us show that $a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A'_i[d_{ij}(\mathcal{C}'-\mathcal{C}-1)+\mathcal{E}'_{ij}]$ for all $i \in \{1\dots m\}$ and for all $j \in \{1\dots k\}$. We distinguish three cases:

- $d_{ij} > 0$. Then $\alpha \leq \mathcal{C}' - \mathcal{C} - 1$ implies $d_{ij}\alpha \leq d_{ij}(\mathcal{C}' - \mathcal{C} - 1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A'_i[d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.

- $d_{ij} < 0$. Then $\alpha \leq \mathcal{C}' - \mathcal{C} - 1$ implies $d_{ij}\alpha \geq d_{ij}(\mathcal{C}' - \mathcal{C} - 1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \geq d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \geq A'_i[d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.

- $d_{ij} = 0$. The goal trivially holds.

Thus $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}(\mathcal{C}' - \mathcal{C} - 1) + \mathcal{E}'_{ij}] + \mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq \mathcal{C}' - \mathcal{C} - 1$ implies $b_\alpha\alpha \geq b_\alpha(\mathcal{C}' - \mathcal{C} - 1)$. Therefore $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}(\mathcal{C}' - \mathcal{C} - 1)\mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(\mathcal{C}' - \mathcal{C} - 1) \leq 0$ by condition 8.

□

By means of the previous theorem, (an equivalent version of) the desired invariants can be discovered. However, to the best of our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature. See Section 3.3.5 for other examples that can be handled by means of this extension.

3.3.5 Experimental evaluation

The method presented in Section 3.3.2 has been implemented in the tool CppInv[2]. The tool is able to generate automatically array invariants for wide a range of programs. The following table shows some of them, together with the corresponding loop invariants:

[2]The tool, together with a sample of example programs it can analyze, can be downloaded at www.cs.upc.edu/~albert/cppinv-ArrayInv.tar.gz.

Heap property:

```
const int N;
assume(N >= 0);
int A[2*N], i;
for (i = 0; 2*i+2 < 2*N; ++i)
  if (A[i]>A[2*i+1] or A[i]>A[2*i+2])
    break;
```

Loop invariants:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+2]$
$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+1]$

Partial initialization:
[Gopan et al., 2005]

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], i, j;
for (i = 0, j = 0; i < N; ++i)
  if (A[i] == B[i])
    C[j++] = i;
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \leq \alpha + i - j$
$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \geq \alpha$

Array palindrome:

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N/2; ++i)
  if (A[i] != A[N-i-1]) break;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = A[N-\alpha-1]$

Array initialization:
[Gopan et al., 2005]

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N; ++i)
  A[i] = 2*i+3;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 2\alpha + 3$

Array insertion:

```
const int N;
int A[N], i, x, j;
assume(0 <= i and i < N);
for (x = A[i], j = i-1;
     j >=0 and A[j] > x; --j)
  A[j+1] = A[j];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-j-2 : A[i-\alpha] \geq x+1$

Sequential initialization:
[Halbwachs and Péron, 2008]

```
const int N;
assume(N > 0);
int A[N], i;
for (i = 1, A[0] = 7; i < N; ++i)
  A[i] = A[i-1] + 1;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-2 : \ A[\alpha+1] = A[\alpha] + 1$

Array copy:
[Halbwachs and Péron, 2008]

```
const int N;
assume(N >= 0);
int A[N], B[N], i;
for (i = 0; i < N; ++i)
  A[i] = B[i];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = B[\alpha]$

First not null:
[Halbwachs and Péron, 2008]

```
const int N;
assume(N >= 0);
int A[N], s, i;
for (i = 0, s = N; i < N; ++i)
  if (s == N and A[i] != 0) {
    s=i;
    break;
  }
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 0$

Array partition:
[Beyer et al., 2007c]

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], a, b, c;
for (a=0, b=0, c=0; a < N; ++a)
  if (A[a] >= 0) B[b++]=A[a];
  else           C[c++]=A[a];
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq b-1 : B[\alpha] \geq 0$
$\forall \alpha : 0 \leq \alpha \leq c-1 : C[\alpha] < 0$

Array maximum:
[Halbwachs and Péron, 2008]

```
const int N;
assume(N > 0);
int A[N], i, max;
for (i = 1, max = A[0]; i < N; ++i)
  if (max < A[i])
    max = A[i];
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq i-1 : A[\alpha] \leq max$

First occurrence:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l, u;
// A is sorted in ascending order
for (l = 0, u = N; l < u; ) {
  int m = (l+u)/2;
  if (A[m] < x) l = m+1;
  else u = m;   }
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] < x$
$\forall \alpha : 0 \leq \alpha \leq N-1-u : A[N-1-\alpha] \geq x$

Sum of pairs:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l = 0, u = N-1;
// A is sorted in ascending order
while (l < u)
  if      (A[l] + A[u] < x) l = l+1;
  else if (A[l] + A[u] > x) u = u-1;
  else break;
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] + A[u] < x$
$\forall \alpha : 0 \leq \alpha \leq N-u-2 : A[N-1-\alpha] + A[l] > x$

As a final experiment, we have run CppInv over a collection of programs written by students. It consists of 38 solutions to the problem of finding the first occurrence of an element in a sorted array of size N in $\mathcal{O}(\log N)$ time. These solutions have been taken from the online learning environment for computer programming Jutge.org (see www.jutge.org), which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite corresponds to all submitted iterative programs that have been accepted, i.e., such that for all input tests the output matches the expected one. These programs can be considered more realistic code than the examples above (**First occurrence** program), since most often they are not the most elegant solution but a working one with many more conditional statements than necessary. For example, here is an instance of such a program:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int e = 0, d = N - 1, m, pos;
  bool found = false, exit  = false;
  while (e <= d and not exit) {
    m = (e+d)/2;
    if      (x > A[m]) {
      if (not found) e = m+1;
      else exit = true;
    }
    else if (x < A[m]) {
      if (not found) d = m-1;
      else exit = true;
    }
    else {
      found = true; pos = m; d = m-1;
    }
  }
  if (found) {
    while (x  == A[pos-1]) --pos;
    return pos;
  }
  return -1;
}
```

This particular example is interesting because, with the aid of our tool, we realized that it does not work in $\mathcal{O}(\log N)$ time as required, and is thus a false positive. Namely, our tool produces the following invariants for the first loop:

$$\begin{array}{lrll} \forall\alpha: & 0 & \leq \alpha & \leq e-1 \quad : A[\alpha] < x\,, \\ \forall\alpha: & d+1 & \leq \alpha & \leq N-1 \quad : A[\alpha] \geq x\,. \end{array}$$

By manual inspection one can see that $found \rightarrow (A[pos] = x \wedge d = pos - 1)$ and $exit \rightarrow found$ are also invariant. Therefore, if on exit of the loop the property $e \leq d$ holds, then *exit* and *found* are true and, with all this information, it is unknown whether the contents of the array between e and $pos-1$ are equal to x. Since this segment can be arbitrarily long, the second loop may take $\mathcal{O}(N)$ time to find the first occurrence of x. This reasoning allowed us to cook an input for which indeed the program behaves linearly. On the other hand, by means of the generated invariants it can be seen that the problem is that the loop may be exited too early, and that by replacing in the first loop the body of the first conditional by `e = m+1` and the second one by `d = m-1`, the program becomes correct and meets the complexity requirements.

In general, for all programs in the benchmark suite our tool was able to find automatically both standard invariants. The time consumed was very different depending mainly on how involved the code was. The number of looping transitions for these benchmarks ranged from 6 to 36. The main problem as regards efficiency is that in its current form our prototype exhaustively generates first all scalar invariants and then, using all of them, generates all array invariants.

Execution times can be improved by annotating the code with instances of templates where the unknown parameters are bounded. In this setting, runtimes varied from 10 to 108 seconds with an average time of 36.27 seconds. A correlation between the runtime and the number of transitions could be established for almost all cases. However, since the current procedure exhaustively looks for all the solutions, execution times also depend on the number of alternative invariants to the standard properties. Anyway, further work is needed to heuristically guide the search of scalar invariants, so that only useful information is inferred.

We also applied our tool to some of the submissions rejected by Jutge.org. In some cases the generated invariants helped us to fix the program. E.g.,

for the following code:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int i = 0, j = N-1;
  while (i <= j) {
    if (x == A[i]) return  i;
    if (x < A[i])  return -1;
    int m = (i+j)/2;
    if (x < A[m]) j = m-1;
    else          i = m+1; }
  return -1; }
```

In this case, the generated invariants are:

$$\begin{aligned} \forall\alpha: \quad & 0 \leq \alpha \leq i-1 \quad : A[\alpha] \leq x\,, \\ \forall\alpha: \quad & j+1 \leq \alpha \leq N-1 \quad : A[\alpha] > x\,. \end{aligned}$$

One may notice that the first invariant should have a strict inequality, and that this problem may be due to a wrong condition in the last conditional. Indeed, by replacing the condition $x < A[m]$ by $x \leq A[m]$, we obtain a set of invariants that allow proving the correctness of the program.

3.3.6 Related work comparison

Some of the techniques for the synthesis of quantified invariants for programs with arrays fall into the framework of *abstract interpretation*, as it was explained in Section 3.1.1. In comparison with the techniques presented in this work, the approaches of [Gopan et al., 2005; Halbwachs and Péron, 2008] force all array accesses to be of the form $\alpha + k$. As a consequence, programs like *Array palindrome* or *Heap property* (see Section 3.3.5) cannot be handled. Moreover, the universally quantified variable is not allowed to appear outside array accesses. For this reason, our analysis can be more precise, e.g., in the *Array initialization* and the *Partial initialization* [Gopan et al., 2005] examples. Another technique based on abstract interpretation is presented in [Gulwani et al., 2008a]. While their approach can discover more general properties than the one described in this work, it requires that the user provides templates to guide the analysis.

Unlike most of the *predicate abstraction*-based techniques (see Section 3.1.2), our approach does not require programs to be annotated with assertions, thus allowing one to analyze code embedded into large programs, or

with predicates, which sometimes require ingenuity from the user. To alleviate the need of supplying predicates, in [Cousot, 2004] *parametric predicate abstraction* was introduced. However, the properties considered there express relations between all elements of two data collections, while our approach is able to express pointwise relations.

Another group of techniques is based on *first-order theorem proving* (see Section 3.1.4). One of the problems of the methods described in [Kovács and Voronkov, 2009; Hoder et al., 2011] is the limited capability of arithmetic reasoning of the theorem prover (as opposed to SMT solvers, where arithmetic reasoning is hard-wired in the theory solvers). The approach described in [McMillan, 2008], based on interpolating theorem proving, in addition to suffering from similar arithmetic reasoning problems as [Kovács and Voronkov, 2009], also requires program assertions.

Other methods use *computational algebra* (see Section 3.1.3). One of the limitations of [Henzinger et al., 2010a] is that array variables are required to be either write-only or read-only. Hence, unlike our method, programs such as *Sequential initialization* [Halbwachs and Péron, 2008] and *Array insertion* (see Section 3.3.5) cannot be handled.

Finally, a technique that belongs to the *constraint-based* methods (see Section 3.1.5) and covers the array property fragment [Bradley et al., 2006] is presented in [Beyer et al., 2007b]. The language of the invariants generated in this thesis is outside the array property fragment, since we can generate properties where indices do not necessarily occur in array accesses (e.g., see the *Array initialization* or the *Partial initialization* examples in Section 3.3.5).

Chapter 4

Termination Proving

Termination analysis is critical to the process of ensuring the stability and usability of software systems. Termination bugs are difficult to trace and are hardly notified: as they do not arise as system failures but as unresponsive behavior, when faced to them users tend to restart their devices without reporting to software developers.

Despite Alan Turing showed that the termination problem is undecidable [Turing, 1936], recent research advances make practical termination proving tools possible [Cook et al., 2011]. Turing's major result proved that we *cannot* build a procedure which determines *for all* programs whether or not a given program will always finish running, however, we can construct one that is able to solve the problem for a large set of programs of interest answering "unknown" otherwise.

Since techniques to prove termination are incomplete, failure to prove termination does not immediately indicate the existence of a non-terminating execution. Therefore, all methods in practice are focused on either proving termination or non-termination.

4.1 Termination and non-termination

When proving non-termination [Gupta et al., 2008; Velroyen and Rümmer, 2008; Brockschmidt et al., 2012] the goal is to find a counterexample, an input for which program execution reaches a state that is infinitely visited. Thus, this approximation is centered on finding bugs. On the other hand, the aim of proving termination is to find a formal proof that a program

always terminates. This ensures the absence of *all* termination bugs. A potential drawback of this approach is that generating a proof is in general harder than finding a counterexample, although in termination analysis this is not always the case.

The classical method to prove termination, proposed by Turing [Turing, 1949], is to find a *ranking* function, a function that maps every program state to a value in a well-order and decreases for every possible program transition. Since there are no infinite descending chains with respect to a well-founded relation, if we find a ranking function, we can conclude that the program must eventually terminate.

Due to the fact that program state and control flow usually depend on integer variables, natural numbers are often chosen as the well-order for the termination argument. In this case, the goal is to find an integer expression over the program variables that decreases in every iteration and its lower bounded by zero.

Example 4.1. Consider the transition system of Fig. 2.6. The function $z - y$ is a ranking function for the loop because its value decreases after every iteration and it is lower bounded by 0 (recall only transition $\tau_{3.1}$ is feasible).

4.2 Termination arguments

The problem with Turing's method is that finding a *single* ranking function for each program loop (a strongly connected component) is typically difficult, even for simple programs. Furthermore, in some cases no function into natural numbers exists that suffices to prove termination, which forces to use ranking functions into well-orders that are much more complex than the natural numbers.

A kind of functions mapped into such well-orders are *lexicographic* functions, which are obtained as the result of the cartesian product of well-ordered sets. The fundamental property of lexicographical orders that makes them so useful is that it preserves well-orders of finite products and, therefore, we can construct ranking functions as the composition of individual ranking functions provided that they form a lexicographical order [Bradley et al., 2005; Colón and Sipma, 2002; Cook et al., 2013].

The process of proving termination is usually split into two tasks, namely, searching for a candidate function and then checking whether it is a valid ranking function. Finding candidates for both single and lexicographic functions is the difficult part of proving termination, whereas checking that a given candidate is actually a ranking function is the easy one.

This situation was reversed with the emergence of a new kind of termination arguments, the so-called *disjunctive* termination arguments [Cook et al., 2006; Tsitovich et al., 2011]. The idea behind this approach is to build, in the manner of lexicographic functions, a termination argument from a set of individual ranking functions. The difference is that, in this case, the composition is constructed directly by the union of well-founded relations. Since the union of well-founded relations is not in general a well-founded relation, it is necessary to impose an additional condition, specifically, that the disjunctive termination argument must hold not only between the states before and after any *single* iteration of each loop, but before and after *any number* of iterations of every loop.

The advantage of disjunctive termination arguments is that finding each component of the termination argument is easier than finding a single candidate that covers all transitions or a lexicographic order for every individual ranking function. However, now checking if the termination argument is valid requires much effort. In particular, the check is usually codified as a problem of reachability of some program locations.

Although the trend during the last few years have been to develop techniques based on disjunctive termination arguments [Chawdhary et al., 2008; Berdine et al., 2007; Cook et al., 2007; Manolios and Vroon, 2006; Berdine et al., 2006; Cook et al., 2006; Podelski and Rybalchenko, 2005], a recent paper [Cook et al., 2013] presents evidences that there is still much more room for improvement in techniques based on lexicographic ranking functions. In particular, the author presents an improved method for searching lexicographic ranking functions achieving important speedups with respect to the dominant technique.

For all the previous techniques, the synthesis of termination arguments is tackled mainly using approaches for invariant inference like constraint solving and abstract interpretation which were described in Chapter 3. Note the similarity between the problem of finding an invariant and a ranking function. For instance, in the latter case we could want to discover a function

$f(\overline{x})$ such that $f(\overline{x}) \geq 0$ and $f(\overline{x}) > f(\overline{x}')$ after an iteration, which could be seen as the same as finding an loop invariant without assuming its satisfaction at the beginning. Moreover, in order to infer that a function maps into a well-order, it is usually necessary to discover *supporting* invariants that help to prove some of the ranking function conditions.

4.3 Proving termination using Max-SMT

In this section we show how Max-SMT can be exploited when applying the constraint-based method for proving program termination. We assume that every program is modeled with a transition system (see Section 2.2), only *scalar* variables $\overline{v}$ are declared in the program, and they are the only variables of the transition system, i.e., $\mathcal{V} = \overline{v}$.

4.3.1 Basis of the termination argument

The basic idea of the approach we follow for proving program termination [Colón and Sipma, 2002] is to argue by contradiction that no transition is infinitely executable. First of all, no disabled transition can be infinitely executable trivially. Moreover, one just needs to focus on transitions joining locations in the same strongly connected component (SCC): if a transition is executed over and over again, then its pre and post locations must belong to the same SCC. So let us assume that one has found a *ranking function* for such a transition τ, according to:

Definition 4.2. Let $\tau = (\ell, \ell', \rho)$ be a transition such that ℓ and ℓ' belong to the same SCC, denoted by C. A function $R : \overline{v} \to \mathbb{Z}$ is said to be a *ranking function* for τ if:

- **[Boundedness]** $\rho \models R \geq 0$
- **[Strict Decrease]** $\rho \models R > R'$
- **[Non-increase]** $\forall . \hat{\tau} = (\hat{\ell}, \hat{\ell}', \hat{\rho}) \in \mathcal{T}$ such that $\hat{\ell}, \hat{\ell}' \in C$: $\hat{\rho} \models R \geq R'$

Note that boundedness and strict decrease *only* depend on τ, while non-increase depends on *all* transitions in the SCC.

The key result is that if $\tau = (\ell, \ell', \rho)$ admits a ranking function R, then it is finitely executable. Indeed, first notice that if one can execute τ from a

state (ℓ, σ) then $R(\sigma) \geq 0$, because of boundedness. Also, the value of R at the states along any path contained in C cannot increase, thanks to the non-increase property. Moreover, in any cycle contained in C traversing τ, the value of R strictly decreases, due to the strict decrease property. Now, let us assume that there was a computation where τ was executed infinitely. Such a computation would eventually visit only locations in C. Because of the previous observations, by evaluating R at the states at which τ is executed we could construct an infinitely decreasing sequence of non-negative integers, a contradiction.

Finitely executable transitions can be safely removed from the transition system as regards termination analysis. This in turn may break the SCC's into smaller pieces. If by applying this reasoning recursively one can prove that all transitions are finitely executable, then the transition system is terminating.

4.3.2 Supporting invariants

Invariant maps (see Section 3.2) are fundamental when analyzing program termination. For instance, a transition $\tau = (\ell, \ell', \rho)$ is proved to be disabled if there is an invariant $\mu(\ell)$ at location ℓ such that $\mu(\ell) \wedge \rho$ is unsatisfiable. In general, if μ is an invariant map, then any transition $\tau = (\ell, \ell', \rho)$ can be safely strengthened by replacing the transition relation ρ by $\mu(\ell) \wedge \rho$.

In [Colón and Sipma, 2002] linear invariants are exhaustively computed before termination analysis. In the same paper a heuristic approach is also presented, which only requires a light-weight invariant generator by restricting to single-variable ranking functions. Another solution is proposed in [Bradley et al., 2005], where invariant generation is not performed eagerly but on demand. By formulating both invariant and ranking function synthesis as constraint problems, both can be solved simultaneously, so that only the necessary supporting invariants for the targeted ranking functions –namely, *lexicographic linear ranking functions*– need to be discovered.

4.3.3 Illustration of the Max-SMT method

Based on [Colón and Sipma, 2002; Bradley et al., 2005], we present a Max-SMT constraint-based approach for proving termination. The crucial observation in our method is that, albeit our goal is to show that transitions

```
int main() {
    int x, y, z;
ℓ1:  while (y ≥ 1) {
        x--;
    ℓ2:  while (y < z) {
            x++; z--;
        }
    y = x + y;
    } }
```

$\Theta(\ell_1) \equiv \mathit{true}$

$\Theta(\ell_2) \equiv \mathit{false}$

$$\rho_{\tau_1}: \; y \geq 1, \;\; x' = x - 1, \;\; y' = y, \;\; z' = z$$
$$\rho_{\tau_2}: \; y < z, \;\; x' = x + 1, \;\; y' = y, \;\; z' = z - 1$$
$$\rho_{\tau_3}: \; y \geq z, \;\; x' = x, \;\; y' = x + y, \;\; z' = z$$

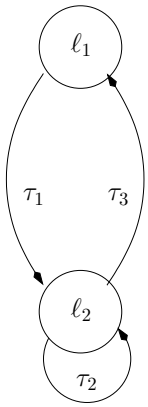


Figure 4.4. Program and its transition system.

cannot be executed infinitely by finding a ranking function or an invariant that disables them, if we only discover an invariant, or an invariant and a *quasi-ranking function* that almost fulfills all needed properties for well-foundedness, we have made some progress: either we can remove *part of a transition* and/or we have improved our knowledge on the behavior of the program. A natural way to implement this idea is by considering that some of the constraints are *hard* (the ones guaranteeing invariance) and others are *soft* (those guaranteeing well-foundedness) in a Max-SMT framework. Moreover, by giving different weights to the constraints we can set priorities and favor those invariants and (quasi-) ranking functions that lead to the furthest progress.

Example 4.3. Let us show the first rounds of the termination analysis of the program in Fig. 4.4. In the first round, the solver finds the invariant $y \geq 1$ at ℓ_2 and the ranking function z for τ_2. While $y \geq 1$ can be added to τ_3 (resulting into a new transition τ_3'), the ranking function allows eliminating τ_2 from the termination transition system (see Fig. 4.6 (b)).

In the second round, the solver cannot find a ranking function. However, thanks to the Max-SMT formulation, it can produce the quasi-ranking function x, which is non-increasing and strict decreasing for τ_1, but not bounded. This quasi-ranking function can be used to split transition τ_1 into two new transitions $\tau_{1.1}$ and $\tau_{1.2}$ as follows:

$$\rho_{\tau_{1.1}}: \; \boldsymbol{x \geq 0}, \;\; y \geq 1, \;\; x' = x - 1, \;\; y' = y, \;\; z' = z$$
$$\rho_{\tau_{1.2}}: \; \boldsymbol{x < 0}, \;\; y \geq 1, \;\; x' = x - 1, \;\; y' = y, \;\; z' = z$$

Then $\tau_{1.1}$ is immediately removed, since x is a ranking function for it. The current termination transition system is given in Fig. 4.6 (c).

Further refinements are possible. E.g., the termination transition system can also be used for generating properties that are guaranteed to eventually hold at a location for some computations. More specifically, we devised the following light-weight approach for generating what we call *termination implications*. The rationale is that, if we find a property J_ℓ that is implied by all transitions going into ℓ and ℓ is finally reached, then J_ℓ must hold. Then this termination implication can be propagated forward to the transitions going out from ℓ.

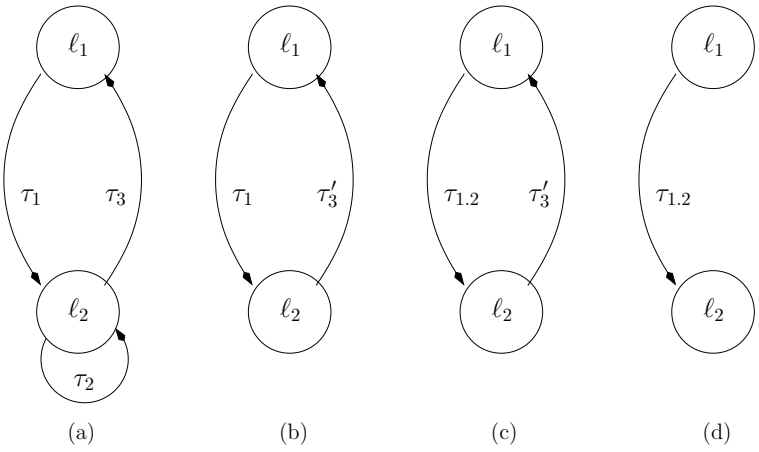
Example 4.5. Let us carry on with the termination analysis started in example 4.3. In the third and final round, the termination implication $x < 0$ is generated at ℓ_2, together with the ranking function y for transition τ_3'. Note that the termination implication is crucial to prove the strict decrease of y for τ_3', and that the previously generated invariant $y \geq 1$ at ℓ_2 is needed to ensure boundedness. Now τ_3' can be removed, which makes the graph acyclic (see Fig. 4.6 (d)). This concludes the termination proof, which consist of the composition of three ranking functions, namely, (x, y, z).

4.3.4 Formal description of the Max-SMT method

In this section we first describe a constraint-based method for termination analysis that uses SMT and identify some of its shortcomings. Then we show how Max-SMT can be used to overcome these limitations.

An SMT approach to proving termination

Following the approach described in Section 4.3.1 [Colón and Sipma, 2002], to show that a transition τ is finitely executable and thus discard it, one needs either a disability argument or a ranking function for it. To this end we construct a constraint system, i.e. an SMT formula, whose solutions correspond to either an invariant that proves disability, or a ranking function. Given an SCC, the constraint system, if satisfiable, will allow discarding (at least, but possibly more than) one of the transitions in the SCC. By iterating this procedure until no cycles are left we will obtain a termination argument for the SCC.



$\Theta(\ell_1) \equiv \mathit{true} \qquad \Theta(\ell_2) \equiv \mathit{false}$

$$
\begin{array}{lllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_{1.2}}: \ x < 0, & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z \\
\rho_{\tau_3'}: \ y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

Figure 4.6. Evolution of the termination transition system: initially (a) and after the first (b), second (c) and third (d) round.

To construct the constraint system, first of all we consider:

- for each location ℓ, a linear invariant template $I_\ell(\overline{v}) \equiv i_{\ell,0} + \sum_{v \in \overline{v}} i_{\ell,v} \cdot v \leq 0$, where $i_{\ell,0}$, $i_{\ell,v}$ are unknown;

- a linear ranking function template $R(\overline{v}) \equiv r_0 + \sum_{v \in \overline{v}} r_v \cdot v$, where r_0, r_v are unknown.

Recall that ranking functions are associated to transitions, not to locations. However, instead of introducing a template for each transition, we just have one single template, which, if the constraint system has a solution, will be a ranking function for a transition *to be determined by the solver*.

Similarly to [Bradley et al., 2005], we take the following constraints from the definitions of inductive invariant and ranking function:

Initiation:	For $\ell \in \mathcal{L}$:	$\mathbb{I}_\ell \stackrel{def}{=} \Theta(\ell) \vdash I_\ell$
Disability:	For $\tau = (\ell, \ell', \rho) \in \mathcal{T}$:	$\mathbb{D}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash 1 \leq 0$
Consecution:	For $\tau = (\ell, \ell', \rho) \in \mathcal{T}$:	$\mathbb{C}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash I'_{\ell'}$
Boundedness:	For $\tau = (\ell, \ell', \rho) \in \mathcal{T}$:	$\mathbb{B}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq 0$
Strict Decrease:	For $\tau = (\ell, \ell', \rho) \in \mathcal{T}$:	$\mathbb{S}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R > R'$
Non-increase:	For $\tau = (\ell, \ell', \rho) \in \mathcal{T}$:	$\mathbb{N}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq R'$

Let L and T be the sets of locations and transitions in the SCC in hand, respectively. Let also P be the set of *p*ending transitions, i.e., which have not been proved to be finitely executable yet. Then we build the next constraint system:

$$\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge \bigvee_{\tau \in P} (\mathbb{D}_\tau \vee (\mathbb{B}_\tau \wedge \mathbb{S}_\tau)) \wedge ((\bigwedge_{\tau \in P} \mathbb{N}_\tau) \vee \bigvee_{\tau \in P} \mathbb{D}_\tau).$$

The first two conjuncts guarantee that an invariant map is computed; the other two, that at least one of the pending transitions can be discarded. Notice that, if there is no disabled transition, we ask that *all* transitions in P are non-increasing, but only that at least *one* transition in P (the next to be removed) is both bounded and strict decreasing. Note also that for finding invariants one has to take into account *all* transitions in the SCC, even those that have already been proved to be finitely executable: otherwise some reachable states might not be covered, and the invariant generation would become unsound. Hence in our termination analysis we consider two transition systems: the original transition system for invariant synthesis, whose transitions are T and which remains all the time the same; and the *termination transition system*, whose transitions are P, i.e., where transitions already shown to be finitely executable have been removed. This duplication is similar to the *cooperation graph* of [Brockschmidt et al., 2013].

However, this first approach is problematic when a ranking function needs several invariants. A possible solution is to add more templates iteratively, so that for example initially invariants consisting of a single linear inequality are tried, if unsuccessful then invariants consisting of a conjunction of two linear inequalities are tried, etc. But when proceeding in this way, all problems before the right number of invariants is found are unsatisfiable. This is wasteful, as no constructive information is retrieved from unsatisfiable constraint systems.

Another problem with this method for analyzing termination is that the kind of termination proofs it yields may be too restricted. More specifically, when one proves that a transition τ is finitely executable, then a single termination argument shows there is no computation where τ appears infinitely. Although this produces compact proofs, on the other hand sometimes there may not exist such a unique reason for termination, and it becomes necessary a more fine-grained examination. However, the approach as presented so far does not provide a natural way or guidance for refining the analysis.

A Max-SMT approach to proving termination

The main contribution of our work is to show that the constraint system can be expressed in such a way that, even when it turns out to be unsatisfiable, some information useful for refining the termination analysis can be obtained. The key observation is that, even though our aim is to prove transitions to be finitely executable (by finding a ranking function or an invariant that disables them), if we just find an invariant, or an invariant and a *quasi-ranking function* that is close to fulfill all required conditions, we have progressed in our analysis.

The idea is to consider the constraints guaranteeing invariance as *hard*, so that any solution to the constraint system will satisfy them, while the rest are *soft*. Let us consider propositional variables $p_{\mathbb{B}}$, $p_{\mathbb{S}}$ and $p_{\mathbb{N}}$, which intuitively represent if the conditions of boundedness, strict decrease and non-increase in the definition of ranking function are violated respectively, and corresponding weights $\omega_{\mathbb{B}}$, $\omega_{\mathbb{S}}$ and $\omega_{\mathbb{N}}$. We consider now the next constraint system (where soft constraints are written $[\cdot, \omega]$, and hard ones as usual):

$$\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} \Big(\mathbb{D}_\tau \vee \mathbb{C}_\tau \Big) \wedge \bigvee_{\tau \in P} \Big(\mathbb{D}_\tau \vee \big((\mathbb{B}_\tau \vee p_{\mathbb{B}}) \wedge (\mathbb{S}_\tau \vee p_{\mathbb{S}}) \big) \Big) \wedge$$
$$\Big(\big(\bigwedge_{\tau \in P} \mathbb{N}_\tau \big) \vee \bigvee_{\tau \in P} \mathbb{D}_\tau \vee p_{\mathbb{N}} \Big) \wedge [\neg p_{\mathbb{B}}, \omega_{\mathbb{B}}] \wedge [\neg p_{\mathbb{S}}, \omega_{\mathbb{S}}] \wedge [\neg p_{\mathbb{N}}, \omega_{\mathbb{N}}].$$

Note that ranking functions have cost 0, and (if no transition is disabled) functions that fail in any of the conditions are penalized with the respective weight. Thus, the Max-SMT solver looks for the best solution and gets a ranking function if feasible; otherwise, the weights guide the search to get invariants and quasi-ranking functions that satisfy as many conditions as possible.

Hence this Max-SMT approach allows recovering information even from problems that would be unsatisfiable in the initial method. This information can be exploited to perform dynamic trace partitioning [Mauborgne and Rival, 2005] as follows. Assume that the optimal solution to the above Max-SMT formula has been computed, and let us consider a transition $\tau \in P$ such that $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ evaluates to true in the solution. Then we distinguish several cases depending on the properties satisfied by τ and the computed function R:

- If τ is disabled then it can be removed.
- If R is non-increasing and satisfies boundedness and strict decrease for τ, then τ can be removed too: R is a ranking function for it.
- If R is non-increasing and satisfies boundedness for τ but not strict decrease, one can split τ in the termination transition system into two new transitions: one where $R > R'$ is added to τ, and another one where $R = R'$ is enforced. Then the new transition with $R > R'$ is automatically eliminated, as R is a ranking function for it. Equivalently, this can be seen as adding $R = R'$ to τ. Now, if the solver could not prove R to be a true ranking function for τ because it was missing an invariant, this transformation will guide the solver to find that invariant so as to disable the transition with $R = R'$.
- If R is non-increasing and satisfies strict decrease for τ but not boundedness, the same technique from above can be applied: it boils down to adding $R < 0$ to τ.
- If R is non-increasing but neither strict decrease nor boundedness are fulfilled for τ, then τ can be split into two new transitions: one with $R < 0$, and another one with $R \geq 0 \wedge R = R'$.
- If R does not satisfy the non-increase property, then it is rejected; however, the invariant map from the solution can be used to strengthen the transition relations for the following iterations of the termination analysis.

Note this analysis may be worth applying on other transitions τ in the termination transition system apart from those that make $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge$

$(\mathbb{S}_\tau \vee p_{\mathbb{S}}))$ true. E.g., if R is a ranking function for a transition τ but fails to be so for another one τ' because strict decrease does not hold, then, according to the above discussion, τ' can be strengthened with $R = R'$.

On the other hand, working in this iterative way requires imposing additional constraints to avoid getting to a standstill. Namely, in the case where non-increase does not hold and so one would like to exploit the invariant, it is necessary to impose that the invariant is not redundant. More in detail, let us consider a fixed location ℓ, and let $I_\ell^{(1)}, \ldots, I_\ell^{(k)}$ be the previously computed invariants at location ℓ. Then I_ℓ, the invariant to be generated at ℓ, is redundant if it is implied by $I_\ell^{(1)}$, ..., $I_\ell^{(k)}$, i.e., if $\mathbb{E}_\ell \stackrel{def}{=} \forall \overline{v}\ (I_\ell^{(1)}(\overline{v}) \wedge \ldots \wedge I_\ell^{(k)}(\overline{v}) \rightarrow I_\ell(\overline{v}))$. So we impose $p_{\mathbb{N}} \rightarrow \neg \bigwedge_{\ell \in L} \mathbb{E}_\ell$ to ensure that violating non-increase leads to non-redundant invariants. Conditions are added similarly to avoid redundant quasi-ranking functions.

Another advantage of this Max-SMT approach is that by using different weights we can express priorities over conditions. Since, as explained above, violating the property of non-increase invalidates the computed function R, it is convenient to make $\omega_{\mathbb{N}}$ the largest weight. On the other hand, when non-increase and boundedness are fulfilled but not strict decrease an equality is added to the transition, whereas when non-increase and strict decrease are fulfilled but not boundedness just an inequality is added. As we prefer the former to the latter, in our implementation (see Section 4.3.7) we set $\omega_{\mathbb{B}} > \omega_{\mathbb{S}}$.

A further improvement is the generation of *termination implications*. A termination implication at a location ℓ is an assertion $J(\overline{v})$ such that any transition in the *termination transition system* that leads into ℓ implies it, i.e., it holds that $\rho \models J(\overline{v}')$, where ρ is the relation of the transition. Thus, J will *eventually* hold when ℓ is reached (although, unlike ordinary invariants, may not initially be true; see Section 4.3.3). Hence, it can be propagated forward in the termination transition system to the transitions going out from ℓ. To produce termination implications, for each location ℓ a new linear inequality template $J_\ell(\overline{v})$ is introduced and the following constraint is imposed: $\bigwedge_{\tau=(\hat{\ell},\ell,\rho)\in P}(\mathbb{D}_\tau \ \vee \ I_{\hat{\ell}} \wedge \rho \vdash J'_\ell)$. Additional constraints are enforced to ensure that new termination implications are not redundant with the already computed invariants and termination implications.

4.3.5 Related work

Our research builds upon [Bradley et al., 2005], where the constraint-based method (see Section 3.2) was first applied to termination. However, we extend this work in several aspects. First, in that approach only linear programs with unnested loops can be handled, while we can deal with arbitrary control structures. Moreover, in [Bradley et al., 2005] the generation of their lexicographic ranking functions requires a higher-level loop that, before sending the constraint problem to the solver, determines the precedence of the transitions in the lexicographic order. On the other hand, in our approach this outer loop is not needed. Finally, thanks to assigning weights to the constraints, unlike [Bradley et al., 2005] we do not need to stipulate the number of supporting invariants that will be needed a priori, and hence our constraint problems are simpler. Further, weights allow us to guide the solving engine in the search of appropriate ranking functions and invariants.

In [Cook et al., 2013], the lexicographic approach of [Bradley et al., 2005] is extended so as to handle programs with complex control flow. However, their method still requires to search for the right ordering of the transitions in order to obtain a successful termination proof. Moreover, in this technique the procedures for synthesizing ranking functions and auxiliary invariants do not share enough information, while in this thesis these mechanisms are tightly coupled. Finally, in [Brockschmidt et al., 2013] a method closely related to ours is presented. Both approaches, which have been developed independently, go in the same direction of achieving a better cooperation between the invariant and the ranking function syntheses. Still, a significant difference is that we can exploit the quasi-ranking functions produced in the absence of ranking functions in order to progress in the termination analysis.

4.3.6 Implementation

The method presented in Section 4.3.4 has been implemented in the tool CppInv[1]. This section describes this implementation.

CppInv admits code written in C++ as well as in the language of T2 [Cook et al., 2013]. The system analyses programs with integer variables, linear expressions and function calls. Variables of other data types, such

[1] CppInv, together with all benchmarks used in the experimental evaluation of Section 4.3.7, is available at `www.cs.upc.edu/~albert/cppinv-Term.tar.gz`.

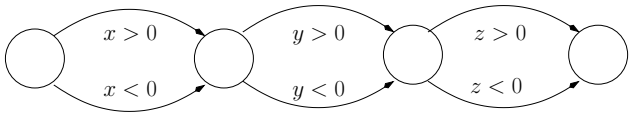


Figure 4.7. Chain of locations obtained from a sequence of statements `assume`($x \neq 0$); `assume`($y \neq 0$); `assume`($z \neq 0$). Note disequalities are not natively supported, and so have to be split into disjunctions of inequalities.

as floating-point variables, are treated as unknown values. Function calls are handled with techniques similar to those in [Cook et al., 2009], although currently the returned value is ignored. Further, for recursive functions, after a function call we assign unknowns to all variables that can be modified in the call (i.e., global variables and variables passed by reference).

In the transformation from the source code to the internal transition system representation, CppInv attempts to reduce the number of locations by composing transitions. Still, this preprocessing may result in an exponential growth in the number of transitions. As our technique does not require minimized transition systems for soundness, the tool stops this location minimization if a threshold number of transitions is reached. Moreover, whenever a chain of locations connected by transitions that do not modify variables (see Fig. 4.7) is detected, CppInv does not attempt to eliminate the locations: since no variable is updated, in these transitions any function satisfies the non-increase condition, while no ranking function is possible. For this reason, when producing the constraints, these transitions are ignored as far as termination is concerned, and are only considered for the generation of invariants.

Once the input is represented as a transition system, the actual termination analysis starts. See procedure ProvedTransSysTerm in Algorithm 4.8.

The SCC's are computed and topologically sorted, and each SCC is processed according to this order. Processing an SCC involves first performing a copy of the transitions for keeping track of those not proven finitely executable yet. Then the initial conditions are updated with the strongest postconditions of the incoming transitions from previous SCC's, where the strongest postcondition of a transition relation $\rho(\overline{v}, \overline{v}')$ is the assertion $\mathrm{SPost}(\rho)(\overline{v}) \equiv \exists\, \overline{w}\ \rho(\overline{w}, \overline{v})$. Finally the SCC is analysed for termination. If it could not be proved terminating, the procedure stops. Otherwise the next SCC is dealt with.

Algorithm 4.8 Proc. ProvedTransSysTerm

Input: Transition System $\mathcal{S} = (\overline{v}, \mathcal{L}, \Theta, \mathcal{T})$
Output: **true** if transition system $\mathcal{S}$ can be proved terminating

1: $(\mathcal{C}, \prec) \leftarrow \mathsf{ComputeSCCsAndTopologicallySort}(\mathcal{S})$ {$\mathcal{C}$ is the list of SCC's topologically sorted according to ordering $\prec$}
2: **for all** $(C \in \mathcal{C}$ **by** $\prec)$ **do**
3: $(L, T) \leftarrow (\mathsf{Locations}(C), \mathsf{Transitions}(C))$
4: $P \leftarrow \mathsf{Copy}(T)$
5: **for all** $(\ell \in L : \exists(\hat{\ell}, \ell, \rho) \in \mathcal{T}$ with $\hat{\ell} \in \widehat{C} \prec C)$ **do**
6: $\Theta(\ell) \leftarrow \Theta(\ell) \vee \mathrm{SPost}(\rho)$
7: **end for**
8: **if not** $\mathsf{ProvedSCCTerm}(L, T, P)$ **then**
9: **return false**
10: **end if**
11: **end for**
12: **return true**

Procedure ProvedSCCTerm in Algorithm 4.9 orchestrates the analysis of termination of SCC's. It takes as arguments: a set of locations L and a set of transitions T, corresponding to an SCC of the transition system; and the *termination transition system*: a non-empty set $P \subseteq T$ of transitions that still have to be proved finitely executable. As explained in Section 4.3.1, one may assume that the graph induced by P is strongly connected. The function returns **true** if all transitions in P can be proved finitely executable. We found out that, instead of directly solving the full constraint system introduced in Section 4.3.4, in practice it is preferable to proceed by phases. Each phase[2] (procedures DisTrans, RankFun and TermImpl) attempts to remove transitions from P by different means, and returns **true** if P has become empty or it is no longer strongly connected. In the former case, we are done. In the latter, the same procedure is recursively called. If after all phases P is non-empty, we report failure to prove termination.

In the first phase (procedure DisTrans), CppInv attempts to eliminate transitions with disability arguments by generating the appropriate invariants (neither ranking functions nor termination implications are considered at this point). This is achieved by solving the following Max-SMT formula: $\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge (\bigvee_{\tau \in T} \mathbb{D}_\tau \vee p_{\mathbb{D}}) \wedge [\neg p_{\mathbb{D}}, \omega_{\mathbb{D}}]$[3], where $p_{\mathbb{D}}$ is a

[2] These phases have a time limit in our implementation although this is not made explicit in the pseudo-code shown below.

[3] Constraints that avoid redundancy are not included for simplicity.

Algorithm 4.9 Proc. ProvedSCCTerm

Input: Set of locations L, set of transitions T, and set of transitions P
Output: **true** if all transitions in P can be proved finitely executable

```
1: if DisTrans(L, T, P) or RankFun(L, T, P) or TermImpl(L, T, P) then
2:   if P = ∅ then
3:     return true
4:   end if
5:   C ← ComputeSCCs(P)
6:   for all (C' ∈ C) do
7:     T' ← Transitions(C')
8:     if T' ≠ ∅ and not ProvedSCCTerm(L, T, T') then
9:       return false
10:    end if
11:  end for
12:  return true
13: else
14:  return false
15: end if
```

propositional variable meaning that no transition can be disabled, and $\omega_{\mathbb{D}}$ is the corresponding weight. Transitions that are detected to be disabled (by means of a call to an SMT solver) are removed both from the original and the termination transition system. Invariants are used to strengthen the transition relations as described in Section 4.3.2. The process is repeated while new transitions can be disabled.

In the second phase (procedure RankFun), the system eliminates transitions by using ranking functions as arguments (termination implications are not considered at this point). If the computed function R satisfies the non-increase property, then each of the transitions τ in the termination transition system is examined and either removed if R is a ranking function for τ, or split otherwise, as described in Section 4.3.4.

The third and final phase (procedure TermImpl, not detailed here) is very similar to the previous one, with the difference that termination implications are also included.

As regards the constraints, we restrain ourselves to invariants and ranking functions with *integer* coefficients, since this allows us to exploit efficient non-linear solving techniques [Borralleras et al., 2012; Larraz et al., 2014b]. Moreover, in order to perform integer reasoning, we use the encoding technique explained in Section 3.2.2 based on Farkas' Lemma. Then, CppInv

Algorithm 4.10 Proc. DisTrans

Input: Set of locations L, set of transitions T, and set of transitions P
Output: **true** if P has become empty or it's no longer strongly connected

1: $cont \leftarrow$ **true**
2: **while** $cont$ **do**
3: $cont \leftarrow$ **false**
4: **for all** $(\tau = (\ell, \ell', \rho) \in P)$ **do**
5: **if** ρ is UNSAT **then** {τ is disabled}
6: $(T, P) \leftarrow (T - \{\tau\}, P - \{\tau\})$
7: **end if**
8: **end for**
9: **if** $P = \emptyset$ **then**
10: **return true**
11: **end if**
12: $H \leftarrow \bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge \bigvee_{\tau \in T} (\mathbb{D}_\tau \vee p_{\mathbb{D}})$
13: $S \leftarrow [\neg p_{\mathbb{D}}, \omega_{\mathbb{D}}]$
14: $(I, c) \leftarrow \mathsf{Solve}(H \wedge S)$ {I invariant map, c cost of solution}
15: **if** $c = \infty$ **then**
16: **break** {No solution to hard clauses}
17: **end if**
18: **for all** $(\ell \in L, (\ell, \ell', \rho) \in \mathcal{T})$ **do** {Strengthen relation with invariant}
19: $\rho \leftarrow \rho \wedge I(\ell)$
20: **end for**
21: $cont \leftarrow c = 0$
22: **end while**
23: **return not** $\mathsf{IsStronglyConnected}(P)$

uses BARCELOGIC for solving the generated constraints (see Section 2.1.3).

4.3.7 Experimental evaluation

The method presented in this section has been implemented in the tool CPPINV[4]. Here we show experiments that evaluate the performance of CPPINV on a wide set of examples, which have been taken from the online programming learning environment Jutge.org [Petit et al., 2012] (see `www.jutge.org`), and from benchmark suites in [Brockschmidt et al., 2013] and in research.microsoft.com/en-us/projects/t2/. We provide here a comparison with the new version of T2, which according to the results given

[4] CPPINV, together with all benchmarks used in the experimental evaluation, is available at `www.cs.upc.edu/~albert/cppinv-Term.tar.gz`.

Algorithm 4.11 Proc. RankFun

Input: Set of locations L, set of transitions T, and set of transitions P
Output: **true** if P has become empty or it's no longer strongly connected

1: **while** (**true**) **do**
2: $H \leftarrow \bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} \mathbb{C}_\tau \wedge \bigvee_{\tau \in P} \big((\mathbb{B}_\tau \vee p_{\mathbb{B}}) \wedge (\mathbb{S}_\tau \vee p_{\mathbb{S}})\big) \wedge \bigwedge_{\tau \in P} (\mathbb{N}_\tau \vee p_{\mathbb{N}})$
3: $S \leftarrow [\neg p_{\mathbb{B}}, \omega_{\mathbb{B}}] \wedge [\neg p_{\mathbb{S}}, \omega_{\mathbb{S}}] \wedge [\neg p_{\mathbb{N}}, \omega_{\mathbb{N}}]$
4: $(I, R, c) = \mathsf{Solve}(H \wedge S)$
5: **if** $c = \infty$ **then**
6: **return false** {No solution to hard clauses}
7: **end if**
8: **for all** $(\ell \in L, (\ell, \ell', \rho) \in \mathcal{T})$ **do** {Strengthen relation with invariant}
9: $\rho \leftarrow \rho \wedge I(\ell)$
10: **end for**
11: **for all** $(\tau = (\ell, \ell', \rho) \in P)$ **do**
12: **if** ρ is UNSAT **then** {τ is disabled}
13: $(T, P) \leftarrow (T - \{\tau\}, P - \{\tau\})$
14: **end if**
15: **end for**
16: **if** $\mathsf{NonIncrease}(R)$ **then**
17: **for all** $(\tau \in P)$ **do**
18: **if** $\mathsf{Bounded}(\tau, R)$ **and** $\mathsf{StrictDecrease}(\tau, R)$ **then**
19: $P \leftarrow P - \{\tau\}$
20: **else**
21: $\mathsf{Split}(\tau, R, P)$ {Splits τ}
22: **end if**
23: **end for**
24: **end if**
25: **if** $P = \emptyset$ **or not** $\mathsf{IsStronglyConnected}(P)$ **then**
26: **return true**
27: **end if**
28: **end while**

in [Brockschmidt et al., 2013] is performing much better when proving termination than most of the existing tools, including TERMINATOR [Cook et al., 2006], APROVE [Otto et al., 2010] or ARMC [Podelski and Rybalchenko, 2007], among others. We have tried CPROVER [Tsitovich et al., 2011] and LOOPFROG [Kroening et al., 2009] as well, but the results were not good on these sets of benchmarks. All experiments were performed on an Intel Core i7 with 3.40GHz clock speed and 16 GB of RAM.

	noMS	MS	MS+QR	MS+QR+TI	T2
Set1	210	218	226	236	245
Set2	242	249	259	272	275(+3)

Table 4.12. Results with benchmarks from T2

The first two considered sets of benchmarks are those provided by the T2 developers. Following the experiments in [Brockschmidt et al., 2013], we have set a 300 secs. timeout. In order to show the impact of the different techniques described so far in this chapter, Table 4.12 presents the number of programs proved terminating while adding the different ingredients:

- *(noMS)* implements the generation of invariants and ranking functions using a translation to SMT(NA), but without using Max-SMT, i.e. with all constraints *hard*. The fact that this plain version can already prove many instances hints on the goodness of our underlying algorithm and the impact of using our NA-solver in this application.

- *(MS)* implements the generation of invariants and ranking functions using Max-SMT(NA), where the constraints imposed by the ranking function are added as *soft*.

- *(MS+QR)* adds to the previous case the possibility to use quasi-ranking functions.

- *(MS+QR+TI)* adds to the previous case the possibility to infer termination implications.

Note that every added improvement allows us to prove some more instances, while none is lost due to the additional complexity of the constraints generated.

Moreover, by looking into the results in more detail, we have observed that our tool and T2 complement each other to some extent: in Set1 CppInv can prove 5 instances which cannot be proved by T2, while we cannot prove 14 which can be handled by T2; similarly, in Set2 CppInv can prove 8 programs which cannot be proved by T2, while we cannot prove 11 that can be handled by T2 (the +3 in Table 4.12 refers to 3 instances which include constructs not supported by CppInv). The average time in YES answers of T2 is 2.9 secs and of CppInv is 12.8 secs.

	CppInv	T2		CppInv	T2
P11655	324	328	P40685	324	329
P12603	143	140	P45965	780	793
P12828	707	710	P70756	243	235
P16415	81	81	P81966	2663	926
P24674	171	168	P82660	174	177
P33412	478	371	P84219	325	243

Table 4.13. Results with benchmarks from Jutge.org.

In Table 4.13, we show the comparison of CppInv (with all described techniques) and T2 on our benchmarks from the programming learning environment Jutge.org, which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite consists of thousands of solutions written by students to 12 different programming problems. These programs can be considered challenging examples since most often they are not the most elegant solution but one with many more conditional statements than necessary. In this case, due to the size of the benchmark suite, for the execution of both tools we have set a 120 secs. timeout. The average time in YES answers of T2 is 1.7 secs. and of CppInv is 1.6 secs. Note that, in order to run these benchmarks in T2, we have translated them into T2 format using our intermediate transition graph. This may be a disadvantage for T2, as it happens in the reverse way when CppInv is run on T2 benchmark set. In particular, we think that the bad performance of T2 in the problem sets P33412, P81966 and P84219 may be related to the way we handle division, which is crucial in these examples.

4.4 Proving non-termination using Max-SMT

In this section we show how Max-SMT-based invariant generation can be exploited for proving non-termination of programs. We assume that every program is modeled with a transition system (cf. Section 2.2), and only scalar variables are declared. We will distinguish between two kind of variables, namely, the program variables $\overline{v}$, and the non-deterministic variables $\overline{u}$. The latter, described in the next section, are introduced to model non-determinism, and they represent arbitrary values.

4.4.1 Modeling of non-determinism

When proving termination, non-determinism might have the effect of preventing fulfillment of a particular property for all the reachable states. However, if a termination proof like the described in Subsection 4.3.1 is found, without a special concern about non-determinism, then we can be sure that the proof is correct. In contrast, when proving non-termination it is crucial to reason about non-determinism to ensure correctness of a proof.

For the sake of presentation, we assume that the non-determinism of programs can come only from non-deterministic assignments of the form $i :=$ `nondet`(), where $i \in \overline{v}$ is a program variable. Note that, however, this assumption still allows one to encode other kinds of non-determinism. For instance, any non-deterministic branching of the form **if**$(*)\{\}$ **else**$\{\}$ can be cast into this framework by introducing a new program variable $k \in \overline{v}$ and rewriting into the form $k :=$ `nondet`(); **if**$(k \geq 0)\{\}$ **else**$\{\}$.

The transition relation of a non-deterministic assignment of the form i := `nondet`(), where $i \in \overline{v}$, is represented by the formula $i' = u_1$, where $u_1 \in \overline{u}$ is a fresh non-deterministic variable. Note that u_1 is not a program variable, i.e., $u_1 \notin \overline{v}$, and is added only to model the non-deterministic assignment. Thus, without loss of generality on the kind of non-deterministic programs we can model, we will assume that every non-deterministic variable appears in at most one transition relation. A transition that includes a non-deterministic variable in its transition relation is called *non-deterministic* (abbreviated as `nondet`).

Given a transition relation $\rho = \rho(\overline{v}, \overline{u}, \overline{v}')$, we will use $\rho(\overline{v})$ to denote the *conditional part of* ρ, i.e., the conjunction of linear inequalities in ρ containing only variables in $\overline{v}$. For a transition system modeling real programs, the

following conditions are true:

$$\text{For } \tau = (\ell, \ell', \rho) \in \mathcal{T} : \forall \overline{v}, \overline{u}\, \exists \overline{v}'.\ \rho(\overline{v}) \rightarrow \rho(\overline{v}, \overline{u}, \overline{v}'). \tag{4.1}$$

$$\text{For } \ell \in \mathcal{L} : \bigvee_{(\ell,\ell',\rho)} \rho(\overline{v}) \triangleq \texttt{true}. \tag{4.2}$$

$$\text{For } \tau_1 = (\ell, \ell_1, \rho_1), \tau_2 = (\ell, \ell_2, \rho_2) \in \mathcal{T}, \tau_1 \neq \tau_2 : \rho_1(\overline{v}) \wedge \rho_2(\overline{v}) \triangleq \texttt{false}. \tag{4.3}$$

Condition (4.1) guarantees that next values for the program variables always exist if the conditional part of the transition holds. Condition (4.2) expresses that, for any location, at least one of the outgoing transitions from that location can always be executed. Finally, condition (4.3) says that any two different transitions from the same location are mutually exclusive, i.e., conditional branching is always deterministic.

Example 4.14. Let us consider the program shown in Fig. 4.15. Note how the two non-deterministic assignments have been replaced in the CFG by assignments to fresh non-deterministic variables u_1 and u_2. Condition (4.2) is trivially satisfied for ℓ_0 and ℓ_2, since the conditional part of their outgoing transition relations is empty. Regarding ℓ_1, clearly the formula $x \geq y \vee x < y$ is a tautology. Condition (4.3) is also easy to check: the conditional parts of $\rho_{\tau_2}, \rho_{\tau_3}$ and ρ_{τ_4} are pairwise unsatisfiable. Finally, condition (4.1) trivially holds since the primed parts of the transition relations consist of equalities whose left-hand side is always a different variable.

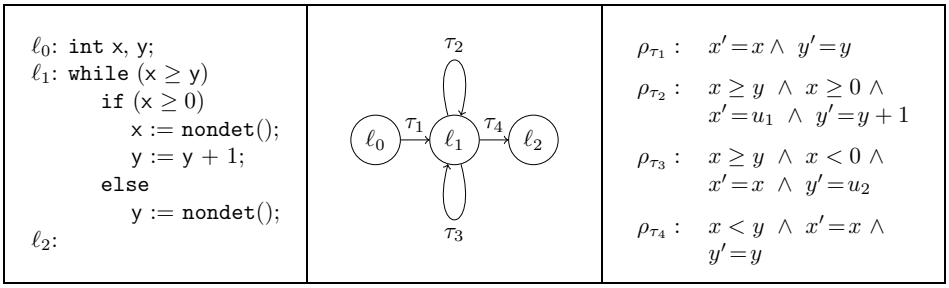


Figure 4.15. Program involving non-deterministic assignments, together with its CFG

4.4.2 Overview of the Max-SMT approach

Our method analyses each *Strongly Connected SubGraph (SCSG)* of a program's control flow graph and, by means of Max-SMT solving, tries to find

a formula at every node of the SCSG that satisfies certain properties. First, the formula has to be a *quasi-invariant*, i.e., it must satisfy the consecution condition of inductive invariants, but not necessarily the initiation condition. Hence, if it holds at the node during execution once, then it continues to hold from then onwards. Second, the formula has to be *edge-closing*, meaning that it forbids taking any of the outgoing edges from that node that exit the SCSG. Now, once we have computed an edge-closing quasi-invariant for every node of the SCSG, if a state is reached that satisfies one of them, then program execution will remain within the SCSG from then onwards. The existence of such a state is tested with an off-the-shelf reachability checker. If it succeeds, we have proved non-termination of the original program, and the edge-closing quasi-invariants of the SCSG and the trace given by the reachability checker form the witness of non-termination.

Our approach differs from previous methods in two major ways. First, edge-closing quasi-invariants are more generic properties than the witnesses for non-termination produced by other provers, and thus are likely to carry more information and be more useful in bug finding. Second, our non-termination witnesses include SCSGs, which are larger structures than, e.g., *lassos*. Note that the number of SCSGs present in any CFG is finite, while the number of lassos is infinite. Because of these differences, our method is more likely to converge. Moreover, lasso-based methods can only handle periodic non-termination, while our approach can deal with aperiodic non-termination too.

Our technique is based on constraint solving for invariant generation (see Section 3.2) and is goal-directed. Before discussing it formally, we describe it with a simple example. Consider the program in Fig. 4.16(a). The CFG for this program is shown in Fig. 4.16(b). The edges of the CFG represent the transitions between the locations. For every transition τ, we denote the formula of its transition relation by $\rho_\tau(i, j, i', j')$. The unprimed variables represent the values of the variables before the transition, and the primed ones represent the values after the transition. By $\rho_\tau(i, j)$ we denote the *conditional part of* τ, which only involves the pre-variables. Fig. 4.16(c) shows all non-trivial (i.e. with at least one edge) SCSGs present in the CFG. For every SCSG, the dashed edges are those that exit the SCSG and hence are not part of it. Note that SCSG-1 is a maximal strongly connected subgraph, and thus is a strongly connected component of the CFG. Notice

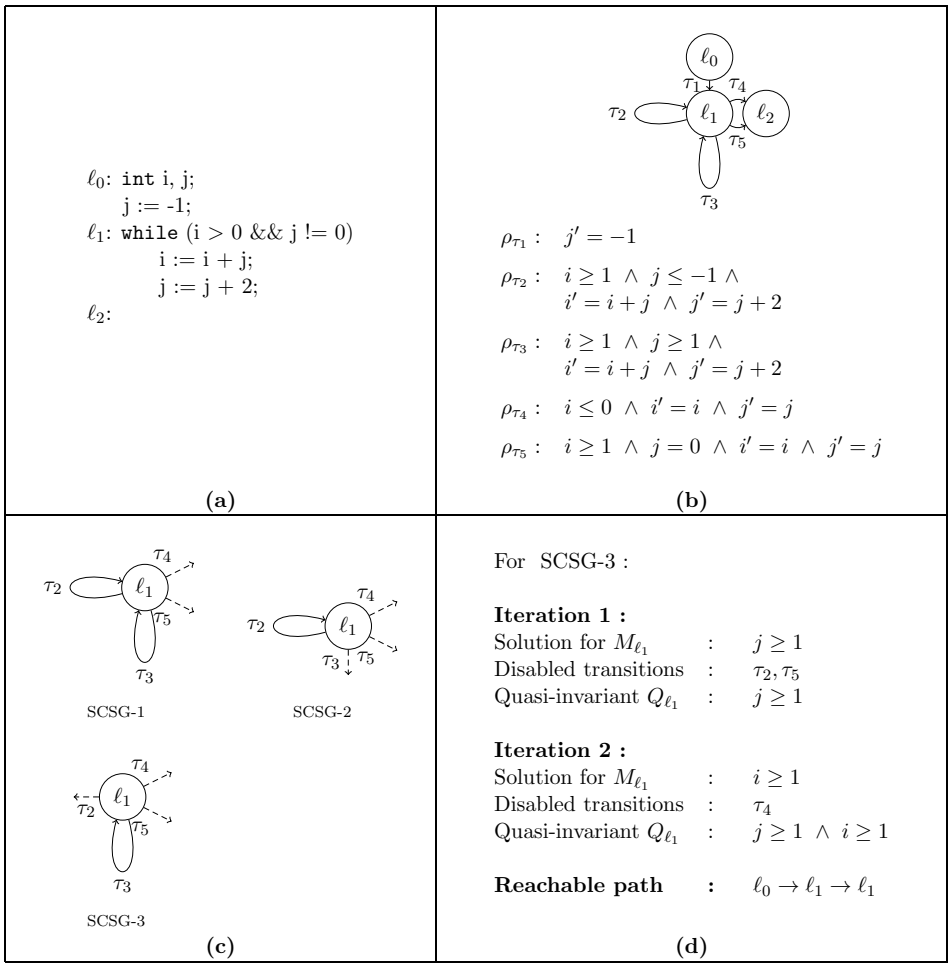


Figure 4.16. Example program (a) together with its corresponding CFG (b), non-trivial SCSGs (c) and non-termination analysis (d)

also that τ_3 is an additional exit edge for SCSG-2, and similarly τ_2 is an exit edge for SCSG-3. The non-termination of this example comes from SCSG-3.

Our approach considers every SCSG of the graph one by one. In every iteration of our method, we try to find a formula at every node of the SCSG under consideration. This formula is originally represented as a template with unknown coefficients. We then form a system of constraints involving the template coefficients in the Max-SMT framework. In a Max-SMT problem, some of the constraints are *hard*, meaning that any solution to the system of constraints must satisfy them, and others are *soft*, which may or may not be satisfied. Soft constraints carry a weight, and the goal of the Max-SMT solver is to find a solution for the hard constraints such that

the sum of the weights for the soft constraints violated by the solution is minimized. In our method, essentially the hard constraints encode that the formula should obey the consecution condition, and every soft constraint encodes that the formula will disable an exit edge. A solution to this system of constraints assigns values to template coefficients, thus giving us the required formula at every node.

Consider the analysis of SCSG-3 (refer to Fig. 4.16(d)). Note that there is a single node ℓ_1 and a single transition τ_3 in SCSG-3. We denote by $E = \{\tau_2, \tau_4, \tau_5\}$ the set of exit edges for SCSG-3. By $Q_{\ell_1}(i, j)$ we denote the quasi-invariant at node ℓ_1. Initially $Q_{\ell_1}(i, j) \triangleq \texttt{true}$. In the first iteration, for node ℓ_1 we assign a template $M_{\ell_1}(i, j) : a.i + b.j \leq c$.

We then form the Max-SMT problem consisting of the following system of hard and soft constraints:

(**Consecution**) $\forall\, i, j, i', j'.\ M_{\ell_1}(i, j)\ \wedge\ Q_{\ell_1}(i, j)\ \wedge\ \rho_{\tau_3}(i, j, i', j') \rightarrow M_{\ell_1}(i', j')$

(**Edge-Closing**) For all $\tau \in E$: $\forall\, i, j.\ M_{\ell_1}(i, j)\ \wedge\ Q_{\ell_1}(i, j) \rightarrow \neg\rho_{\tau}(i, j)$

The consecution constraint is hard, while the edge-closing constraints are soft (with weight, say, 1). The edge-closing constraint for $\tau \in E$ encodes that, from any state satisfying $M_{\ell_1}(i, j) \wedge Q_{\ell_1}(i, j)$, the transition τ is disabled and cannot be executed.

In the first iteration, a solution for M_{ℓ_1} gives us the formula $j \geq 1$. This formula satisfies the edge-closing constraints for τ_2 and τ_5. We conjoin this formula to Q_{ℓ_1}, updating it to $Q_{\ell_1}(i, j) \triangleq j \geq 1$. We also update $E = \{\tau_4\}$ by removing τ_2 and τ_5, as these edges are now disabled.

In the second iteration, we again consider the same template $M_{\ell_1}(i, j)$ and try to solve the Max-SMT problem above with updated $Q_{\ell_1}(i, j)$ and E. This time we get a solution that gives us the formula $i \geq 1$, which satisfies the edge-closing constraint for τ_4. We again update $Q_{\ell_1}(i, j) \triangleq j \geq 1 \wedge i \geq 1$ by conjoining the new formula. We update $E = \emptyset$ by removing the disabled edge τ_4. Now all exit edges have been disabled, and thus the quasi-invariant $Q_{\ell_1}(i, j)$ is edge-closing.

In the final step of our method, we use a reachability checker to determine if any state satisfying $Q_{\ell_1}(i, j)$ at location ℓ_1 is reachable. This test succeeds, and a path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_1$ is obtained. Notice that the path goes through the loop once before we reach the required state. At this point, we have proved non-termination of the original program.

4.4.3 Quasi-invariants and non-termination

Here we will introduce the core concept of this work, that of a *quasi-invariant*: a property such that, if it is satisfied at a location during execution once, then it continues to hold at that location from then onwards. The importance of this notion resides in the fact that it is a key ingredient in our witnesses of non-termination: if each location of an SCSG can be mapped to a quasi-invariant that is *edge-closing*, i.e., that forbids executing any of the outgoing transitions that leave the SCSG, and the SCSG can be reached at a state satisfying the corresponding quasi-invariant, then the program is non-terminating (if `nondet` transitions are present, additional properties are required, as will be seen below). A constructive proof of this claim is given at the end of this section.

First of all, let us define basic notation. For a strongly connected subgraph (SCSG) $\mathcal{C}$ of a program's CFG, we denote by $\mathcal{L}^{\mathcal{C}}$ the set of locations of $\mathcal{C}$, and by $\mathcal{T}^{\mathcal{C}}$ the set of edges of $\mathcal{C}$. We define $\mathcal{E}^{\mathcal{C}} \stackrel{def}{=} \{\tau = (\ell, \ell', \rho) \mid \ell \in \mathcal{L}^{\mathcal{C}}, \tau \notin \mathcal{T}^{\mathcal{C}}\}$ to be the set of exit edges of $\mathcal{C}$.

Consider a map $\mathcal{Q}$ that assigns a formula $Q_\ell(\overline{v})$ to each of the locations $\ell \in \mathcal{L}^{\mathcal{C}}$. Consider also a map $\mathcal{U}$ that assigns a formula $U_\tau(\overline{v}, \overline{u})$ to each transition $\tau \in \mathcal{T}^{\mathcal{C}}$, which represents the *restriction* that the non-deterministic variables must obey.[5] The map $\mathcal{Q}$ is a *quasi-invariant map* on $\mathcal{C}$ with restriction $\mathcal{U}$ if:

(Consecution)

$$\text{For } \tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}} : \forall \overline{v}, \overline{u}, \overline{v}'.\ Q_\ell(\overline{v}) \wedge \rho(\overline{v}, \overline{u}, \overline{v}') \wedge U_\tau(\overline{v}, \overline{u}) \rightarrow Q_{\ell'}(\overline{v}') \quad (4.4)$$

Condition (4.4) says that, whenever a state at $\ell \in \mathcal{L}^{\mathcal{C}}$ satisfying Q_ℓ is reached and a transition from ℓ to ℓ' can be executed, then the resulting state satisfies $Q_{\ell'}$. This condition corresponds to the consecution condition for inductive invariants (see Section 3.2). Since inductive invariants are additionally required to satisfy initiation conditions, we refer to properties satisfying condition (4.4) as quasi-invariants, hence the name for $\mathcal{Q}$.

Example 4.17. In order to explain the roles of $\mathcal{Q}$ and $\mathcal{U}$, consider the program in Fig. 4.15. It is easy to see that if $x \geq y$ were a quasi-invariant at ℓ_1, the program would be non-terminating (provided ℓ_1 is reachable with a state such that $x \geq y$). However, due to the non-deterministic assignments,

[5] For the sake of presentation, we assume that U_τ is defined for all transitions, whether they are deterministic or not. In the former case, by convention U_τ is `true`.

the property is not a quasi-invariant. On the other hand, if we add the restrictions $U_{\tau_2} := u_1 \geq x + 1$ and $U_{\tau_3} := u_2 \leq y$, which constrain the non-deterministic choices in the assignments, the quasi-invariant holds and non-termination is proved.

Additionally, our method also needs that $\mathcal{Q}$ and $\mathcal{U}$ are *reachable* and *unblocking*:

$$\textbf{(Reachability)}\ \exists\ \ell \in \mathcal{L}^{\mathcal{C}}.\ \exists\ \sigma\ \textit{s.t.}\ (\ell, \sigma) \text{ is reachable and } \sigma \models Q_\ell(\overline{v}) \tag{4.5}$$

$$\textbf{(Unblocking)}\ \text{For } \tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}} : \forall \overline{v} \exists \overline{u}.\ Q_\ell(\overline{v}) \wedge \rho(\overline{v}) \rightarrow U_\tau(\overline{v}, \overline{u}) \tag{4.6}$$

Condition (4.5) says that there exists a computation reaching a state (ℓ, σ) such that σ satisfies the quasi-invariant at location ℓ.

As for condition (4.6), consider a state σ at some $\ell \in \mathcal{L}^{\mathcal{C}}$ satisfying $Q_\ell(\overline{v})$. This condition says that, for any transition $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$ from ℓ, if σ satisfies the conditional part $\rho(\overline{v})$, then we can always make a choice for the non-deterministic assignment that obeys the restriction $U_\tau(\overline{v}, \overline{u})$.

The last property we require from quasi-invariants is that they are edge-closing. Formally, the quasi-invariant map $\mathcal{Q}$ on $\mathcal{C}$ is *edge-closing* if it satisfies all of the following constraints:

$$\textbf{(Edge-Closing)}\ \text{For } \tau = (\ell, \ell', \rho) \in \mathcal{E}^{\mathcal{C}} : \forall \overline{v}.\ Q_\ell(\overline{v}) \rightarrow \neg\rho(\overline{v}) \tag{4.7}$$

Condition (4.7) says that, from any state at $\ell \in \mathcal{L}^{\mathcal{C}}$ that satisfies $Q_\ell(\overline{v})$, all the exit transitions are disabled and cannot be executed.

The following is the main result of this section:

Theorem 4.18. *$\mathcal{Q}, \mathcal{U}$ that satisfy (4.4), (4.5), (4.6) and (4.7) for a certain SCSG $\mathcal{C}$ of a CFG P imply non-termination of P.*

In order to prove Theorem 4.18, we need the following lemma:

Lemma 4.19. *Let us assume that $\mathcal{Q}, \mathcal{U}$ satisfy (4.4), (4.6) and (4.7) for a certain SCSG $\mathcal{C}$. Let (ℓ, σ) be a state such that $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{v})$. Then there exists a state (ℓ', σ') such that $\ell' \in \mathcal{L}^{\mathcal{C}}$, $\sigma' \models Q_{\ell'}(\overline{v})$ and $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$ for a certain $\tau \in \mathcal{T}^{\mathcal{C}}$.*

Proof. By condition (4.2) (which is implicitly assumed to hold), there is a transition τ of the form (ℓ, ℓ', ρ) for a certain $\ell' \in \mathcal{L}$ such that $\sigma \models \rho(\overline{v})$. Now, by virtue of condition (4.7), since $\sigma \models Q_\ell(\overline{v})$ we have that $\tau \in \mathcal{T}^{\mathcal{C}}$. Thus, $\ell' \in \mathcal{L}^{\mathcal{C}}$. Moreover, thanks to condition (4.6) and $\sigma \models Q_\ell(\overline{v})$ and $\sigma \models \rho(\overline{v})$,

we deduce that there exist values ν for the non-deterministic variables $\overline{u}$ such that $(\sigma, \nu) \models U_\tau(\overline{v}, \overline{u})$. Further, by condition (4.1) (which is again implicitly assumed), we have that there exists a state σ' such that $(\sigma, \nu, \sigma') \models \rho(\overline{v}, \overline{u}, \overline{v}')$. All in all, by condition (4.4) and the fact that $\sigma \models Q_\ell(\overline{v})$ and $(\sigma, \nu, \sigma') \models \rho(\overline{v}, \overline{u}, \overline{v}')$ and $(\sigma, \nu) \models U_\tau(\overline{v}, \overline{u})$, we get that $\sigma' \models Q_{\ell'}(\overline{v}')$, or equivalently by renaming variables, $\sigma' \models Q_{\ell'}(\overline{v})$. So (ℓ', σ') satisfies the required properties. □

Now we are ready to prove Theorem 4.18:

Proof of Theorem 4.18. We will construct an infinite computation, which will serve as a witness of non-termination. Thanks to condition (4.5), we know that there exist a location $\ell \in \mathcal{L}^{\mathcal{C}}$ and a state σ such that (ℓ, σ) is reachable and $\sigma \models Q_\ell(\overline{v})$. As (ℓ, σ) is reachable, there is a computation π whose last state is (ℓ, σ). Now, since $\mathcal{Q}, \mathcal{U}$ satisfy (4.4), (4.6) and (4.7) for $\mathcal{C}$, and $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{v})$, we can apply Lemma 4.19 to inductively extend π to an infinite computation of P. □

4.4.4 Computing proofs of non-termination

In this section we explain how proofs of non-termination are effectively computed. As outlined in Subsection 4.4.2, first of all we exhaustively enumerate the SCSGs of the CFG. For each SCSG $\mathcal{C}$, our non-termination proving procedure Prove-NT, which will be described below, is called. By means of Max-SMT solving, this procedure iteratively computes an unblocking quasi-invariant map $\mathcal{Q}$ and a restriction map $\mathcal{U}$ for $\mathcal{C}$. If the construction is successful and eventually edge-closedness can be achieved, and moreover the quasi-invariants of $\mathcal{C}$ can be reached, then the synthesized $\mathcal{Q}, \mathcal{U}$ satisfy the properties of Theorem 4.18, and therefore the program is guaranteed not to terminate.

In a nutshell, the enumeration of SCSGs considers a strongly connected component (SCC) of the CFG at a time, and then generates all the SCSGs included in that SCC. More precisely, first of all the SCCs are considered according to a topological ordering in the CFG. Then, once an SCC $\mathcal{S}$ is fixed, the SCSGs included in $\mathcal{S}$ are heuristically enumerated starting from $\mathcal{S}$ itself (since taking a strictly smaller subgraph would imply discarding some transitions a priori arbitrarily), then simple cycles in $\mathcal{S}$ (as they are easier to deal with), and then the rest of SCSGs included in $\mathcal{S}$.

Algorithm 4.20 Proc. PROVE-NT

Input: SCSG $\mathcal{C}$, CFG P

For $\ell \in \mathcal{L}^{\mathcal{C}}$, set $Q_\ell(\overline{v}) \leftarrow \mathbf{true}$
For $\tau \in \mathcal{T}^{\mathcal{C}}$, set $U_\tau(\overline{v}, \overline{u}) \leftarrow \mathbf{true}$
$E^{\mathcal{C}} \leftarrow \mathcal{E}^{\mathcal{C}}$
while $E^{\mathcal{C}} \neq \emptyset$ **do**
  At $\ell \in \mathcal{L}^{\mathcal{C}}$, assign a template $M_\ell(\overline{v})$
  At $\tau \in \mathcal{T}^{\mathcal{C}}$, assign a template $N_\ell(\overline{v}, \overline{u})$
  Solve Max-SMT problem with hard constraints (4.8), (4.9), (4.10) and soft constraints (4.11)
  if no model for hard clauses is found **then**
    return Unknown, $\perp$
  end if
  For $\ell \in \mathcal{L}^{\mathcal{C}}$, let $\widehat{M_\ell}(\overline{v})$ = Solution for $M_\ell(\overline{v})$
  For $\tau \in \mathcal{T}^{\mathcal{C}}$, let $\widehat{N_\tau}(\overline{v}, \overline{u})$ = Solution for $N_\tau(\overline{v}, \overline{u})$
  For $\ell \in \mathcal{L}^{\mathcal{C}}$, set $Q_\ell(\overline{v}) \leftarrow Q_\ell(\overline{v}) \wedge \widehat{M_\ell}(\overline{v})$
  For $\tau \in \mathcal{T}^{\mathcal{C}}$, set $U_\tau(\overline{v}, \overline{u}) \leftarrow U_\tau(\overline{v}, \overline{u}) \wedge \widehat{N_\tau}(\overline{v}, \overline{u})$
  Remove from $E^{\mathcal{C}}$ disabled edges
end while
for all $\ell \in \mathcal{L}^{\mathcal{C}}$ **do**
  if Reachable (ℓ, σ) in P s.t. $\sigma \models Q_\ell(\overline{v})$ **then**
    let π = reachable path to (ℓ, σ)
    return Non-Terminating, $(\mathcal{Q}, \mathcal{U}, \pi)$
  end if
end for
return Unknown, $\perp$

Then, once the SCSG $\mathcal{C}$ is fixed, our non-termination proving procedure PROVE-NT (Algorithm 4.20) is called. The procedure takes as input an SCSG $\mathcal{C}$ of the program's CFG, and the CFG itself. For every location $\ell \in \mathcal{L}^{\mathcal{C}}$, we initially assign a quasi-invariant $Q_\ell(\overline{v}) \triangleq \texttt{true}$. Similarly, for every transition $\tau \in \mathcal{T}^{\mathcal{C}}$, we initially assign a restriction $U_\tau(\overline{v}, \overline{u}) \triangleq \texttt{true}$. The set $E^{\mathcal{C}}$ keeps track of the exit edges of $\mathcal{C}$ that have not been discarded yet, and hence at the beginning we have $E^{\mathcal{C}} = \mathcal{E}^{\mathcal{C}}$. Then we iterate in a loop in order to strengthen the quasi-invariants and restrictions till $E^{\mathcal{C}} = \emptyset$, that is, all the exit edges of $\mathcal{C}$ are disabled.

In every iteration we assign a template $M_\ell(\overline{v}) \equiv m_{\ell,0} + \sum_{v \in \overline{v}} m_{\ell,v} \cdot v \leq 0$ to each $\ell \in \mathcal{L}^{\mathcal{C}}$. We also assign a template $N_\tau(\overline{v}, \overline{u}) \equiv n_{\tau,0} + \sum_{v \in \overline{v}} n_{\tau,v} \cdot v + \sum_{u \in \overline{u}} n_{\tau,u} \cdot u \leq 0$ to each $\tau \in \mathcal{T}^{\mathcal{C}}$.[6] Then we form the Max-SMT problem

[6] Actually templates $N_\tau(\overline{v}, \overline{u})$ are only introduced for `nondet` transitions. To simplify

with the following constraints:[7]

- For $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$:
$$\forall \overline{v}, \overline{u}, \overline{v}'.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \rho(\overline{v}, \overline{u}, \overline{v}') \wedge U_\tau(\overline{v}, \overline{u}) \wedge N_\tau(\overline{v}, \overline{u}) \rightarrow M_{\ell'}(\overline{v}') \tag{4.8}$$
- For $\ell \in \mathcal{L}^{\mathcal{C}}$: $\exists \overline{v}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \bigvee_{\tau=(\ell,\ell',\rho)\in\mathcal{T}^{\mathcal{C}}} \rho(\overline{v})$ (4.9)
- For $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$:
$$\forall \overline{v} \exists \overline{u}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \rho(\overline{v}) \rightarrow U_\tau(\overline{v}, \overline{u}) \wedge N_\tau(\overline{v}, \overline{u}) \tag{4.10}$$
- For $\tau = (\ell, \ell', \rho) \in \mathcal{E}^{\mathcal{C}} : \forall \overline{v}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \rightarrow \neg\rho(\overline{v})$ (4.11)

The constraints (4.8), (4.9) and (4.10) are hard, while the constraints (4.11) are soft.

The Max-SMT solver finds a solution $\widehat{M_\ell}(\overline{v})$ for every $M_\ell(\overline{v})$ for $\ell \in \mathcal{L}^{\mathcal{C}}$ and a solution $\widehat{N_\tau}(\overline{v}, \overline{u})$ for every $N_\ell(\overline{v}, \overline{u})$ for $\tau \in \mathcal{T}^{\mathcal{C}}$. The solution satisfies the hard constraints and as many soft constraints as possible. In other words, it is the best solution for hard constraints that disables the maximum number of transitions. We then update $Q_\ell(\overline{v})$ for every $\ell \in \mathcal{L}^{\mathcal{C}}$ by strengthening it with $\widehat{M_\ell}(\overline{v})$, and update $U_\tau(\overline{v}, \overline{u})$ for every $\tau \in \mathcal{T}^{\mathcal{C}}$ by strengthening it with $\widehat{N_\tau}(\overline{v}, \overline{u})$. We then remove all the disabled transitions from $E^{\mathcal{C}}$ and continue the iterations of the loop with updated $\mathcal{Q}$, $\mathcal{U}$ and $E^{\mathcal{C}}$. Note that, even if none of the exit edges is disabled in an iteration (i.e. no soft constraint is met), the quasi-invariants found in that iteration may be helpful for disabling exit edges later.

When all exit transitions are disabled, we exit the loop with the unblocking edge-closing quasi-invariant map $\mathcal{Q}$ and the restriction map $\mathcal{U}$.

Finally, we check whether there exists a reachable state (ℓ, σ) such that $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{v})$ with an off-the-shelf reachability checker. If this test succeeds, we report non-termination along with $\mathcal{Q}$,$\mathcal{U}$ and the path π reaching (ℓ, σ) as a witness of non-termination.

The next theorem formally states that PROVE-NT proves non-termination:

Theorem 4.21. If procedure PROVE-NT terminates on input SCSG $\mathcal{C}$ and CFG P with Non-Terminating, $(\mathcal{Q},\ \mathcal{U},\ \pi)$, then program P is non-

the presentation, we assume that for other transitions, $N_\tau(\overline{v}, \overline{u})$ is `true`.

[7]For clarity, leftmost existential quantifiers over the unknowns of the templates are implicit.

terminating, and $(\mathcal{Q}, \mathcal{U}, \pi)$ allow building an infinite computation of P.

Proof. Let us prove that, if PROVE-NT terminates with Non-Terminating, then the conditions of Theorem 4.18, i.e., conditions (4.4), (4.5), (4.6) and (4.7) are met.

First of all, let us prove by induction on the number of iterations of the **while** loop that conditions (4.4) and (4.6) are satisfied, and also that for $\tau = (\ell, \ell', \rho) \in \mathcal{E}^{\mathcal{C}} - E^{\mathcal{C}}$,

$$\forall \overline{v}.\ Q_\ell(\overline{v}) \to \neg\rho(\overline{v}).$$

Before the loop is executed, for all locations $\ell \in \mathcal{L}^{\mathcal{C}}$ we have that $Q_\ell(\overline{v}) \triangleq \texttt{true}$ and for all $\tau \in \mathcal{T}^{\mathcal{C}}$ we have that $U_\tau(\overline{v}, \overline{u}) \triangleq \texttt{true}$. Conditions (4.4) and (4.6) are trivially met. The other remaining condition holds since initially $E^{\mathcal{C}} = \mathcal{E}^{\mathcal{C}}$.

Now let us see that each iteration of the loop preserves the three conditions. Regarding (4.4), by induction hypothesis we have that for $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{v}, \overline{u}, \overline{v}'.\ Q_\ell(\overline{v}) \wedge \rho(\overline{v}, \overline{u}, \overline{v}') \wedge U_\tau(\overline{v}, \overline{u}) \to Q_{\ell'}(\overline{v}').$$

Moreover, the solution computed by the Max-SMT solver satisfies constraint (4.8), i.e., has the property that for $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{v}, \overline{u}, \overline{v}'.\ Q_\ell(\overline{v}) \wedge \widehat{M_\ell}(\overline{v}) \wedge \rho(\overline{v}, \overline{u}, \overline{v}') \wedge U_\tau(\overline{v}, \overline{u}) \wedge \widehat{N_\tau}(\overline{v}, \overline{u}) \to \widehat{M_{\ell'}}(\overline{v}').$$

Altogether, we have that for $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{v}, \overline{u}, \overline{v}'.(Q_\ell(\overline{v}) \wedge \widehat{M_\ell}(\overline{v})) \wedge \rho(\overline{v}, \overline{u}, \overline{v}') \wedge (U_\tau(\overline{v}, \overline{u}) \wedge \widehat{N_\tau}(\overline{v}, \overline{u})) \to (Q_{\ell'}(\overline{v}') \wedge \widehat{M_{\ell'}}(\overline{v}')).$$

Hence condition (4.4) is preserved.

As for condition (4.6), the solution computed by the Max-SMT solver satisfies constraint (4.10), i.e., has the property that for $\tau = (\ell, \ell', \rho) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{v} \exists \overline{u}.\ (Q_\ell(\overline{v}) \wedge \widehat{M_\ell}(\overline{v})) \wedge \rho(\overline{v}) \to (U_\tau(\overline{v}, \overline{u}) \wedge \widehat{N_\tau}(\overline{v}, \overline{u})).$$

Thus, condition (4.6) is preserved.

Regarding the last property, note that the transitions $\tau = (\ell, \ell', \rho) \in E^{\mathcal{C}}$

that satisfy the soft constraints (4.11), i.e., such that

$$\forall \overline{v}.\ (Q_\ell(\overline{v}) \wedge \widehat{M_\ell}(\overline{v})) \rightarrow \neg\rho(\overline{v})$$

are those removed from $E^{\mathcal{C}}$. Therefore, this preserves the property that for $\tau = (\ell, \ell', \rho) \in \mathcal{E}^{\mathcal{C}} - E^{\mathcal{C}}$,

$$\forall \overline{v}.\ Q_\ell(\overline{v}) \rightarrow \neg\rho(\overline{v}).$$

Now, if the **while** loop terminates, it must be the case that $E^{\mathcal{C}} = \emptyset$. Thus, on exit of the loop, condition (4.7) is fulfilled.

Finally, if Non-Terminating is returned, then there is a location $\ell \in \mathcal{L}^{\mathcal{C}}$ and a state satisfying $\sigma \models Q_\ell(\overline{v})$ such that state (ℓ, σ) is reachable. That is, condition (4.5) is satisfied.

Hence, all conditions of Theorem 4.18 are fulfilled. Therefore, P does not terminate. Moreover, the proof of Theorem 4.18 gives a constructive way of building an infinite computation by means of $\mathcal{Q}$, $\mathcal{U}$ and π. □

Note that constraint (4.9):

$$\text{For } \ell \in \mathcal{L}^{\mathcal{C}} : \exists \overline{v}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \bigvee_{\tau=(\ell,\ell',\rho)\in\mathcal{T}^{\mathcal{C}}} \rho(\overline{v})$$

is not actually used in the proof of Theorem 4.21, and thus is not needed for the correctness of the approach. Its purpose is rather to help Prove-NT to avoid getting into dead-ends unnecessarily. Namely, without (4.9) it could be the case that for some location $\ell \in \mathcal{L}^{\mathcal{C}}$, we computed a quasi-invariant that forbids all transitions $\tau \in \mathcal{T}^{\mathcal{C}}$ from ℓ. Since Prove-NT only strengthens quasi-invariants and does not backtrack, if this situation were reached the procedure would probably not succeed in proving non-termination.

Now let us describe how constraints are effectively solved. First of all, constraints (4.8), (4.9), and (4.11) are universally quantified over integer variables. Following the same ideas of constraint-based linear invariant generation (see Section 3.2), these constraints are soundly transformed into an existentially quantified formula in NRA by abstracting program and non-deterministic variables and considering them as reals, and then applying Farkas' Lemma. As regards constraint (4.10), the alternation of quantifiers

in

$$\forall\overline{v}\exists\overline{u}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \rho(\overline{v}) \rightarrow U_\tau(\overline{v},\overline{u}) \wedge N_\tau(\overline{v},\overline{u})$$

is dealt with by introducing a template $P_{u,\tau}(\overline{v}) \equiv p_{u,\tau,0} + \sum_{v\in\overline{v}} p_{u,\tau,v} \cdot v$ for each $u \in \overline{u}$ and skolemizing. This yields[8] the formula

$$\forall\overline{v}.\ Q_\ell(\overline{v}) \wedge M_\ell(\overline{v}) \wedge \rho(\overline{v}) \rightarrow U_\tau(\overline{v}, P_{\overline{u},\tau}(\overline{v})) \wedge N_\tau(\overline{v}, P_{\overline{u},\tau}(\overline{v})),$$

which implies constraint (4.10), and to which the above transformation into NRA can be applied. Note that, since the Skolem function is not symbolic but an explicit linear function of the program variables, potentially one might lose solutions.

Finally, once a weighted formula in NRA containing hard and soft clauses is obtained, (some of the) existentially quantified variables are forced to take integer values, and the resulting problem is handled by a Max-SMT(NIA) solver [Nieuwenhuis and Oliveras, 2006; Larraz et al., 2014b]. In particular, the unknowns of the templates $P_{u,\tau}(\overline{v})$ introduced for skolemizing non-deterministic variables are imposed to be integers. Since program variables have integer type, this guarantees that only integer values are assigned in the non-deterministic assignments of the infinite computation that proves non-termination.

There are some other issues about our implementation of the procedure that are worth mentioning. Regarding how the weights of the soft clauses are determined, we follow a heuristic aimed at discarding "difficult" transitions in $\mathcal{E}^{\mathcal{C}}$ as soon as possible. Namely, the edge-closing constraint (4.11) of transition $\tau = (\ell, \ell', \rho) \in \mathcal{E}^{\mathcal{C}}$ is given a weight which is inversely proportional to the number of literals in $\rho(\overline{v})$. Thus, transitions with few literals in their conditional part are associated with large weights, and therefore the Max-SMT solver prefers to discard them over others. The rationale is that for these transitions there may be more states that satisfy the conditional part, and hence they may be more difficult to rule out. Altogether, it is convenient to get rid of them before quasi-invariants become too constrained.

Finally, as regards condition (4.3), our implementation can actually handle transition systems for which this condition does not hold. This may be interesting in situations where, e.g., non-determinism is present in conditional statements, and one does not want to introduce additional variables

[8] Again, existential quantifiers over template unknowns are implicit.

and locations as was done in Section 4.4.1 for presentation purposes. The only implication of overriding condition (4.3) is that, in this case, the properties that must be discarded in soft clauses of condition (4.11) are not the transitions leaving the SCSG under consideration, but rather the negation of the transitions staying within the SCSG.

4.4.5 Related work

Several systems for proving non-termination have recently been developed. One of these is, e.g., the tool TNT [Gupta et al., 2008], which proceeds in two phases. The first phase exhaustively generates candidate lassos. The second one checks each lasso for possible non-termination by seeking a *recurrent set* of states, i.e., a set of states that is visited infinitely often along the infinite path that results from unrolling the lasso. This is carried out by means of constraint solving, as in our approach. But while there is an infinite number of lassos in a program, our SCSGs can be finitely enumerated. Further, we can handle unbounded non-determinism, whereas TNT is limited to deterministic programs.

Other methods for proving non-termination that use an off-the-shelf reachability checker like our technique have also been proposed [Gulwani et al., 2008b; Chen et al., 2014]. In [Gulwani et al., 2008b], the reachability checker is used on instrumented code for inferring weakest preconditions, which give the most general characterization of the inputs under which the original program is non-terminating. While in [Gulwani et al., 2008b] non-determinism can be dealt with in a very restricted manner, the method in [Chen et al., 2014] can deal with unbounded non-determinism as we do. In the case of [Chen et al., 2014], the reachability checker is iteratively called to eliminate every terminating path through a loop by restricting the state space, and thus may diverge on many loops. Our method does not suffer from this kind of drawbacks.

Some other approaches exploit theorem-proving techniques. For example, the tool Invel [Velroyen and Rümmer, 2008] analyzes non-termination of Java programs using a combination of theorem proving and invariant generation. Invel is only applicable to deterministic programs. Another tool for proving non-termination of Java programs is AProVE [Giesl et al., 2006], which uses SMT solving as an underlying reasoning engine. The main drawback of their method is that it is required that either recurrent

sets are singletons (after program slicing) or loop conditions themselves are invariants. Our technique does not have such restrictions.

Finally, the tool TRex [Harris et al., 2011] integrates existing non-termination proving approaches within a Terminator-like [Andreas et al., 2006] iterative procedure. Unlike TRex, which is aimed at sequential code, Atig et al. [Atig et al., 2012] focus on concurrent programs: they describe a non-termination proving technique for multi-threaded programs, via a reduction to non-termination reasoning for sequential programs. Our work should complement both of these approaches, since we provide significant advantages over the underlying non-termination proving tools that were previously used.

4.4.6 Experimental evaluation

In this section we evaluate the performance of a prototype implementation of the techniques proposed here in our termination analyzer CppInv, available at `www.cs.upc.edu/~albert/cppinv-Term.tar.gz` together with all of the benchmarks. This tool admits code written in (a subset of) C++ as well as in the language of T2 [Cook et al., 2013]. The system analyses programs with integer variables, linear expressions and function calls, as well as array accesses to some extent. As a reachability checker we use CPA [Beyer and Keremoglu, 2011].

Altogether, we compare CppInv with the following tools:

- T2 [Cook et al., 2013] version CAV'13 (henceforth, T2-CAV), which implements an algorithm that tightly integrates invariant generation and termination analysis [Brockschmidt et al., 2013].
- T2 [Cook et al., 2013] version TACAS'14 (henceforth, T2-TACAS), which reduces the problem of proving non-termination to the search of an under-approximation of the program guided by a safety prover [Chen et al., 2014].
- Julia [Spoto et al., 2010], which implements a technique described by Payet and Spoto [Payet and Spoto, 2009] that reduces non-termination to constraint logic programming.
- AProVE [Giesl et al., 2006] with the Java Bytecode front-end, which uses the SMT-based non-termination analysis in [Brockschmidt et al.,

2012].

- A reimplementation of TNT [Gupta et al., 2008] by the authors of [Chen et al., 2014] that uses Z3 [de Moura and Bjørner, 2008] as an SMT back-end.

Unfortunately, because of the unavailability of some of the tools (T2-TACAS, T2-CAV, TNT) or the fact that they do not admit a common input language (Julia, AProVE), it was not possible to run all these systems on the same benchmarks on the same computer. For this reason, for each of the tables below we consider a different family of benchmarks taken from the literature and provide the results of executing our tool (on a 3.40 GHz Intel Core i7 with 16 GB of RAM) together with the data of competing systems reported in the respective publications. Note that the results of third-party systems in those publications may have some inaccuracies, due to, e.g., the conversion of benchmarks in different formats. However, in those cases the distances between the tools seem to be significant enough to draw conclusions on their relative performance.

Table 4.22 shows comparative results on benchmarks taken from [Chen et al., 2014]. In that paper, the tools T2-TACAS, AProVE, Julia and TNT are considered. The time limit is set to 60 seconds both in that work as well as in the executions of CppInv. The benchmarks are classified according to three categories: (a) all the examples in the benchmark suite known to be non-terminating previously to [Chen et al., 2014]; (b) all the examples in the benchmark suite known to be terminating previously to [Chen et al., 2014]; and (c) the rest of instances. Rows of the table correspond to non-termination provers. Columns are associated to each of these three categories of problems. Each column is split into three subcolumns reporting the number on "non-terminating" answers, the number of timed outs, and the number of other answers (which includes "terminating" and "unknown" answers), respectively. Even with the consideration that experiments were conducted on different machines, the results in columns (a) and (c) of Table 4.22 show the power of the proposed approach on these examples. As for column (b), we found out that instance `430.t2` was wrongly classified as terminating. Our witness of non-termination has been manually verified.

Table 4.23 (a), which follows a similar format to Table 4.22, compares CppInv, T2-CAV and AProVE on benchmarks from [Brockschmidt et al.,

	(a)			(b)			(c)		
	Nonterm	TO	Other	Nonterm	TO	Other	Nonterm	TO	Other
CppInv	70	6	5	1	16	237	113	35	9
T2-TACAS	51	0	30	0	45	209	82	3	72
AProVE	0	61	20	0	142	112	0	139	18
Julia	3	8	70	0	40	214	0	91	66
TNT	19	3	59	0	48	206	32	12	113

Table 4.22. Experiments with benchmarks from [Chen et al., 2014]

2013] (all with a time limit of 300 seconds). Note that, in the results reported in [Brockschmidt et al., 2013], due to a wrong abstraction in the presence of division, T2 was giving two wrong non-termination answers (namely, for the instances `rlft3.t2` and `rlft3.c.i.rlft3.pl.t2.fixed.t2`, for which the termination proofs produced by CppInv[Larraz et al., 2013a] have been checked by hand). For this reason we have discarded those two programs from the benchmark suite. In this case, the performance of our tool is slightly worse than that of T2-CAV. However, it has to be taken into account that T2-CAV was exploiting the cooperation between the termination and the non-termination provers, while we still do not apply this kind of optimizations.

In Table 4.23 (b), CppInv is compared with the results of Julia and AProVE from [Brockschmidt et al., 2012] on Java programs extracted from [Velroyen and Rümmer, 2008]. CppInv was run on C++ versions of these benchmarks, which admitted a direct translation from Java. The time limit was set to 60 seconds. Columns represent respectively the number of terminating instances (YES), non-terminating instances (NO), instances for which the construction of the proof failed before the time limit (MAYBE), and timeouts (TO). For these instances AProVE gets slightly better results than CppInv. However, it should be taken into account that four programs of this set of benchmarks include non-linear expressions, which we cannot handle. Moreover, when compared on third-party benchmarks (see Tables 4.22 and 4.23 (a)), our results are better.

Finally, Table 4.23 (c) shows the results of running our tool on programs from the online programming learning environment Jutge.org [Petit et al., 2012] (see `www.jutge.org`), which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. As a paradigmatic example in which it is easy to write wrong non-terminating

(a)

	Nonterm	TO	Other
CppInv	167	39	243
T2-CAV	172	14	263
AProVE	0	51	398

(b)

	YES	NO	MAYBE	TO
CppInv	1	44	9	1
AProVE	1	51	0	3
Julia	1	0	54	0

(c)

	YES	NO	MAYBE	TO
Binary search	2745	484	22	391

Table 4.23. Experiments with benchmarks from [Brockschmidt et al., 2013] (a), from [Velroyen and Rümmer, 2008] (b) and from Jutge.org (c)

code, we have considered the exercise **Binary Search**. The programs in this benchmark suite can be considered challenging since, having been written by students, their structure is often more complicated than necessary. In this case the time limit was 60 seconds. As can be seen from the results, for a ratio of 89% of the cases, CppInv is able to provably determine in less than one minute if the program is terminating or not.

All in all, the experimental results show that our technique, although it is general and is not tuned to particular problems, is competitive with the state of the art and performs reasonably and uniformly well on a wide variety of benchmarks.

Chapter 5

Compositional Program Analysis

To have impact on everyday software development, a verification engine needs to be able to process the millions of lines of code often encountered in mature software projects. At the same time, the analysis should be repeated every time developers commit a change, and should report feedback in the course of minutes, so that fixes can be applied promptly. Consequently, a central theme in recent research on automated program verification has been *scalability*. As a natural solution to this problem, *compositional* program analyses [Godefroid et al., 2010; Calcagno et al., 2011; Li et al., 2013] have been proposed. They analyze program parts (semi-)independently and then combine the results to obtain a whole-program proof.

For this, a compositional analysis has to predict likely intermediate assertions that allow us to break whole-program reasoning into many instances of local reasoning. This strategy makes the individual reasoning steps easier, allows distributing the analysis to a number of compute nodes [Albarghouthi et al., 2012b] and applies to all kinds of programs. For sequential programs we can guess and prove intermediate summaries of loops and/or procedures while simultaneously using the guessed summary during the analysis of the outer loop and/or procedure. In the analysis of concurrent programs a similar strategy can be employed to reason about threads without considering all possible interleavings with other threads.

The disadvantage of compositional analyses has traditionally been one

of precision: local analyses must blindly choose the intermediate assertions. While in some domains (*e.g.* heap) some heuristics have been proposed [Calcagno et al., 2011], effective strategies for guessing and/or refining useful intermediate assertions or summaries in arithmetic domains remains an open problem.

In this thesis a new method for predicting and refining intermediate arithmetic assertions for compositional reasoning about sequential programs is introduced. A key component in our approach is Max-SMT solving. Recall Max-SMT solvers can deal with hard and soft constraints, where hard constraints are mandatory, and soft constraints are those that we would like to hold, but are not required to. Here, hard constraints express what is needed for the soundness of our analysis, while soft ones favor the solutions that are more useful for our technique.

More precisely, we use Max-SMT to iteratively infer *quasi-invariants*[1], which prove the validity of a property, given that a precondition holds.

Hence, if the precondition holds, the program is proved safe. Otherwise, thanks to a novel program transformation technique we call *narrowing*[2], we exploit the failing quasi-invariants to focus on what is missing in the safety proof of the program. Then new quasi-invariants are sought, and the process is repeated until the safety proof is finally completed. Based on this, we introduce a new bottom-up program analysis procedure that infers quasi-invariants in a goal-directed manner, starting from a property that we wish to prove for the program. Our approach makes distributing analysis tasks as simple as in other bottom-up analyses, but also enjoys the precision of CEGAR-based provers.

Although in the present work compositional analysis is only applied to safety proving, we are confident about extending the method to also prove termination (so that supporting invariants do not need to follow from the direct context, as described in Section 4.3.2), and then to *existential* properties such as reachability and non-termination (thus avoiding calls to an external reachability checker as explained in Section 4.4.2).

[1]This concept was previously introduced in Section 4.4 to prove program non-termination.

[2]This narrowing is inspired by the narrowing in term rewrite systems, and is unrelated to the notion with the same name used in abstract interpretation.

5.1 Illustration of the method

In this section, we illustrate the core concepts of our approach by using some small examples. We will give the formal definition of the used methods in Section 5.2.

5.1.1 Quasi-invariants

We handle programs by considering one strongly connected component (SCC) $\mathcal{C}$ of the control-flow graph at a time, together with the sequential parts of the program leading to $\mathcal{C}$, either from initial states or other SCCs. Instead of program invariants, for each SCC we synthesize *quasi-invariants*. These are inductive properties that we choose such that they may not always hold whenever the SCC is reached, but once they hold, then they are always satisfied.

As an example, consider the program snippet in Figure 5.1, where we do not assume any knowledge about the rest of the program. To prove the assertion, we need an inductive property $\mathcal{Q}$ for the loop preceding it, such that $\mathcal{Q}$ together with the negation of the loop condition $i > 0$ implies the assertion. Using our constraint-solving based method CondSafe (cf. Section 5.2.1), we find $\mathcal{Q}_1 = x + 5 \cdot i \geq 0$. The property $\mathcal{Q}_1$ can be seen now as a *precondition* at the loop entry for the validity of the assertion.

while $i > 0$ **do**
  $x := x + 5$;
  $i := i - 1$;
done
assert$(x \geq 0)$;

Figure 5.1

5.1.2 Combining quasi-invariants

Once we have found a quasi-invariant for an SCC, we use the generated preconditions as postconditions for its preceding SCCs in the program.

while $j > 0$ **do**
  $j := j - 1$;
  $i := i + 1$;
done

Figure 5.2

As an example, assume that the loop from Figure 5.1 is directly preceded by the loop in Figure 5.2. We now use the precondition $\mathcal{Q}_1$ we obtained earlier as input to our quasi-invariant synthesis method, similarly to the assertion in Figure 5.1. Thus, we now look for an inductive property $\mathcal{Q}_2$ that, together with $\neg(j > 0)$, implies $\mathcal{Q}_1$. In this case we obtain the quasi-invariant

$\mathcal{Q}_2 = j \geq 0 \wedge x + 5 \cdot (i + j) \geq 0$ for the loop. As with $\mathcal{Q}_1$, now we can see $\mathcal{Q}_2$ as a precondition at the loop entry, and propagate $\mathcal{Q}_2$ up to the preceding SCCs in the program.

5.1.3 Recovering from failures

When we cannot prove that a found precondition always holds, we try to recover and find an alternative precondition. In this process, we make use of the results obtained so far, and *narrow* the program using our intermediate results. As an example, consider the loop in Figure 5.3.

We again apply our method CondSafe to find a quasi-invariant for this loop which, together with the loop condition, implies the assertion in the loop body. As it can only synthesize conjunctions of linear inequalities, it produces the quasi-invariant $\mathcal{Q}_3 = x > y$ for the loop. However, assume that the precondition $\mathcal{Q}_3$ could not be proven to always hold in the context of our example. In that case, we use the obtained information to narrow the program and look for another precondition.

```
while unknown() do
  assert(x ≠ y);
  x := x + 1;
  y := y + 1;
done
```

Figure 5.3

Intuitively, our program narrowing reflects that states represented by the quasi-invariant found earlier are already proven to be safe. Hence, we only need to consider states for which the negation of the quasi-invariant holds, i.e., we can add its negation as an assumption to the program. In our example, this yields the modified version of Figure 5.3 displayed in Figure 5.4. Another call to CondSafe then yields the quasi-invariant $\mathcal{Q}'_3 = x < y$ for the loop. This means that we can ensure the validity of the assertion if before the conditional statement we satisfy that $\neg(x > y) \rightarrow x < y$, or equivalently, $x \neq y$. In general, this narrowing allows us to find (some) disjunctive invariants.

```
if ¬(x>y) then
  while ¬(x>y) do
    assert(x ≠ y);
    x := x + 1;
    y := y + 1;
  done
fi
```

Figure 5.4

5.2 Proving safety

Most automated techniques for proving program safety iteratively construct *inductive program invariants* as over-approximations of the reachable state space. Starting from the known set of initial states, a process to discover more reachable states and refine the approximation is iterated, until it finally reaches a fixed point (i.e., the invariant is inductive) and is strong enough to imply program safety. However, this requires taking the whole program into account, which is sometimes infeasible or undesirable in practice.

In contrast to this, our method starts with the known unsafe states, and iteratively constructs an under-approximation of the set of safe states, with the goal of showing that all initial states are contained in that set. For this, we introduce the notion of *conditional safety*.

As in previous chapters, we model programs by means of transition systems (see Section 2.2). In what follows, we will use programs and transition systems as interchangeable terms. We assume only *scalar* variables $\overline{v}$ are declared in programs, i.e., $\mathcal{V} = \overline{v}$. Additionally, we denote in this chapter an *assertion* by (τ, φ), a pair of a transition τ and a formula φ, meaning that the formula φ must hold after transition τ. The reader is also referred to definitions about States and Executions from Section 2.2.2 to follow the rest of the chapter.

Intuitively, when proving that a program is $(\tilde{\tau}, \tilde{\varphi})$-*conditionally safe for the assertion* (τ, φ) we consider evaluations starting after a $\rightarrow_{\tilde{\tau}}$ $(\tilde{\ell}, \tilde{\sigma})$ step, where $\tilde{\sigma}$ satisfies $\tilde{\varphi}$, instead of evaluations starting at an initial state. In particular, a program that is $(\tau_0, \texttt{true})$-conditionally safe for (τ, φ) for all initial transitions τ_0 is (unconditionally) safe for (τ, φ).

Definition 5.5 (Conditional safety)**.** Let $\mathcal{P}$ be a program, $\tau, \tilde{\tau}$ transitions and $\varphi, \tilde{\varphi}$ conjunctions of linear inequalities over $\overline{v}$. The program $\mathcal{P}$ is $(\tilde{\tau}, \tilde{\varphi})$-*conditionally safe* for the assertion (τ, φ) if for any evaluation that contains $\rightarrow_{\tilde{\tau}} (\tilde{\ell}, \tilde{\sigma}) \rightarrow^*_{\mathcal{P}} (\bar{\ell}, \bar{\sigma}) \rightarrow_t (\ell, \sigma)$, we have $\tilde{\sigma} \models \tilde{\varphi}$ implies that $\sigma \models \varphi$. In that case we say that the assertion $(\tilde{\tau}, \tilde{\varphi})$ is a *precondition* for the *postcondition* (τ, φ).

Conditional safety is "transitive" in the sense that if a set of transitions $\mathcal{E} = \{\widetilde{\tau_1}, \ldots, \widetilde{\tau_m}\}$ dominates τ,[3] and for all $i = 1, \ldots, m$ we have $\mathcal{P}$ is $(\widetilde{\tau_i}, \widetilde{\varphi_i})$-

[3]We say a set of transitions $\mathcal{E}$ dominates transition τ if every path in the CFG from an initial location that contains τ must also contain some $\tilde{\tau} \in \mathcal{E}$.

conditionally safe for (τ, φ) and $\mathcal{P}$ is safe for $(\widetilde{\tau}_i, \widetilde{\varphi}_i)$, then $\mathcal{P}$ is also safe for (τ, φ). In what follows we exploit this observation to prove program safety by means of conditional safety.

A *program component* $\mathcal{C}$ of a program $\mathcal{P}$ is an SCC of the control-flow graph, and its *entry transitions* (or *entries*) are those transitions $\tau = (\ell, \ell', \rho)$ such that $\tau \notin \mathcal{C}$ but ℓ' appears in $\mathcal{C}$. We denote the set of entry transitions of $\mathcal{C}$ with $\mathcal{E}_\mathcal{C}$.

By considering each SCC as a single node, we can obtain from $\mathcal{P}$ a DAG of SCCs. In this way we can partition $\mathcal{P}$ into a set of components, corresponding to the SCCs, and the respective sets of entry transitions, which correspond to the edges in the DAG that interconnect these components. Our technique analyzes components independently, and communicates the results of these analyses to the analysis of other components along entry transitions.

Given a component $\mathcal{C}$ and an assertion (τ, φ) such that $\tau \notin \mathcal{C}$ but the source node of τ appears in $\mathcal{C}$, we call τ an *exit transition* of $\mathcal{C}$. For such exit transitions, we compute a *sufficient* condition $\psi_{\tilde{\tau}}$ for each entry transition $\tilde{\tau} \in \mathcal{E}_\mathcal{C}$ such that $\mathcal{C} \cup \{\tau\}$ is $(\tilde{\tau}, \psi_{\tilde{\tau}})$-conditionally safe for (τ, φ). Then we continue reasoning backwards following the DAG and try to prove that $\mathcal{P}$ is safe for each $(\tilde{\tau}, \psi_{\tilde{\tau}})$. If we succeed, following the argument above we will have proved $\mathcal{P}$ safe for (τ, φ).

In the following, we first discuss how to prove conditional safety of single program components in Section 5.2.1, and then present the algorithm that combines these local analyses to construct a global safety proof in Section 5.2.2.

5.2.1 Synthesizing local conditions

Here we restrict ourselves to a program component $\mathcal{C}$ and its entry transitions $\mathcal{E}_\mathcal{C}$, and assume we are given an assertion $(\tau_{\text{exit}}, \varphi)$, where $\tau_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \ell_{\text{exit}}, \rho_{\text{exit}})$ is an exit transition of $\mathcal{C}$ (i.e., $\tau_{\text{exit}} \notin \mathcal{C}$ and $\tilde{\ell}_{\text{exit}}$ appears in $\mathcal{C}$). We show next how a precondition (τ, ψ) for $(\tau_{\text{exit}}, \varphi)$ can be obtained for each $\tau \in \mathcal{E}_\mathcal{C}$. Here we only consider the case of φ being a single clause (i.e., a disjunction of literals); if φ is in CNF, each conjunct is handled separately. Now, the preconditions on the entry transitions will be determined by a *quasi-invariant*, which like a standard invariant is inductive, but not necessarily initiated in all program runs. Indeed, this initiation condition is

what we will extract as precondition and propagate backwards to preceding program components in the DAG.

Definition 5.6 (Quasi-Invariant)**.** We say a map $\mathcal{Q}$, from locations $\mathcal{L}$ to conjunctions of linear inequalities over $\overline{v}$, is a *quasi-invariant* for a program (component) $\mathcal{P}$ if for all $(\ell, \sigma) \rightarrow_{\mathcal{P}} (\ell', \sigma')$, $\sigma \models \mathcal{Q}(\ell)$ implies $\sigma' \models \mathcal{Q}(\ell')$.

Quasi-invariants are convenient tools to express conditions for safety proving, allowing reasoning in the style of "if the condition for $\mathcal{Q}$ holds, then the assertion (τ, φ) holds".

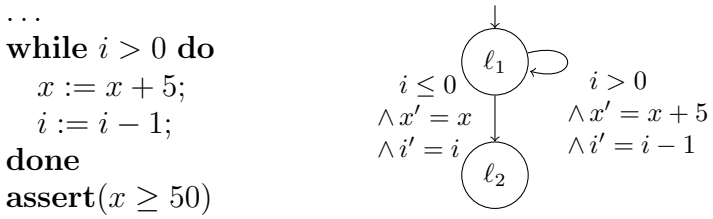


Figure 5.7. Source code of program snippet and its CFG.

Example 5.8. Consider the program snippet in Figure 5.7. A quasi-invariant supporting safety of this program part is $\mathcal{Q}_5(\ell_1) \equiv x + 5 \cdot i \geq 50$, $\mathcal{Q}_5(\ell_2) \equiv x \geq 50$. In fact, any quasi-invariant $\mathcal{Q}_m(\ell_1) \equiv x + m \cdot i \geq 50$ with $0 \leq m \leq 5$ would be a quasi-invariant that, together with the negation of the loop condition $i \leq 0$, implies $x \geq 50$.

We use a Max-SMT-based constraint-solving approach to generate quasi-invariants. Unlike in Section 4.4, to use information about the initialization of variables before a program component, we take into account the entry transitions $\mathcal{E}_\mathcal{C}$. The precondition for each entry transition is then the quasi-invariant that has been synthesized at its target location.

To find quasi-invariants, we construct a constraint system. For each location ℓ in $\mathcal{C}$ we create a template $I_{\ell,k}(\overline{v}) \equiv \wedge_{1 \leq j \leq k}\ I_{\ell,j,k}(\overline{v})$ which is a conjunction of k linear inequations[4] of the form $I_{\ell,j,k}(\overline{v}) \equiv i_{\ell,j} + \sum_{v \in \overline{v}} i_{\ell,j,v} \cdot v \leq 0$, where the $i_{\ell,j}$, $i_{\ell,j,v}$ are fresh variables. We then transform the conditions for a quasi-invariant proving safety for the assertion $(\tau_{\text{exit}}, \varphi)$ to the constraints in Figure 5.9.

[4]In our overall algorithm, k is initially 1 and increased in case of failures.

Initiation:	For $\tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}, 1 \leq j \leq k$:	$\mathbb{I}_{\tau,j,k}$	$\stackrel{def}{=}$	$\rho \rightarrow I'_{\ell',j,k}$
Consecution:	For $t = (\ell, \ell', \rho) \in \mathcal{C}$:	$\mathbb{C}_{\tau,k}$	$\stackrel{def}{=}$	$I_{\ell,k} \wedge \rho \rightarrow I'_{\ell',k}$
Safety:	For $\tau_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \ell_{\text{exit}}, \rho_{\text{exit}})$:	$\mathbb{S}_k$	$\stackrel{def}{=}$	$I_{\tilde{\ell}_{\text{exit}},k} \wedge \rho_{\text{exit}} \rightarrow \varphi'$

Figure 5.9. Constraints used in $\mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau_{\text{exit}}, \varphi))$

In the overall constraint system, we mark the **Consecution** and **Safety** constraints as hard requirements. Thus, any solution to these constraints is a quasi-invariant *implying our assertion*. However, as we mark the **Initiation** constraints as soft, the found quasi-invariants may depend on preconditions not implied by the direct context of the considered component. On the other hand, the Max-SMT solver prefers solutions that require fewer preconditions. Overall, we create the following Max-SMT formula

$$\mathbb{F}_k \stackrel{def}{=} \bigwedge_{\tau \in \mathcal{C}} \mathbb{C}_{\tau,k} \wedge \bigwedge_{\tau \in \mathcal{E}_\mathcal{C}, 1 \leq j \leq k} \left(\mathbb{I}_{\tau,j,k} \vee \neg p_{\mathbb{I}_{\tau,j,k}}\right) \wedge \mathbb{S}_k \wedge \bigwedge_{\tau \in \mathcal{E}_\mathcal{C}, 1 \leq j \leq k} [p_{\mathbb{I}_{\tau,j,k}}, \omega_\mathbb{I}],$$

where the $p_{\mathbb{I}_{\tau,j,k}}$ are propositional variables which are true if the **Initiation** condition $\mathbb{I}_{\tau,j,k}$ is satisfied, and $\omega_\mathbb{I}$ is the corresponding weight. We use $\mathbb{F}_k$ in our procedure CondSafe in Algorithm 5.10.

Algorithm 5.10 Proc. CondSafe for computing a quasi-invariant

Input: program component $\mathcal{C}$, entry transitions $\mathcal{E}_\mathcal{C}$, assertion $(\tau_{\text{exit}}, \varphi)$ s.t. τ_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause

Output: None | $\mathcal{Q}$, where $\mathcal{Q}$ maps locations in $\mathcal{C}$ to conj. of inequations

```
1: k ← 1
2: repeat
3:    construct formula F_k from C, E_C and (τ_exit, φ)
4:    σ ← Max-SMT-solver(F_k)
5:    if δ is a model then
6:       Q ← {ℓ ↦ δ(I_{ℓ,k}) | ℓ in C}
7:       return Q                    {(conditionally) safe, return solution}
8:    end if
9:    k ← k + 1
10: until k > MAX_CONJUNCTS
11: return None
```

In CondSafe, we iteratively try "larger" templates of more conjuncts of linear inequations (in our implementation, MAX_CONJUNCTS is 3) until we either give up or finally find a quasi-invariant. Note, however, that

here we are only trying to prove safety for *one* clause at a time, which reduces the number of required conjuncts as compared to dealing with a whole CNF in a single step. If the Max-SMT solver is able to find a model for $\mathbb{F}_k$, then we instantiate our invariant templates $I_{\ell,k}$ with the values found for the template variables in the model δ, obtaining a quasi-invariant $\mathcal{Q}$. When we obtain a result, for every entry transition $\tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}$ the quasi-invariant $\mathcal{Q}(\ell')$ is a precondition that implies safety for the assertion $(\tau_{\text{exit}}, \varphi)$. The following theorem states the correctness of this procedure.

Theorem 5.11. Let $\mathcal{C}$ be a component, $\mathcal{E}_\mathcal{C}$ its entry transitions, and $(\tau_{\text{exit}}, \varphi)$ an assertion with τ_{exit} an exit transition of $\mathcal{C}$ and φ a clause. If the procedure call $\mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau_{\text{exit}}, \varphi))$ returns $\mathcal{Q} \neq \mathsf{None}$, then $\mathcal{Q}$ is a quasi-invariant for $\mathcal{C}$ and $\mathcal{P}$ is $(\tau, \mathcal{Q}(\ell'))$-conditionally safe for $(\tau_{\text{exit}}, \varphi)$ for all $\rho = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}$.

Proof. That $\mathcal{Q}$ is a quasi-invariant follows directly from the structure of the generated constraints.

We prove the claim about conditional safety by contradiction via induction over the length of evaluations. Assume that there is an unsafe execution

$$(\ell_1, \sigma_1) \to_{\tau_1} (\ell_2, \sigma_2) \to_{\tau_2} \ldots \to_{\tau_n} (\ell_n, \sigma_n)$$

of length $n \in \mathbb{N}_{>1}$ such that $t_1 \in \mathcal{E}_\mathcal{C} \cup \mathcal{C}$ (i.e., ℓ_2 is always a location in $\mathcal{C}$), $\tau_n = \tau_{\text{exit}}$, $\sigma_2 \models \mathcal{Q}(\ell_2)$ and $\sigma_n \not\models \varphi$. We will show that no such evaluation can exist, implying our proposition as the special case $\tau_1 \in \mathcal{E}_\mathcal{C}$.

As the component graph is a DAG, $\tau_1 \in \mathcal{E}_\mathcal{C} \cup \mathcal{C}$ and τ_{exit} is an exit transition of $\mathcal{C}$, we have $\tau_i \in \mathcal{C}$ for all $1 < i < n$.

We first consider the case $n = 2$ ($n = 1$ would be the case where τ_1 is both an entry and exit transition, and thus infeasible). Let $\tau_2 = (\ell_1, \ell_2, \rho_2)$. Then, $\sigma_2 \models \mathcal{Q}(\ell_2) \equiv \delta(I_{\ell_2,k})$ by choice and definition, and $\delta(I_{\ell_2,k}) \wedge \rho_2 \to \varphi'$ by constraint $\mathbb{S}_k$. Thus, no unsafe evaluation of length 2 is possible.

We now assume $n > 2$ and that the proposition has been proven for evaluations of length $n-1$. Let $\tau_2 = (\ell_2, \ell_3, \rho_2)$. For length n, we have that the valuations σ_2, σ_3 satisfy ρ_2, and $\sigma_2 \models \mathcal{Q}(\ell_2) \equiv \delta(I_{\ell_2,k})$. These are the premises of our consecution constraint $\mathbb{C}_{\tau_2,k} \equiv I_{\ell_2,k} \wedge \rho_2 \to I_{\ell_3,k}$, and thus $\sigma_3 \models \delta(I_{\ell_3,k}) \equiv \mathcal{Q}(\ell_3)$. Hence, we instead have to consider the evaluation of length $n-1$ starting in (ℓ_2, σ_2), which by our hypothesis is infeasible. □

5.2.2 Propagating local conditions

In this section, we explain how to use the local procedure CondSafe to prove safety of a full program. To this end we now consider the full DAG of program components. As outlined above, the idea is to start from the assertion provided by the user, call the procedure CondSafe to obtain preconditions for the entry transitions of the corresponding component, and then use these preconditions as assertions for preceding components, continuing recursively. If eventually for each initial transition the transition relation implies the corresponding preconditions, then safety has been proven. If we fail to prove safety for certain assertions, we backtrack, trying further possible preconditions and quasi-invariants.

The key to the precision of our approach is our treatment of failed proof attempts. When the procedure CondSafe finds a quasi-invariant $\mathcal{Q}$ for $\mathcal{C}$, but proving $(\tau, \mathcal{Q}(\ell'))$ as a postcondition of the preceding component fails for some $\tau = (\ell, \ell', \rho) \in \mathcal{E}_{\mathcal{C}}$, we can still use the $\mathcal{Q}$ to *narrow* our program representation and filter out evaluations that are already known to be safe.

As outlined above, in our proof process we treat each clause of the conjunction $\mathcal{Q}(\ell')$ separately, and pass each one as its own assertion to preceding program components, allowing for a fine-grained *program-narrowing* technique. By construction of $\mathcal{Q}$, evaluations that satisfy all literals of $\mathcal{Q}(\ell')$ after executing $\tau = (\ell, \ell', \rho) \in \mathcal{E}_{\mathcal{C}}$ are safe. Thus, among the evaluations that use τ, we only need to consider those where at least one literal in $\mathcal{Q}(\ell')$ does not hold. Hence, we *narrow* each entry transition by conjoining it with the negation of the conjunction of all literals for which we could not prove safety (see line 19 in Algorithm 5.12). Note that if there is more than one literal in this conjunction, then the negation is a disjunction, which in our programming model implies splitting transitions. So, in order to avoid a combinatorial explosion, when narrowing a transition ρ with $\neg(L_1 \wedge \cdots \wedge L_n)$, our implementation is adding n transitions $\rho \wedge \neg L_1$, $\rho \wedge L_1 \wedge \neg L_2$, $\ldots$, $\rho \wedge L_1 \wedge \ldots \wedge \neg L_n$.

We can narrow program components similarly. For a transition $\tau = (\ell, \ell', \rho) \in \mathcal{C}$, we know that if either $\mathcal{Q}(\ell)$ or $\mathcal{Q}(\ell')'$ holds in an evaluation passing through t, the program is safe. Thus, we narrow the program by replacing ρ by $\rho \wedge \neg\mathcal{Q}(\ell) \wedge \neg\mathcal{Q}(\ell')'$ (see line 20 in Algorithm 5.12).

This narrowing allows us to generate disjunctive quasi-invariants, where each result of CondSafe is one disjunct. Note that not *all* disjunctive invariants can be discovered like this, as each intermediate result needs to

Algorithm 5.12 Proc. CheckSafe for proving a program safe for an assertion

Input: Program $\mathcal{P}$, a (possibly narrowed) component $\mathcal{C}$, (possibly narrowed) entries $\mathcal{E}_\mathcal{C}$, assertion $(\tau_{\text{exit}}, \varphi)$ s.t. τ_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause

Output: Safe | Maybe

let $(\ell_{\text{exit}}, \tau_{\text{exit}}, \ell'_{\text{exit}}) = \tau_{\text{exit}}$
if $(\rho_{\text{exit}} \to \varphi')$ **then**
  return Safe
else if $\ell_{\text{exit}} = \ell_0$ **then** {Base case}
  return Maybe
end if
$\mathcal{Q} \leftarrow \mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau_{\text{exit}}, \varphi))$ {Find quasi-invariant}
if $\mathcal{Q} = \mathsf{None}$ **then**
  return Maybe
end if
for all $\tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}$, $L \in \mathcal{Q}(\ell')$ **do** {Propagate backwards}
  $\tilde{\mathcal{C}} \leftarrow \mathsf{component}(\ell, \mathcal{P})$
  $\mathcal{E}_{\tilde{\mathcal{C}}} \leftarrow \mathsf{entries}(\tilde{\mathcal{C}}, \mathcal{P})$
  $\mathsf{res}[\tau, L] \leftarrow \mathsf{CheckSafe}(\mathcal{P}, \tilde{\mathcal{C}}, \mathcal{E}_{\tilde{\mathcal{C}}}, (\tau, L))$
end for
if $\forall \tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}, L \in \mathcal{Q}(\ell')$. $\mathsf{res}[\tau, L] = \mathsf{Safe}$ **then**
  return Safe {Precondition holds in all preceding SCCs}
else
  $\hat{\mathcal{E}}_\mathcal{C} \leftarrow \{(\ell, \rho \wedge \neg(\bigwedge_{\substack{L \in \mathcal{Q}(\ell') \\ \mathsf{res}[\tau, L] = \mathsf{Maybe}}} L'), \ell') \mid \tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}\}$
  $\hat{\mathcal{C}} \leftarrow \{(\ell, \rho \wedge \neg\mathcal{Q}(\ell')' \wedge \neg\mathcal{Q}(\ell), \ell') \mid (\ell, \ell', \rho) \in \mathcal{C}\}$ {Narrow component}
  return $\mathsf{CheckSafe}(\mathcal{P}, \hat{\mathcal{C}}, \hat{\mathcal{E}}_\mathcal{C}, (\tau_{\text{exit}}, \varphi))$
end if

be inductive using the disjuncts found so far. However, it works well for so-called *phase-change* algorithms [Sharma et al., 2011], where execution of a loop goes through different phases.

Based on this, we can now formulate our overall safety proving procedure CheckSafe in Algorithm 5.12. The procedure expects a program, a component, its entry transitions and an assertion $(\tau_{\text{exit}}, \varphi)$ as input. The helper procedures component and entries are used to find the program component for a given location and the entry transitions for a component. The result of CheckSafe is either Maybe when the proof failed, or Safe if it succeeded. In the latter case, we have managed to create a chain of quasi-invariants that

imply that $(\tau_{\text{exit}}, \varphi)$ always holds.

Finally, the next theorem claims that CheckSafe is sound.

Theorem 5.13. Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component and $\mathcal{E}_\mathcal{C}$ its entries. Given an assertion $(\tau_{\text{exit}}, \varphi)$ such that τ_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause, if $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau_{\text{exit}}, \varphi)) = \mathsf{Safe}$, then $\mathcal{P}$ is safe for $(\tau_{\text{exit}}, \varphi)$.

Proof. We prove the proposition by induction over the number u of recursive calls of CheckSafe.

In the base case $u = 0$, we have that $\tau_{\text{exit}} \to \varphi'$, i.e., the condition to prove is always a consequence of using the transition τ_{exit}, and the claim trivially holds.

Let now $u > 0$, and we assume that the proposition has been shown for all calls of CheckSafe that return Safe and need at most $u - 1$ recursive calls of CheckSafe.

We now consider a program evaluation

$$(\ell_0, \sigma_0) \to_{\tau_0} \ldots \to_{\tau_{m-1}} (\ell_m, \sigma_m) \to_{\tau_m} \ldots \to_{\tau_n} (\ell_n, \sigma_n)$$

with $\tau_{m-1} \in \mathcal{E}_\mathcal{C}$ and $\tau_n = \tau_{\text{exit}}$.

First, we consider the case that CheckSafe returns Safe in line 17, where all preconditions are satisfied for all entry transitions. By the condition $\forall L \in \mathcal{Q}(\ell_m).\ \mathsf{res}[\tau_{m-1}, L] = \mathsf{Safe}$ and our induction hypothesis, we know that the program is safe for $(\tau_{m-1}, \mathcal{Q}(\ell_m))$. By construction of $\mathcal{Q}$ and Theorem 5.11, this then implies that the program is safe for $(\tau_{\text{exit}}, \varphi)$.

The second case is returning the result of CheckSafe on the narrowed program in lines 19 and 20. For this, we need to prove that our program narrowing is indeed correct. Assume now that the considered evaluation is unsafe, i.e., that $\sigma_n \not\models \varphi$. We will show that our narrowing preserves unsafe evaluations. Then, as the recursive call of CheckSafe (with recursion depth $u - 1$) is correct by our induction hypothesis, we can conclude that Safe is only returned if there are no unsafe evaluations.

We first consider the narrowing $\hat{\mathcal{E}}_\mathcal{C}$. By our induction hypothesis, we know that $\sigma_m \models (\bigwedge_{L \in \mathcal{Q}(\ell_m), \mathsf{res}[\tau_{m-1}, L] = \mathsf{Safe}} L')$ holds. Now assume that the narrowed version of τ_{m-1} is not enabled anymore because of the added condition. Then $\sigma_m \not\models \neg(\bigwedge_{L \in \mathcal{Q}(\ell_m), \mathsf{res}[\sigma_{m-1}, L] = \mathsf{Maybe}} L')$ holds, and thus $\sigma_m \models \bigwedge_{L \in \mathcal{Q}(\ell_m)} L'$. But then, our evaluation is safe (by the same argument as in the proof of Theorem 5.11), contradicting our assumption that

the considered evaluation is unsafe. Thus, unsafe evaluations are not broken by our narrowing of entry transitions.

Similarly, we now consider the narrowing $\hat{\mathcal{C}}$. We consider an evaluation step $(\ell_w, \sigma_w) \to_{\tau_w} (\ell_{w+1}, \sigma_{w+1})$ with $\tau_w \in \hat{\mathcal{C}}$. If the narrowed version $\hat{\tau}_w \in \hat{\mathcal{C}}$ of τ_w cannot be used, then either $\sigma_w \models \mathcal{Q}(\ell_w)$ or $\sigma_{w+1} \models \mathcal{Q}(\ell_{w+1})$ holds, and again, by an argument similar to the proof of Theorem 5.11, this contradicts the assumption that our evaluation is unsafe. Thus, unsafe evaluations are preserved by narrowing of the program component. □

Example 5.14. We demonstrate CheckSafe on the program displayed on Figure 5.15, called $\mathcal{P}$ in the following, which is an extended version of the example from Figure 5.3.

We want to prove the assertion $(\tau_5, x \neq y)$. Hence we make a first call CheckSafe$(\mathcal{P}, \{\tau_4\}, \{\tau_3\}, (\tau_5, x \neq y))$: the non-trivial SCC containing ℓ_2 is $\{t_4\}$ and its entry transitions are $\{t_3\}$. Hence, we call CondSafe$(\{\tau_4\}, \{\tau_3\}, (\tau_5, x \neq y))$ and the resulting quasi-invariant for ℓ_2 is either $x < y$ or $y < x$. Let us assume it is $y < x$. In the next step, we propagate this to the predecessor SCC $\{\tau_2\}$, and call CheckSafe$(\mathcal{P}, \{\tau_2\}, \{\tau_1\}, (\tau_3, y < x))$.

In turn, this leads to calling CondSafe$(\{\tau_2\}, \{\tau_1\}, (\tau_3, y < x))$ to our synthesis subprocedure. No quasi-invariant supporting this assertion can be found, and hence None is returned by CondSafe, and consequently Maybe is returned by CheckSafe. Hence, we return to the original SCC $\{\tau_4\}$ and its entry $\{\tau_3\}$, and then by narrowing we obtain two new transitions:

$$\tau_4' = (\ell_2, \ell_2, x' = x + 1 \wedge y' = y + 1 \wedge \neg(y < x)),$$
$$\tau_3' = (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y \wedge \neg(y < x)).$$

Using these, we call CheckSafe$(\mathcal{P}, \{\tau_4'\}, \{\tau_3'\}, (\tau_5, x \neq y))$. The next call to CondSafe then yields the quasi-invariant $x < y$ at ℓ_2, which is in turn propagated backwards with the call CheckSafe$(\mathcal{P}, \{\tau_2\}, \{\tau_1\}, (\tau_3', x < y))$. This then yields a quasi-invariant $x < y$ at ℓ_1, which is finally propagated back in the call CheckSafe$(\mathcal{P}, \{\}, \{\}, (\tau_1, x < y))$, which directly returns Safe.

5.2.3 Improving performance

The basic method CheckSafe can be extended in several ways to improve performance. In the following, we present a number of techniques that are

useful to reduce the runtime of the algorithm and distribute the required work. It is important to note that none of these techniques influences the precision of the overall framework.

Using quasi-invariants to disable transitions When proving an assertion, it is often necessary to find invariants that show the unfeasibility of some transition, which allows disabling it. In our framework, the required invariants can be conditional as well. Therefore, CheckSafe must be called recursively to prove that the quasi-invariant is indeed invariant. In our implementation, we generate constraints such that every solution provides quasi-invariants either implying the postcondition or disabling some transition. By imposing different weights, we make the Max-SMT solver prefer solutions that imply the postcondition.

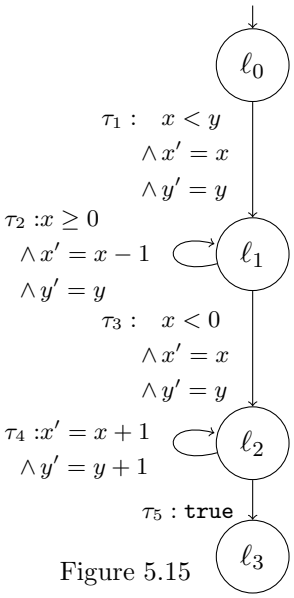


Figure 5.15

Handling unsuccessful proof attempts One important aspect is that the presented algorithm does not *learn* facts about the reachable state space, and so duplicates work when assertions appear several times. To alleviate this for *unsuccessful* recursive invocations of CheckSafe, we introduce a simple memoization technique to avoid repeating such calls. So when $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau, \varphi)) = \mathsf{Maybe}$, we store this result, and use it for all later calls of $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau, \varphi))$. This strategy is valid as the return value Maybe indicates that our method cannot prove the assertion (τ, φ) at all, meaning that later proof attempts will fail as well. In our implementation, this memoization of unsuccessful attempts is local to the initial call to CheckSafe. The rationale is that, when proving unrelated properties, it is likely that few calls are shared and that the book-keeping does not pay off.

Handling successful proof attempts When a recursive call yields a *successful* result, we can *strengthen* the program with the proven invariant. Remember that $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (\tau, \varphi)) = \mathsf{Safe}$ means that whenever the transition τ is used in *any* evaluation, φ holds in the succeeding state. Thus, we can make this knowledge explicit in the program and change the transi-

tion in the original program. In our implementation, this strengthening is applied only if the first call to CheckSafe was successful, i.e, no narrowing was applied. The reason is that, if the transition relation τ was obtained through repeated narrowing, in general one needs to split transitions, and it is not correct to just add φ' to τ.

Namely, assume that $\tau_o = (\ell, \ell', \rho_o)$ is the original (unnarrowed) version of a transition $\tau = (\ell, \ell', \rho) \in \mathcal{E}_\mathcal{C}$. As τ is an entry transition of $\mathcal{C}$, we have $\rho = \rho_o \wedge \neg\psi'_1 \wedge \ldots \wedge \neg\psi'_m$ by construction, where ψ_i is the additional constraint we added in the i-th narrowing of component entries. Thus what we proved is that $\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi'$ always holds after using τ_o. Hence, we should replace τ_o in the program with a transition labeled with $\rho_o \wedge (\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi')$. Since we cannot handle disjunctions natively, this implies replacing τ_o by $m + 1$ new transitions.

Note that, unlike memoization, this program modification makes the gained information available to the Max-SMT solver when searching for a quasi-invariant. A similar strategy can be used to strengthen the transitions in the considered component $\mathcal{C}$.

Parallelizing & distributing the analysis Our analysis can easily be parallelized. We have implemented this at two stages. First, at the level of the procedure CondSafe, we try at the same time different numbers of template conjuncts (lines 3-7 in Algorithm 5.10), which requires calling several instances of the solver simultaneously. Secondly, at a higher level, the recursive calls of CheckSafe (line 14 in Algorithm 5.12) are distributed onto several processes. Note that, since narrowing and the "learning" optimizations described above are considered only locally, they can be handled as asynchronous updates to the program kept in each worker, and do not require synchronization operations. Hence, distributing the analysis onto several worker processes, in the style of Bolt [Albarghouthi et al., 2012b], would be possible as well.

Other directions for parallelization, which have not been implemented yet, are to return different quasi-invariants in parallel when the Max-SMT problem in procedure CondSafe has several solutions. Moreover, based on experimental observations that successful safety proofs have a short successful path in the tree of proof attempts, we are also interested in exploring a look-ahead strategy: after calling CondSafe in CheckSafe, we could make

recursive calls of CheckSafe on some processes while others are already applying narrowing.

Iterative proving Finally, one could store the quasi-invariants generated during a successful proof, which are hence invariants, so that they can be reused in later runs. E.g., if a single component is modified, one can reprocess it and compute a new precondition that ensures its postcondition. If this precondition is implied by the previously computed invariant, the program is safe and nothing else needs to be done. Otherwise, one can proceed with the preceding components, and produce respective new preconditions in a recursive way. Only when proving safety with the previously computed invariants in this way fails, the whole program needs to be reprocessed again. This technique has not been implemented yet, as our prototype is still in a preliminary state.

5.3 Related work

Safety proving is an active area of research. In the recent past, techniques based on variations of counterexample guided abstraction refinement have dominated [Ball and Rajamani, 2001; Henzinger et al., 2003; Clarke et al., 2005; McMillan, 2006; Podelski and Rybalchenko, 2007; Bradley, 2011; Grebenshchikov et al., 2012; Albarghouthi et al., 2012a; Cimatti and Griggio, 2012]. These methods prove safety by repeatedly unfolding the program relation using a symbolic representation of program states, starting in the initial states. This process generates an over-approximation of the set of reachable states, where the coarseness of the approximation is a consequence of the used symbolic representation. Whenever a state in the over-approximation violates the safety condition, either a true counterexample was found and is reported, or the approximation is refined (using techniques such as predicate abstraction [Flanagan and Qadeer, 2002] or Craig interpolation [McMillan, 2003a]). When further unwinding does not change the symbolic representation, all reachable states have been found and the procedure terminates. This can be understood as a "top-down" approach (starting from the initial states), whereas our method is "bottom-up" (starting from assertions).

Techniques based on Abstract Interpretation [Cousot and Cousot, 1977b] have had substantial success in the industrial setting. There, an abstract

interpreter is instantiated by an abstract domain whose elements are used to over-approximate sets of program states. The interpreter then evaluates the program on the chosen abstract domain, discovering reachable states. A widening operator, combining two given over-approximations to a more general one representing both, is employed to guarantee termination of the analysis when handling loops.

Recently, the use of abduction (i.e., inference of preconditions for certain facts) in safety proving has been investigated [Dillig et al., 2013]. This work is closest to ours in its overall approach, but uses fundamentally different techniques to find preconditions. It also searches for inductive invariants using a backwards-reasoning technique, constructing verification conditions similar to our constraint systems. However, instead of applying Max-SMT, the approach uses an abduction engine based on maximal universal subsets and quantifier elimination in Presburger arithmetic. Moreover, it does not have an equivalent to our narrowing to exploit failed proof attempts, though a syntactic version of it [Sharma et al., 2011] could be combined with the method. In a similar vein, [Păsăreanu and Visser, 2004] uses straight-line weakest precondition computation and backwards-reasoning to infer loop invariants supporting validity of an assertion. To enforce a generalization towards inductive invariants, a heuristic syntax-based method is used.

Automatically constructing program proofs from independently obtained subproofs has been an active area of research in the recent past. Splitting proofs along syntactic boundaries (e.g., handling procedures separately) has been explored in [Godefroid, 2007; Yorsh et al., 2008; Godefroid et al., 2010; Calcagno et al., 2011; Albarghouthi et al., 2012b]. For each such unit, a summary of its behavior is computed, i.e., an expression that connects certain (classes of) inputs to outputs. Depending on the employed analyzers, these summaries encode inputs that lead to errors [Godefroid, 2007], under- and over-approximations of reachable states [Godefroid et al., 2010], or changes to the heap using separation logic's frame rule [Calcagno et al., 2011]. Finally, [Albarghouthi et al., 2012b] discusses how such compositional analyses can leverage cloud computing environments to parallelize and scale up program proofs.

5.4 Implementation and evaluation

We have implemented the algorithms from Section 5.2.1 and Section 5.2.2 in our early prototype VeryMax, using the Max-SMT solver for non-linear arithmetic in the Barcelogic system (see Section 2.1.3).

The first set (which we will call HOLA-BENCHS) are the 46 programs from the evaluation of safety provers in Dillig et al. [2013] (which were collected from a variety of sources, among others, [Gupta and Rybalchenko, 2009; Gulwani et al., 2008b; Beyer et al., 2007c; Gulavani et al., 2008; Bradley and Manna, 2008; Miné, 2006; Jhala and McMillan, 2006; Sharma et al., 2011, 2012; Gulavani et al., 2006; Gulavani and Rajamani, 2006; Dillig et al., 2012], the NECLA Static Analysis Benchmarks, etc.). The programs are relatively small (they have between 17 and 71 lines of code, and between 1 and 4 nested or consecutive loops), but expose a number of "hard" problems for analyzers. All of them are safe.

On this first benchmark set we compare with three systems. The first two were leading tools in the Software Verification Competition 2015 [Beyer, 2015]: CPAchecker[5] [Beyer and Keremoglu, 2011], which was the overall winner and in particular won the gold medal in the *"Control Flow and Integer Variables"* category, and SeaHorn [Kahsai et al., 205], which got the silver medal, and also won the *"Simple"* category. We also compare with HOLA [Dillig et al., 2013], an abduction-based backwards reasoning tool. Unfortunately, we were not able to obtain an executable for HOLA. For this reason we have taken the experimental data for this tool directly from [Dillig et al., 2013], where it is reported that the experiments were performed on an Intel i5 2.6 GHz CPU with 8 Gb of memory. For the sake of a fair comparison, we have run the other tools on a 4-core machine with the same specification, using the same timeout of 200 seconds. Table 5.16 summarizes the results, reporting the number of successful proofs, failed proofs, and timeouts (TO), together with the respective total runtimes. Overall, we can see that both versions of VeryMax are competitive, and that our parallel version was two times faster than our sequential one on four cores. As a reference, on these examples VeryMax-Seq needed 2.8 overall calls (recursive or after narrowings) on average, with a maximum of

[5]We ran CPAchecker with two different configurations, predicateAnalysis (PA) and sv-comp15 (SV).

16. The number of narrowings was approximately 1, with a maximum of 13. Our memoization technique making use of already failed proof attempts was employed in about one third of the cases.

Tool	Safe	Σ s	Fail	Σ s	TO	Total s
CPAchecker-SV	33	2424.41	3	61.28	10	4489.73
CPAchecker-PA	25	503.05	11	19.72	10	2271.12
SeaHorn	32	7.95	13	3.477	1	211.56
HOLA	43	23.53	0	0	3	623.53
VeryMax-Seq	43	330.25	2	42.00	1	572.27
VeryMax-Par	44	180.28	2	74.69	0	254.97

Table 5.16. Experimental results on HOLA-BENCHS benchmark set.

In our second benchmark set (which we will refer to as NR-BENCHS) we have used integer abstractions of 217 numerical algorithms from [Press et al., 2002]. For each procedure and for each array access in it, we have created two safety problems with one assertion each, expressing that the index is within bounds. In some few cases the soundness of array accesses in the original program depends on properties of floating-point variables, which are abstracted away. So in the corresponding abstraction some assertions may not hold. Altogether, the resulting benchmark suite consists of 6452 problems, of up to 284 lines of C code. Due to the size of this set, and to give more room to exploit parallelism (both tools with which we compare on these benchmarks, CPAchecker and SeaHorn, make use of several cores), we performed the experiments with a more powerful machine, namely, an 8-core Intel i7 3.4 GHz CPU with 16 GB of memory. The time limit is 300 seconds.

Tool	Safe	Σ s	Unsafe	Σ s	Fail	Σ s	TO	Total s
CPAchecker-SV	5978	534621.26	282	9218.96	61	10797.01	131	591886.32
CPAchecker-PA	5854	21230.37	221	758.48	159	774.68	218	79624.17
SeaHorn	6081	4315.31	235	149.71	72	18.07	64	24287.32
VeryMax-Seq	6098	5953.48	0	0	299	20661.59	55	43116.71
VeryMax-Par	6098	4335.10	0	0	354	21034.34	0	25369.44

Table 5.17. Experimental results on NR-BENCHS benchmark set.

The results can be seen in Table 5.17. On these instances, VeryMax is able to prove more assertions than any of the other tools, while being about as fast as SeaHorn, and significantly faster than CPAchecker. Note that VeryMax is at an early stage of development, and is not yet

fully tuned. For example, a number of program slicing techniques have not been implemented yet, which would be very useful for handling larger programs. Thus, we expect that further development will improve the tool performance significantly. The benchmarks and our tool can be found at www.cs.upc.edu/~albert/VeryMax.html.

Chapter 6

Conclusions

In this thesis we have obtained results in program analysis using as starting point the constraint-based method [Colón et al., 2003] and its application to termination analysis [Bradley et al., 2005]. In contrast to the original presentation of this method, in our approach we replace the use of constraint solving techniques by the use of optimization-based techniques, being Max-SMT our key tool for handling constraints. Thanks to this, we have obtained a new method for proving automatically termination [Larraz et al., 2013a] and non-termination [Larraz et al., 2014a] of sequential programs, and developed a new framework for compositional analysis [Brockschmidt et al., 2015] of program properties. Additionally, we have provided a version of the constraint-based method that applies to generate invariants for programs with arrays [Larraz et al., 2013b].

In Chapter 3 we present our new constraint-based method for the generation of universally quantified invariants of array programs. Unlike other techniques, it does not require extra predicates nor assertions. It does not need the user to provide a template either, but it can take advantage of hints by partially instantiating the global template considered here. We also provide extensions of the approach for sorted arrays. To our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature.

For future work, we plan to extend our approach to a broader class of programs. As a first step we plan to relax Theorem 3.13, so that, e.g., overwriting on positions in which the invariant already holds is allowed. We would also like to handle nested loops, so that for instance sorting algo-

rithms can be analyzed. Another line of work is the extension of the family of properties that our approach can discover as invariants. E.g., a possibility could be considering disjunctive properties, or allowing quantifier alternation. The former allows analyzing algorithms such as sentinel search, while the latter is necessary to express that the output of a sorting algorithm is a permutation of the input.

Moreover, the invariants that our method generates depend on the coefficients and expressions obtained in each of its three phases, which in turn depend on the previous linear relationship analysis of scalar variables. We leave for future research to study how to make the approach resilient to changes in the outcome of the different phases, paying special attention to the use of Max-SMT techniques.

In Chapter 4 we describe new methods for proving and disproving termination. In Section 4.3 we present our novel Max-SMT constraint-based approach to proving termination. Thanks to expressing the synthesis of a ranking function and a supporting invariant as a Max-SMT problem, we achieve a better guided and more fine-grained termination analysis than SMT-based methods. Max-SMT reveals to be a convenient framework for constraint-based termination analysis. In addition to our method, other techniques such as *unaffecting score maximization* [Cook et al., 2013] can be naturally modeled in Max-SMT. However, one of the shortcomings of our approach as it was presented, is that invariant synthesis is restricted to a single strongly connected component (SCC). If invariants from previous SCC's have not been generated but are later required, our technique cannot prove termination. But this can be fixed integrating our termination proving technique into our compositional analysis framework, described in Chapter 5, as it is explained later on.

For future work, an interesting line of research would be to adjust and incorporate to our approach already known techniques from static analysis of programs and automated termination proving. E.g., fixpoint computation à la abstract interpretation [Cousot and Cousot, 1977b] would complement the constraint-based approach in the generation of invariants.

Other techniques from abstract interpretation, such as *view abstractions* [Elder et al., 2010], could also be useful so as to extend the class of ranking functions that we can discover.

Another possible hybridization could be the combination with transition

invariant-based techniques. The point is that even when we do not succeed in proving termination, at least we prove that some part of the transitive closure of the transition relation is disjunctively well-founded, and moreover we are left with a transition system that characterizes the part of the transitive closure that may still contain an infinite execution. Thus, termination counter-examples can still be found by analyzing this residual transition system. On the other hand, if this part of the transition relation is proved disjunctively well-founded too, we can conclude that the program is terminating.

In Section 4.4 we introduce a novel Max-SMT-based technique for proving that programs do not terminate. The key notion of the approach is that of a *quasi-invariant*, which is a property such that if it holds at a location during execution once, then it continues to hold at that location from then onwards. The method considers an *Strongly Connected SubGraph (SCSG)* of the control flow graph at a time, and thanks to Max-SMT solving generates a quasi-invariant for each location. Weights of soft constraints guide the solver towards quasi-invariants that are also *edge-closing*, i.e., that forbid any transition exiting the SCSG. If an SCSG with edge-closing quasi-invariants is reachable, then the program is non-terminating. This last check is performed with an off-the-shelf reachability checker. We have reported experiments with encouraging results that show that a prototypical implementation of the proposed approach has comparable and often better results than state-of-the-art non-termination provers.

As regards future research, a pending improvement is to couple the reachability checker with the quasi-invariant generator, so that the invariants synthesized by the former in unsuccessful attempts are reused by the latter when producing quasi-invariants. Another line for future work is to combine our termination and non-termination techniques. Following a similar approach to [Brockschmidt et al., 2013], if the termination analyzer fails, it can communicate to the non-termination tool the transitions that were proved not to belong to any infinite computation. Conversely, when a failed non-termination analysis ends with an unsuccessful reachability check, one can pass the computed invariants to the termination system, as done in [Harris et al., 2011]. We also plan to extend our programming model to handle more general programs (procedure calls, non-linearities, etc.).

Finally, in Chapter 5 we present a novel approach to compositional safety

verification. Our main contribution is a proof framework that refines intermediate results produced by a Max-SMT-based precondition synthesis procedure. In contrast to most earlier work, we proceed *bottom-up* to compute summaries of code that are guaranteed to be relevant for the proof.

We plan to further extend VeryMax to cover more program features and include standard optimizations (e.g., slicing and constraint propagation with simple abstract domains). It currently handles procedure calls by inlining, and does not support recursive functions yet. We plan to deal with such cases similarly to loops, by introducing templates for function pre/postconditions.

In the future, we are interested in experimenting with alternative precondition synthesis methods (e.g., abduction-based ones). We also want to combine our method with a Max-SMT-based termination proving method, and extend it to *existential* properties such as reachability and non-termination. We expect to combine all of these techniques in an *alternating* procedure like the one explained in [Godefroid et al., 2010] that tries to prove properties at the same time as their duals, and which uses partial proofs to narrow the state space that remains to be considered. Eventually, these methods could be combined to verify arbitrary temporal properties. In another direction, we want to consider more expressive theories to model program features such as arrays or the heap.

Bibliography

Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *Verification, Model Checking, and Abstract Interpretation*, pages 39–55. Springer, 2012a.

Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing top-down interprocedural analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 217–228, 2012b. doi: 10.1145/2254064.2254091.

Teresa Alsinet, Felip Manyà, and Jordi Planes. An efficient solver for weighted max-sat. *Journal of Global Optimization*, 41(1):61–73, 2008.

RC Andreas, B Cook, A Podelski, and A Rybalchenko. Terminator: Beyond safety. *In CAV06, LNCS*, 4144:415–418, 2006.

Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. Sat-based procedures for temporal reasoning. In *Recent Advances in AI Planning*, pages 97–108. Springer, 2000.

Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *Computer Aided Verification*, pages 210–226. Springer, 2012.

Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *Automated DeductionCADE-18*, pages 195–210. Springer, 2002.

Thomas Ball and Sriram K Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Model Checking and Software Verification*, pages 113–130. Springer, 2000.

Thomas Ball and Sriram K Rajamani. The slam toolkit. In *Computer Aided Verification*, pages 260–264. Springer, 2001.

Thomas Ball and Sriram K Rajamani. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

Clark W Barrett, David L Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to sat. In *Computer Aided Verification*, pages 236–249. Springer, 2002.

Josh Berdine, Byron Cook, Dino Distefano, and Peter W Ohearn. Automatic termination proofs for programs with shape-shifting heaps. In *Computer Aided Verification*, pages 386–400. Springer, 2006.

Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *ACM SIGPLAN Notices*, volume 42, pages 211–224. ACM, 2007.

Dirk Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.

Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

Dirk Beyer, Adam J Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE Computer Society, 2004.

Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007a.

Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking, and Abstract Interpretation*, pages 378–394. Springer, 2007b.

Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007c.

Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(75-97):45, 2008.

Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. ISBN 978-1-58603-929-5.

Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic smt solver. In *Computer Aided Verification*, pages 294–298. Springer, 2008.

Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.

Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

Aaron R Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20(4-5):379–405, 2008.

Aaron R Bradley, Zohar Manna, and Henny B Sipma. Linear ranking with reachability. In *Computer Aided Verification*, pages 491–504. Springer, 2005.

Aaron R Bradley, Zohar Manna, and Henny B Sipma. Whats decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexceptions for java bytecode. In *Formal Verification of Object-Oriented Software*, pages 123–141. Springer, 2012.

Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Computer Aided Verification*, pages 413–429. Springer, 2013.

Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Compositional safety verification with Max-SMT. 2015. Submitted, 2015.

Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011.

Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *Programming Languages and Systems*, pages 148–162. Springer, 2008.

Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter OHearn. Proving nontermination via safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–171. Springer, 2014.

Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Computer Aided Verification*, pages 277–293. Springer, 2012.

Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *Static Analysis*, pages 312–327. Springer, 2004.

Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574. Springer, 2005.

Michael A Colón and Henny B Sipma. Practical methods for proving program termination. In *Computer Aided Verification*, pages 442–454. Springer, 2002.

Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification*, pages 420–432. Springer, 2003.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y Vardi. Proving that programs eventually do something good. In *ACM SIGPLAN Notices*, volume 42, pages 265–276. ACM, 2007.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 47–61. Springer, 2013.

Patrick Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, pages 243–268. Springer, 2004.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. *ACM SIGOPS Operating Systems Review*, 11(2):77–94, 1977a.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977b.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

Leonardo De Moura, Harald Rueß, and Maria Sorea. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*. Citeseer, 2002.

Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 181–192, 2012. doi: 10.1145/2254064.2254087.

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *ACM SIGPLAN Notices*, volume 48, pages 443–456. ACM, 2013.

Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

Matt Elder, Denis Gopan, and Thomas W. Reps. View-augmented abstractions. *Electr. Notes Theor. Comput. Sci.*, 267(1):43–57, 2010.

Bernard Elspas, Karl N Levitt, Richard J Waldinger, and Abraham Waksman. An assessment of techniques for proving program correctness. *ACM Computing Surveys (CSUR)*, 4(2):97–147, 1972.

Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, volume 37, pages 191–202. ACM, 2002.

Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B Saxe. Theorem proving using lazy proof explication. In *Computer Aided Verification*, pages 355–367. Springer, 2003.

Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

Steven M German and Ben Wegbreit. A synthesizer of inductive assertions. *Software Engineering, IEEE Transactions on*, 1:68–75, 1975.

Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1):177–210, 1998.

Jurgen Giesl, Peter Schneider-Kamp, and Rene Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130, page 281. Springer, 2006.

Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54, 2007. doi: 10.1145/1190216.1190226.

Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010. doi: 10.1145/1706299.1706307.

Evgueni Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 142–149. IEEE, 2002.

Denis Gopan and Thomas Reps. Lookahead widening. In *Computer Aided Verification*, pages 452–466. Springer, 2006.

Denis Gopan and Thomas Reps. Guided static analysis. In *Static Analysis*, pages 349–365. Springer, 2007.

Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, pages 72–83. Springer, 1997.

Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, pages 169–192. Springer, 1991.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416, 2012. doi: 10.1145/2254064.2254112.

Bhargav S Gulavani and Sriram K Rajamani. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 474–488. Springer, 2006.

Bhargav S Gulavani, Thomas A Henzinger, Yamini Kannan, Aditya V Nori, and Sriram K Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–127. ACM, 2006.

Bhargav S Gulavani, Supratik Chakraborty, Aditya V Nori, and Sriram K Rajamani. Automatically refining abstract interpretations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–458. Springer, 2008.

Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008a.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 281–292, 2008b. doi: 10.1145/1375581.1375616.

Ashutosh Gupta and Andrey Rybalchenko. INVGEN: An efficient invariant generator. In *Computer Aided Verification*, pages 634–640. Springer, 2009.

Ashutosh Gupta, Thomas A Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *ACM SIGPLAN Notices*, volume 43, pages 147–158. ACM, 2008.

Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

William R Harris, Akash Lal, Aditya V Nori, and Sriram K Rajamani. Alternation for termination. In *Static Analysis*, pages 304–319. Springer, 2011.

Ben Hegbreitt. Heuristic methods for mechanically deriving inductive assertions. In *Proceedings of the 3rd international joint conference on Artificial Intelligence*, pages 524–536. Morgan Kaufmann Publishers Inc., 1973.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *ACM SIGPLAN Notices*, volume 37, pages 58–70. ACM, 2002.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking Software*, pages 235–239. Springer, 2003.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. *ACM SIGPLAN Notices*, 39(1):232–244, 2004.

Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. Valigator: A verification tool with bound and invariant generation. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 333–342. Springer, 2008.

Thomas A Henzinger, Thibaud Hottelier, Laura Kovács, and Andrey Rybalchenko. Aligators for arrays (tool paper). In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–356. Springer, 2010a.

Thomas A Henzinger, Thibaud Hottelier, Laura Kovács, and Andrei Voronkov. Invariant and type inference for matrices. In *Verification, Model Checking, and Abstract Interpretation*, pages 163–179. Springer, 2010b.

Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research*, 31(1):1–32, 2008.

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

Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Case studies on invariant generation using a saturation theorem prover. In *Advances in Artificial Intelligence*, pages 1–15. Springer, 2011.

Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 14–25. ACM, 2000.

Ranjit Jhala and Kenneth L McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 459–473. Springer, 2006.

Ranjit Jhala and Kenneth L McMillan. Array abstractions from proofs. In *Computer Aided Verification*, pages 193–206. Springer, 2007.

Dejan Jovanović and Leonardo De Moura. Solving non-linear arithmetic. In *Automated Reasoning*, pages 339–354. Springer, 2012.

Temesghen Kahsai, Jorge A Navas, Arie Gurfinkel, and Anvesh Komuravelli. The SeaHorn verification framework. In *Computer Aided Verification*, 205.

Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

Shmuel Katz and Zohar Manna. *A heuristic approach to program verification*. Stanford University, 1973.

Shmuel Katz and Zohar Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

Laura Kovács. Reasoning algebraically about p-solvable loops. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–264. Springer, 2008.

Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. Loopfrog: A static analyzer for ansi-c programs. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 668–670. IEEE, 2009.

Shuvendu K Lahiri and Randal E Bryant. Predicate abstraction with indexed predicates. *ACM Transactions on Computational Logic (TOCL)*, 9(1):4, 2007.

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using Max-SMT. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 218–225. IEEE, 2013a.

Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In *Verification, Model Checking, and Abstract Interpretation*, pages 169–188. Springer, 2013b.

Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using Max-SMT. In *Computer Aided Verification*, pages 779–796. Springer, 2014a.

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Minimal-model-guided approaches to solving polynomial constraints and extensions. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 333–350, 2014b.

Boyang Li, Isil Dillig, Thomas Dillig, Ken McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 370–384. Springer, 2013.

Han Lin, Kaile Su, and Chu-Min Li. Within-problem learning for efficient lower bound computation in max-sat solving. In *Proceedings of the 23rd national conference on Artificial intelligence*, pages 351–356, 2008.

Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 134–143. ACM, 2007.

Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Computer Aided Verification*, pages 401–414. Springer, 2006.

Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

Ken L McMillan. Craig interpolation and reachability analysis. In *Static Analysis*, pages 336–336. Springer, 2003a.

Kenneth L McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003b.

Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer, 2006.

Kenneth L McMillan. Quantified invariant generation using an interpolating saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2008.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

Robert Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 156–169. Springer, 2006.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of java bytecode by term rewriting. In *RTA*, pages 259–276, 2010.

Corina S Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Model Checking Software*, pages 164–181. Springer, 2004.

Étienne Payet and Fausto Spoto. Experiments with Non-Termination Analysis for Java Bytecode. *Electr. Notes Theor. Comput. Sci.*, 253(5):83–96, 2009.

Jordi Petit, Omer Giménez, and Salvador Roura. Jutge.org: an educational programming judge. In *SIGCSE*, pages 445–450, 2012.

Knot Pipatsrisawat, Akop Palyan, Mark Chavira, Arthur Choi, and Adnan Darwiche. Solving weighted max-sat problems in a reduced search space: A performance analysis. *Journal on Satisfiability Boolean Modeling and Computation*, 4: 191–217, 2008.

David A Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *ACM SIGPLAN Notices*, volume 40, pages 132–144. ACM, 2005.

Andreas Podelski and Andrey Rybalchenko. Armc: the logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages*, pages 245–259. Springer, 2007.

Andreas Podelski and Thomas Wies. Boolean heaps. In *Static Analysis*, pages 268–283. Springer, 2005.

Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2002.

John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

Lawrence Ryan. *Efficient algorithms for clause-learning SAT solvers*. PhD thesis, Theses (School of Computing Science)/Simon Fraser University, 2004.

Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, June 1998. ISBN 9780471982326.

Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification*, pages 703–719. Springer, 2011.

Rahul Sharma, Aditya V Nori, and Alex Aiken. Interpolants as classifiers. In *Computer Aided Verification*, pages 71–87. Springer, 2012.

Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.

Alfred Tarski. A decision method for elementary algebra and geometry. *Collected Works of A. Tarski*, 1953.

Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95. Springer, 2011.

Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

Alan M Turing. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Tests and Proofs*, pages 154–170. Springer, 2008.

Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivančić. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Computer Aided Verification*, pages 352–365. Springer, 2007.

Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17 (2):102–113, 1974.

Yichen Xie and Alex Aiken. Saturn: A sat-based tool for bug detection. In *Computer Aided Verification*, pages 139–143. Springer, 2005.

Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 221–234, 2008. doi: 10.1145/1328438.1328467.