Symbolic Abstraction with SMT Solvers

by

Yi Li

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Symbolic Abstraction with SMT Solvers

Yi Li
Master of Science
Graduate Department of Computer Science
University of Toronto
2013

Numerical invariant generation is an important program analysis task with uses spanning compiler optimizations, bug finding, and verification. Abstract interpretation with numerical domains is one of the most scalable automated techniques for computing numerical invariants. Unfortunately, efficiency is often achieved by employing imprecise operations (e.g., *join* and *abstract post*). Moreover, expressive domains such as *polyhedra* are expensive, and efficient domains such as *intervals* are often too weak to prove interesting program properties.

In this thesis, we present Symba, a novel algorithm that harnesses the power and precision of *SMT solvers*, in order to (1) implement precise (best) abstract transformers over large program segments described as formulas in linear real arithmetic (QF_LRA), thus avoiding the imprecision incurred by abstract post computations over single instructions or basic blocks; (2) avoid the use of imprecise join operations by *symbolically* and efficiently enumerating program paths; and (3) enable invariant generation in the *Template Constraint Matrix* (TCM) domain, a parameterized domain that subsumes intervals, octagons, and octahedra, amongst other domains.

Acknowledgements

This thesis would not be possible without the support of a large number of people who have helped me in different ways.

In particular, I would like to thank my adviser, Marsha Chechik, for her support, encouragement, patience and trust. She has shown me how to communicate ideas more effectively and spent hours on improving my writing.

I also owe a debt of gratitude to Arie Gurfinkel whose knowledge and insights were invaluable in every stage of this project. By being demanding, criticizing and understanding, he has driven me to success.

There are two others who made this possible. Aws Albarghouthi, from the very beginning, has been a welcoming ear of random ideas, a great teacher and friend, and helped me tremendously during my Master's study. The inspiration of this work certainly stems from Zachary Kincaid, who also gave insightful comments in numerous occasions and provided a critical eye throughout the project.

Finally, I would also like to thank my parents for their unparalleled love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Software has become an inseparable part of our lives nowadays. Many software products affect or relate closely to public security and health care, where their reliability is of paramount importance. For example, modern avionics systems rely on software to navigate and control large aircraft that carries hundreds of people. Artificial cardiac pacemaker helps maintain adequate heart rate for patients who suffer from arrhythmia. Software systems in pacemakers monitor the body condition of patients and send out correct electrical signals to hearts. Software failures in such systems are usually life-threatening.

Software verification is an automatic technique that ensures that software meets all the expected requirements and is free of bugs. Approaches to verification can be divided into two fundamental categories, namely dynamic verification that includes traditional testing and experimentation techniques, as well as static verification which is also known as static analysis or program analysis. The static approach is able not only to find bugs but also prove correctness of a program. Research on static analysis has been active for decades and many exciting advances have been made in this field since then. The technique has been successfully applied in proving the correctness of many software systems expressed as source code, such as software in medical devices and nuclear plants.

Program analysis relies on invariants for reasoning about sets of reachable states [1]. Program invariant refers to a proposition associated with a particular program location which is asserted to true whenever the location is reached. Examples include $x = 2y + 1$, "`n=n.child.parent`" and "elements in array a are sorted in ascending order". We can infer interesting properties of a program given good invariants. A large portion of program predicates can be represented by numerical expressions, that is, arithmetic

```
1: n=0, x=0;
2: while (n<100)
3:   n=n+1;
4:   x=x+n;
```

Figure 1.1: Example for illustrating numerical invariant.

relations that hold among numerical variables in a program (e.g., $x = 2y+1$). Numerical invariant generation is thus a key task for analysing programs involving numerical data and operations. As a simple example for numerical invariant, consider a program that sums integers from 1 to 100 (Fig. 1.1). A useful invariant at line 4 is $x = \sum_{i=1}^{n} i$. Suppose we know that $n \leqslant 100$ always holds; we can then prove that $x \leqslant \sum_{i=1}^{100} = 5050$.

Abstract interpretation [2] with numerical domains (such as intervals [3] or octagons [4]), is one of the most scalable automated techniques for computing numerical invariants. While highly efficient, abstract interpretation loses valuable precision due to the following factors:

- Imprecise operations such as *join*, used to avoid exploring exponentially many paths in the size of the program by merging abstract states; and *abstract post* used to interpret program statements in the abstract domain.
- Limitations in the used abstract domain. For example, the *polyhedra* domain [5] allows representation of arbitrary linear inequalities, but suffers from exponential time and space complexity. While domains such as intervals and octagons are more efficient, they can only represent restricted forms of relationships among variables, often limiting their ability to prove certain program properties.

In this thesis, we present Symba, a novel abstract interpretation algorithm that exploits advances in decision procedures to (1) implement *precise* (best) abstract transformers over large program segments, thus avoiding the imprecision incurred by abstract post computations over single instructions and basic blocks; (2) avoid the use of imprecise join operations by *symbolically* and efficiently enumerating program paths; and (3) enable invariant generation in expressive numerical abstract domains.

To achieve these goals, Symba implements the following *symbolic abstraction* algorithm: Given a formula φ and some abstract domain $\mathcal{A}$, find the *best* (tightest) abstraction of φ in $\mathcal{A}$. This allows us to encode a loop-free program fragment as a formula φ (e.g., as in verification condition encodings and bounded model checking [6, 7]) and use the symbolic abstraction algorithm to compute the best abstraction $a \in \mathcal{A}$ of φ. As a result, a can represent an over-approximation of concrete states reachable after executing

```
1: x = y + 1;
2: y = x - y;
3: assert(y == 1);
```

(a)

```
1: if (x < 0)
2:   x = x - 10;
3: else x = x + 10;
4: x = abs(x);
5: assert(x != 5);
```

(b)

Figure 1.2: Examples for illustrating symbolic abstraction.

the program fragment, with no precision loss due to joins or abstract transformers over single instructions. We illustrate the power of Symba below.

1.2 Symbolic Abstraction

Consider the example in Fig. 1.2a. We would like to prove that the assertion is never violated by computing an over-approximation of the set of reachable states at line 3. Suppose we are using the intervals domain in a standard abstract interpretation fashion. After interpreting the first statement, we get the abstract value $\top$, denoting that the bounds of `x` and `y` are unknown. Similarly, after interpreting the second statement, we still get $\top$ as the set of reachable states at line 3. Therefore, we conclude that every possible value for `y` is reachable at line 3, indicating a potential assertion violation. Suppose, instead, that we encode the execution of the three instructions as a formula in quantifier-free linear real arithmetic (QF_LRA), a commonly used theory for encoding program executions [8, 9, 10, 11]. This results in a formula

$$\varphi \equiv x_0 = y_0 + 1 \wedge y_1 = x_0 - y_0,$$

where subscripts are added to represent different versions of the same variable. Given a symbolic abstraction algorithm, we can compute the *best* abstraction $y = 1$ of φ (again, using intervals), since the algorithm realizes that the only satisfying assignment to φ sets y_1 to 1, where y_1 represents the value of `y` at line 3. Therefore, using a symbolic abstraction algorithm, we are able to avoid precision loss due to abstract post computations over single instructions by treating a number of instructions monolithically.

Now consider the example in Fig. 1.2b. We would like to prove that the assertion at line 5 is never violated. An abstract interpreter using intervals joins the abstract states from the two branches of the `if` statement, namely, $x < -10$ and $x \geqslant 10$, resulting in the abstract state $\top$ at line 4. After executing `x = abs(x)` (absolute value), the abstract interpreter concludes that all states where $x \geqslant 0$ are reachable at line 5, thus indicating that the assertion may be violated. On the other hand, using symbolic abstraction, we represent all executions that can reach line 5 using a formula

$$\begin{aligned} \varphi \equiv \quad & ite(x_0 < 0, x_1 = x_0 - 10, x_1 = x_0 + 10) \quad & \text{[lines 1-3]} \\ & \wedge\, ite(x_1 \geqslant 0, x_2 = x_1, x_2 = -x_1) & \text{[line 4]} \end{aligned}$$

where *ite* is the logical if-then-else construct and x_2 represents the value of variable `x` at line 5. Note that all satisfying assignments of φ ensure that $x_2 \geqslant 10$. Since a symbolic abstraction algorithm computes the tightest bound for the variable `x`, it computes the abstract state $x \geqslant 10$ at line 5, allowing us to avoid the imprecise join at line 3 and prove that the assertion at line 5 always holds.

1.3 Symbolic Abstraction with Symba

Symba exploits the power of *Satisfiability Modulo Theories* (SMT) solvers to compute the most precise abstractions of QF_LRA formulas in the general *Template Constraint Matrix* (TCM) [12] domain – a parameterized domain that subsumes intervals, octagons, and octahedra, among others. Specifically, Symba takes a formula φ and a set of templates T as input, where each $t_i \in T$ is a linear expression $c_1x_1 + \cdots + c_nx_n$ over the variables $\{x_i\}_i$ of φ and $c_i \in \mathbb{R}$. Symba computes the strongest formula φ', such that $\varphi \Rightarrow \varphi'$ and

$$\varphi' \equiv t_1 \leqslant k_1 \wedge \cdots \wedge t_{|T|} \leqslant k_{|T|},$$

where $k_i \in \mathbb{R} \cup \{\infty\}$. In other words, Symba computes the least upper bound k_i for each template t_i within φ, where $k_i = \infty$ implies that the template t_i is unbounded. For the example in Fig. 1.2b, $T = \{x, -x\}$, indicating that we would like to find the tightest upper and lower bounds of variable `x` at location 5 (i.e., an intervals analysis).

Symba maintains both an *under*- and an *over-approximation* of the best abstraction φ'. It works by sampling points (models) in φ in a systematic manner, using an SMT solver, and adding the points to the under-approximation in order to extend it. The process continues until the under-approximation is equal to the best abstraction. The key insight underlying Symba is how to carry out the sampling process in an *infinite space* of satisfying assignments, while ensuring convergence to least upper bounds and discovery of unbounded templates.

Symba also maintains an over-approximation of φ', and when it terminates, the two approximations are equivalent. Maintaining an over-approximation allows us to halt Symba at any point and still compute a sound (but possibly not the best) abstraction of

φ. This makes SYMBA resilient to SMT solver failures and resource (e.g., time) depletion.

We have implemented SYMBA and integrated it as an abstract domain in UFO [13], a C analysis and verification framework built in the LLVM compiler infrastructure [14]. We have applied SYMBA to a large number of programs from the software verification competition (SV-COMP) [15], with a total of 2.7MLoC. Standard abstract interpretation techniques failed to prove safety of any of the 373 safe programs, whereas SYMBA produced more precise invariants for 73% of the loops and proved 47 programs safe. In a comparison against other symbolic abstraction techniques, SYMBA demonstrates large efficiency improvements.

1.4 Contributions

We summarize our contributions as follows:

- SYMBA: A novel symbolic abstraction algorithm that exploits the power and precision of SMT solvers for computing precise abstract post over loop-free program fragments. As a result, SYMBA is able to avoid imprecise join operations by symbolically enumerating program paths; and avoid the imprecision of abstract post operations over single instructions and basic blocks. Moreover, SYMBA is parameterized by the abstract domain used, and allows for intervals, octagons, octahedra, as well as others, as defined by the Template Constraint Matrix domain [12].

- An extensive experimental evaluation of SYMBA as (1) an abstract post transformer over loop-free program fragments, (2) as a loop invariant synthesizer, and (3) as a verification algorithm within an abstraction refinement loop. Our results indicate the power and efficiency of SYMBA in comparison with other symbolic abstraction techniques, and its large precision improvements when compared against standard abstract interpretation techniques.

1.5 Organization

In this thesis, we introduce and evaluate our symbolic abstraction algorithm SYMBA. We start by reviewing preliminaries in Chapter 2. In Chapter 3, we demonstrate the operation of SYMBA on simple examples. In Chapter 4, we formalize SYMBA and prove its correctness. In Chapters 5 and 6, we describe our implementation and experimental results. In Chapter 7, we compare SYMBA to related work. We conclude in Chapter 8 with suggestions for future work.

Chapter 2

Background

2.1 Vectors and Matrices

Let $\boldsymbol{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ denote a column vector which is an $n \times 1$ matrix for $n \geqslant 0$. $\mathbf{A}$ denotes a real matrix, while $\mathbf{A}_{i,j}$ represents the entry in the i-th row and the j-th column. The dot product of two column vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$ is defined as $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^T \boldsymbol{b}$ where $\mathbf{A}^T$ is the transpose of $\mathbf{A}$.

Definition 1 (Linear Assertions)**.** *A linear inequality is an expression of the form $a_1x_1 + \cdots + a_nx_n \bowtie b$, where $\bowtie \in \{\geqslant, =\}$ and $a_i, b \in \mathbb{R}$. A linear assertion is a finite conjunction of linear inequalities.*

Matrices can be used to compactly write and work with multiple linear inequalities, i.e., linear assertion. For example, if $\mathbf{A}$ is an $m \times n$ matrix, $\boldsymbol{x}$ designates a column vector of n variables $x_1, x_2, \ldots, x_n$, and $\boldsymbol{b}$ is an $m \times 1$ column vector, then the matrix representation

$$\mathbf{A}\boldsymbol{x} \bowtie \boldsymbol{b}$$

is equivalent to the system of linear inequalities

$$\begin{gathered}\mathbf{A}_{1,1}x_1 + \mathbf{A}_{1,2}x_2 + \cdots + \mathbf{A}_{1,n}x_n \bowtie b_1 \\ \cdots \\ \mathbf{A}_{m,1}x_1 + \mathbf{A}_{m,2}x_2 + \cdots + \mathbf{A}_{m,n}x_n \bowtie b_m.\end{gathered}$$

Equality constraints can be transformed into equivalent inequality constraints. For example, $\alpha_1x_1 + \cdots + \alpha_nx_n = b$ can be written as $\alpha_1x_1 + \cdots + \alpha_nx_n \geqslant b$ in conjunction with $\alpha_1x_1 + \cdots + \alpha_nx_n \leqslant b$. In this thesis, we assume that linear equalities are written as equivalent inequalities.

A *zero matrix* is a matrix with all its entries being zero, denoted by $\mathbf{0}_{i,j}$. Some examples of zero matrices are

$$\mathbf{0}_{1,1} = \begin{bmatrix} 0 \end{bmatrix}, \ \mathbf{0}_{2,2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \ \mathbf{0}_{2,3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

An *identity matrix* of size n is the $n \times n$ square matrix with ones on the main diagonal and zeros elsewhere, denoted by $\mathbf{1}_n$. Some examples of identity matrices are

$$\mathbf{1}_1 = \begin{bmatrix} 1 \end{bmatrix}, \ \mathbf{1}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ \mathbf{1}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Note that we may omit the subscripts for simplicity if the size of the matrices can be easily inferred.

2.2 The Geometry of Polyhedra

Every point x in the n-dimensional Euclidean space $\mathbb{R}^n$ can be represented as a column vector $\boldsymbol{x}$. Let $\{x_1, \ldots, x_k\}$ be a finite set of points in $\mathbb{R}^n$.

Definition 2 (Affine Combinations and Affine Hull). *An affine combination of* $x_1, \ldots, x_k$ *is any point of the form*

$$x = \alpha_1 x_1 + \cdots + \alpha_k x_k,$$

where $\alpha_1, \ldots, \alpha_k$ *are real numbers satisfying*

$$\alpha_1 + \cdots + \alpha_k = 1.$$

The set of all affine combinations of $x_1, \ldots, x_k$ *is known as the affine hull of* $\{x_1, \ldots, x_k\}$.

It can be verified that any affine combination of solutions of a nonhomogeneous system of linear equations is also a solution of this system [16].

Definition 3 (Convex Combinations and Convex Hull). *A convex combination of* $x_1, \ldots, x_k$ *is any point of the form*

$$x = \alpha_1 x_1 + \cdots + \alpha_k x_k,$$

where $\alpha_1, \ldots, \alpha_k$ *are real numbers satisfying*

$$\begin{gathered} \alpha_1 + \cdots + \alpha_k = 1 \\ \alpha_i \geqslant 0 \end{gathered}$$

The set of all convex combinations of $x_1, \ldots, x_k$ is known as the convex hull of $\{x_1, \ldots, x_k\}$.

Definition 4 (Translate)**.** *If $\mathbf{S} \subset \mathbb{R}^n, \check{\boldsymbol{x}} \in \mathbb{R}^n$, the translate of $\mathbf{S}$ to $\check{\boldsymbol{x}}$ is the set $\{\boldsymbol{x} \mid \boldsymbol{x} = \check{\boldsymbol{x}} + \boldsymbol{y}, \boldsymbol{y} \in \mathbf{S}\}$.*

Definition 5 (Rays and Half-Lines)**.** *Let $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$, $\tilde{\boldsymbol{x}} \neq \mathbf{0}$. The ray generated by $\tilde{\boldsymbol{x}}$ is the set $\{\boldsymbol{x} \mid \boldsymbol{x} = \lambda\tilde{\boldsymbol{x}}, \lambda \geqslant 0\}$. If $\check{\boldsymbol{x}} \in \mathbb{R}^n$, then the set $\{\boldsymbol{x} \mid \boldsymbol{x} = \check{\boldsymbol{x}} + \theta\tilde{\boldsymbol{x}}, \theta \geqslant 0\}$ is known as the half-line through $\check{\boldsymbol{x}}$ parallel to the ray generated by $\tilde{\boldsymbol{x}}$.*

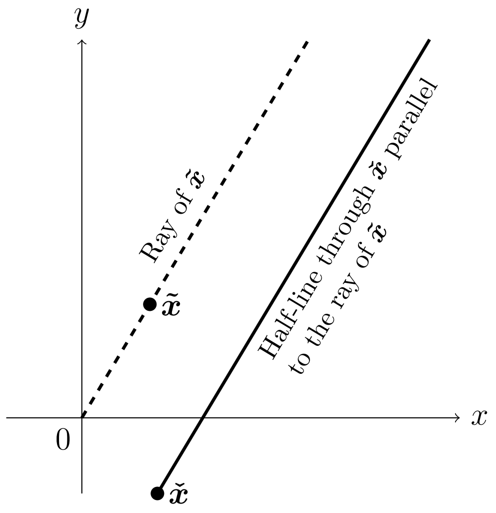
y
Ray of $\tilde{\boldsymbol{x}}$
$\tilde{\boldsymbol{x}}$
Half-line through $\check{\boldsymbol{x}}$ parallel
to the ray of $\tilde{\boldsymbol{x}}$
x
0
$\check{\boldsymbol{x}}$

Figure 2.1: A ray generated by $\tilde{\boldsymbol{x}}$ and a half-line starts at $\check{\boldsymbol{x}}$.

Fig. 2.1 shows a a half-line starting at $\check{\boldsymbol{x}}$ (this point corresponds to $\theta = 0$) which is parallel to the ray generated by $\tilde{\boldsymbol{x}}$. Every ray contains the origin, and a half-line is a translate of a ray.

Definition 6 (Hyperplanes)**.** *A hyperplane in $\mathbb{R}^n$ is the set of all points $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ satisfying a single linear equation, $\alpha_1 x_1 + \cdots + \alpha_n x_n = b$, where α_i, b are given numbers and at least one of the α_i is nonzero, that is, $(\alpha_1, \ldots, \alpha_n) \neq \mathbf{0}$.*

Definition 7 (Half-Spaces)**.** *An open half-space is either of the two open sets produced by the subtraction of a hyperplane from $\mathbb{R}^n$. A closed half-space is the union of an open half-space and the hyperplane that defines it. A closed half-space may be specified as a linear inequality $\alpha_1 x_1 + \cdots + \alpha_n x_n \geqslant b$.*

Definition 8 (Convex Polyhedra and Convex Polytopes)**.** *The intersection of a finite number of half-spaces is known as a convex polyhedral set or a convex polyhedron. A convex polyhedron that is bounded is known as a convex polytope.*

It is clear from the discussion earlier that the solution set of a linear assertion is a closed convex polyhedron.

2.3 Abstract Interpretation

Similar to *data-flow analysis* [17], abstract interpretation [2] is an iterative fixed-point computation based technique. In this section, we briefly recall the framework of abstract interpretation and define some of the terminology and concepts used in this thesis.

2.3.1 Partial Orders and Lattices

Definition 9 (Partial Orders)**.** *A partial order is a mathematical structure $(L, \sqsubseteq)$, where L is a set and $\sqsubseteq$ is a binary relation on L that satisfies the following conditions:*

1. *reflexivity:* $\forall x \in L \cdot x \sqsubseteq x$
2. *transitivity:* $\forall x, y, z \in L \cdot x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
3. *anti-symmetry:* $\forall x, y \in L \cdot x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Let $X \subseteq L$. We say that $y \in L$ is an upper bound for X, written as $X \sqsubseteq y$, if we have $\forall x \in X \cdot x \sqsubseteq y$. Similarly, $y \in L$ is a lower bound for X, written as $y \sqsubseteq X$, if $\forall x \in X \cdot y \sqsubseteq x$. A least upper bound (a.k.a., join), written as $\sqcup X$, is defined by: $X \sqsubseteq \sqcup X \wedge \forall y \in L \cdot X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$. Dually, a greatest lower bound (a.k.a., meet), written as $\sqcap X$, is defined by: $\sqcap X \sqsubseteq X \wedge \forall y \in L \cdot y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$.

Definition 10 (Lattices)**.** *A lattice is a partial order $(L, \sqsubseteq)$ for which a unique $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq L$.*

Notice that a lattice must have a unique largest element $\top$ defined as $\top = \sqcup L$ and a unique smallest element $\bot$ defined as $\bot = \sqcap L$.

2.3.2 Fixed-Points

Let $(L, \preceq)$ and $(\overline{L}, \sqsubseteq)$ be two partially ordered sets. A function $f : L \to \overline{L}$ is *monotone* when $\forall x, y \in L \cdot x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. The existence of the least and greatest fixed-point on a monotonic map is guaranteed by the following theorem.

Theorem 1 (Tarski's Theorem)**.** *Let $(L, \sqsubseteq)$ be any complete lattice. Let $f : L \to L$ be a monotone function on this lattice. Then the set of fix-points is a non-empty complete lattice.*

Consequently, f has a least fixed-point which is defined as $lfp(f) = \bigsqcup_{i \geqslant 0} f^i(\bot)$. Usually, inductive invariants are generated by fixed-point computation.

2.3.3 Galois Connections

Definition 11 (Galois Connections)**.** *Let* $(L, \preceq)$ *and* $(\overline{L}, \sqsubseteq)$ *be two partially ordered sets. A pair* (α, γ) *of maps where* $\alpha : L \to \overline{L}$ *and* $\gamma : \overline{L} \to L$ *is a Galois Connection if and only if*

$$\forall x \in L, \forall y \in \overline{L} \cdot \alpha(x) \sqsubseteq y \Leftrightarrow x \preceq \gamma(y)$$

which is written

$$(L, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\overline{L}, \sqsubseteq).$$

Usually, we call L the concrete set and $\overline{L}$ the abstract set. Similarly, α is called the abstraction function and γ is called the concretization function.

2.3.4 Widening

Definition 12 (Widening Operator)**.** *A widening operator* $\nabla \in \overline{L} \times \overline{L} \to \overline{L}$ *is such that:*

1. *correctness:* $\forall x, y \in \overline{L} \cdot \gamma(x) \sqsubseteq \gamma(x \nabla y) \wedge \gamma(y) \sqsubseteq \gamma(x \nabla y)$,
2. *convergence: for all increasing chains* $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$, *the increasing chain defined by* $y_0 = x_0, \ldots, y_{i+1} = y_i \nabla x_{i+1}, \cdots$ *is not strictly increasing.*

Widening operator can be used to accelerate the convergence of fixed-point computation in domains that have infinite ascending chains such as *intervals* and *polyhedra*.

Chapter 3

Symba by Example

In this chapter, we illustrate the operation of SYMBA on two formulas, a 2-dimensional and a 3-dimensional one.

3.1 A 2-dimensional Example

Consider the formula

$$\varphi \equiv 1 \leqslant y \leqslant 3 \wedge (1 \leqslant x \leqslant 3 \vee x \geqslant 4)$$

containing the real variables x and y and represented pictorially in Fig. 3.1. Suppose that our set of templates is $T = \{y, x+y\}$. That is, we would like to find the least upper bound for y and $x+y$ in φ. (Note that if we want to find a lower bound for y, we can simply add $-y$ as a template to T.)

Initially, the under-approximation, U, of the best abstraction of φ w.r.t. the templates T is *false*, and the over-approximation, O, is *true*. SYMBA alternates between two main operations: GLOBALPUSH, which is used to grow the under-approximation by sampling points (models) of φ that lie outside the under-approximation; and UNBOUNDED, which is used to detect unbounded templates. In this example, 3 is the upper bound for y, and $x+y$ is unbounded. That is, the best abstraction of φ using T is $y \leqslant 3$.

First GlobalPush. SYMBA starts with a GLOBALPUSH by querying an SMT solver for a model of φ that is not a model of U. Suppose the solver returns the point $\mathsf{p_1} = (2, 2)$. The under-approximation U is the best approximation of the set of points found by the solver that is expressible in the TCM domain with the templates T. So, the under-approximation is updated to $U = y \leqslant 2 \wedge x + y \leqslant 4$ (since the maximum values of y and $x+y$ seen so far are 2 and 4, respectively). This is shown as the shaded region U_1 in

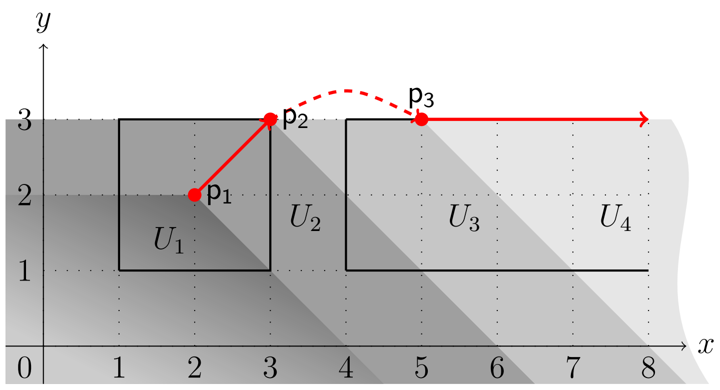

Figure 3.1: Illustration of SYMBA on a 2-D example.

Fig. 3.1.

Unbounded (p_1, y). SYMBA now tries to prove that y is unbounded. First, we categorize points into equivalence classes as follows: Let $\mathcal{E}(\varphi) = \{t = k \mid t \leqslant k \in \varphi\}$, i.e., $\mathcal{E}(\varphi)$ is the set of all atomic formulas appearing in φ with the inequalities replaced by equalities. In our example, $\mathcal{E}(\varphi) = \{x = 1, x = 3, x = 4, y = 1, y = 3\}$. Informally, $\mathcal{E}(\varphi)$ represents the set of edges (boundaries) appearing in Fig. 3.1. The equivalence class $[\mathsf{p}]$ of a point p is $\{e \in \mathcal{E}(\varphi) \mid \mathsf{p} \models e\}$, i.e., the set of equalities in $\mathcal{E}(\varphi)$ satisfied by p. For p_1, $[\mathsf{p}_1] = \{\}$, since p_1 does not lie on any of the boundaries. The intuition underlying UNBOUNDED is finding a ray r from a point p in φ such that a given template t is increasing along r, and r never hits any boundaries of φ (i.e., completely contained in φ).

UNBOUNDED picks a point p_1 and queries the SMT solver for a point p' s.t. $[\mathsf{p}'] = [\mathsf{p}_1]$ and $y(\mathsf{p}_1) < y(\mathsf{p}')$, where $y(\mathsf{p}')$ is the valuation of y at point p'. The point $\mathsf{p}' = (2, 2.5)$ satisfies this condition. Then UNBOUNDED queries for a point p'' s.t. $[\mathsf{p}'] \subset [\mathsf{p}'']$ and $y(\mathsf{p}') \leqslant y(\mathsf{p}'')$. If no such p'' exists, then we know that y is unbounded. Intuitively, we are asking whether we can keep increasing the value of y from p' without *touching* a point p'' on one of the boundaries in $\mathcal{E}(\varphi)$. In this case, such a p'' exists, so it is added to the under-approximation as $\mathsf{p}_2 = (3, 3)$ in Fig. 3.1. Note that p_2 exhibits the upper bound of y; SYMBA detects that and updates the over-approximation O to $y \leqslant 3$. To take p_2 into account, U is updated to become $y \leqslant 3 \wedge x + y \leqslant 6$ (see region U_2 in the figure).

Second GlobalPush. Suppose that SYMBA calls GLOBALPUSH. The result is a point in φ but not in U. Let $\mathsf{p}_3 = (5, 3)$ be the point found by GLOBALPUSH. As a result, U is updated to $y \leqslant 3 \wedge x + y \leqslant 8$ (see region U_3).

Unbounded ($\mathsf{p}_3, x+y$). SYMBA now applies UNBOUNDED to check if $x+y$ is unbounded. Suppose UNBOUNDED picks the point p_3. First, it finds a point $\mathsf{p}' = (6, 3)$ which increases $x + y$ and is in the same equivalence class as p_3. Then, it tries to find a point p'' that

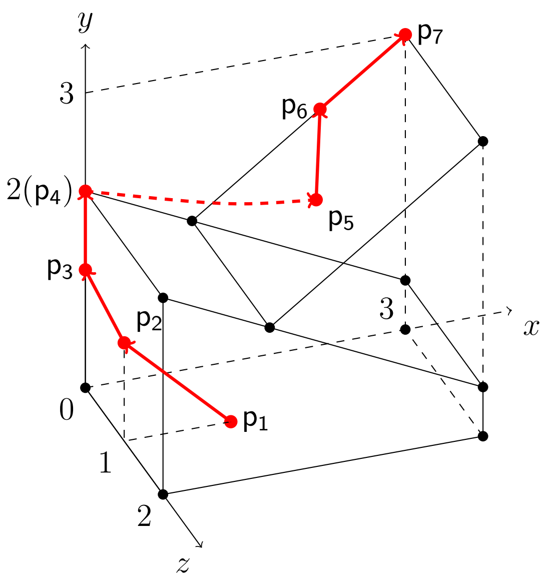

Figure 3.2: Illustration of SYMBA on a 3-D example.

has an equivalence class $[\mathsf{p}'] \subset [\mathsf{p}'']$ and has a greater (or equal) valuation of $x+y$ than p'. Since no such point p'' exists, SYMBA concludes that $x+y$ is unbounded. Intuitively, SYMBA discovers that it is possible to keep finding points, along the boundary $y=3$, that increase $x+y$ without encountering any other boundary, thus concluding that $x+y$ is unbounded. We formally specify and prove the correctness of UNBOUNDED in Section 4.

The under-approximation U is now updated to become $y \leqslant 3$ (region U_4), by dropping the upper bound for $x+y$. At this point, $U=O$, so SYMBA terminates with the best abstraction $y \leqslant 3$.

3.2 A 3-dimensional Example

We now illustrate the workings of SYMBA on the formula

$$\varphi \equiv 0 \leqslant x \leqslant 3 \wedge 0 \leqslant z \leqslant 2 \wedge (2y \leqslant -x+4 \vee 4y = 3x+3),$$

containing the variables x, y, and z and depicted in Fig. 3.2. Suppose, for simplicity, that we would like to find the least upper bound only for y, i.e., $T=\{y\}$.

First GlobalPush. Similar to our previous example, SYMBA starts with $U = \textit{false}$ and $O = \textit{true}$ and uses GLOBALPUSH to find the initial point. Suppose the SMT solver returns the point $\mathsf{p}_1 = (1,0,1)$ denoting values of (x,y,z). Thus, $U = y \leqslant 0$.

Unbounded (p_1, y). To check if y is unbounded, SYMBA applies UNBOUNDED starting from p_1. Since it cannot prove that y is unbounded, it finds the point $\mathsf{p}_2 = (0, 1, 1)$, where $[\mathsf{p}_1] \subset [\mathsf{p}_2]$ and $y(\mathsf{p}_1) < y(\mathsf{p}_2)$, i.e., a point showing that increasing the value of y from p_1 can hit a boundary. After applying Unbounded to p_2, Symba can get the point p_3, and then point p_4 (after applying Unbounded to p_3). As a result, $U = y \leqslant 2$. From point p_4, Symba cannot apply Unbounded, since there does not exist a point p' where $[\mathsf{p}'] = [\mathsf{p}_4]$ that increases the value of y. Intuitively, p_4 represents a local maximum.

Second GlobalPush. To escape the local maximum, Symba uses GlobalPush to query the SMT solver for a point outside U. In this case, it might find the point $\mathsf{p}_5 = (1.8, 2.1, 1)$, and thus U becomes $y \leqslant 2.1$.

Unbounded (p_5, y). Symba continues trying to prove that y is unbounded by performing Unbounded from p_5, leading to p_6 and then p_7. Symba detects that p_7 represents the maximum value of y in φ and terminates with the best abstraction $y \leqslant 3$.

We have illustrated the workings of Symba on two formulas representing non-convex shapes, and showed how it utilizes an SMT solver to find least upper bounds and detect unboundedness of arbitrary linear expressions (expressed as templates). In the following chapters, we describe Symba formally and discuss our implementation and experimental results.

Chapter 4

Symba: The Symbolic Abstraction Algorithm

In this chapter, we provide definitions required for the rest of the thesis and formalize Symba as a set of inference rules.

4.1 Definitions

4.1.1 Formulas

Let $\mathcal{L}$ be a a topologically-closed (i.e., all atoms are non-strict inequalities) subset of Quantifier Free Linear Real Arithmetic (QF_LRA), defined as follows:

$$\begin{aligned} \varphi \in \mathcal{L} &::= \mathit{true} \mid \mathit{false} \mid P \wedge P' \mid P \vee P' \\ P, P' \in \mathit{Atoms} &::= c_1x_1 + \cdots + c_nx_n \leqslant k, n \in \mathbb{N} \\ x_i \in \mathit{Vars} &::= \{x_1, \ldots, x_n\}, \end{aligned}$$

where $c_i, k \in \mathbb{R}$.

We use $[\![\varphi]\!]$ to denote the set of all satisfying assignments (models) of φ. A *model* $p : \mathit{Vars} \to \mathbb{R}$ of φ, denoted $p \models \varphi$, is a valuation of the variables of φ such that $\varphi(p) \equiv \mathit{true}$, where $\varphi(p)$ is φ with every occurrence of a variable x replaced by $p(x)$. Geometrically, p is a point in $\mathbb{R}^n$, and in what follows, we use the terms *model* and *point* to refer to p interchangeably. We use $\mathit{Atoms}(\varphi)$ to denote the set of all *Atoms* appearing in φ.

4.1.2 Equivalence Classes

We define the *equivalence class* $[p]$ of a model p of φ as follows:

$$[p] = \{a \in \mathcal{E}(\varphi) \mid p \models a\}$$

where $\mathcal{E}(\varphi) = \{l = k \mid l \leqslant k \in \mathit{Atoms}(\varphi)\}$, and l is a linear expression of the form $c_1x_1 + \cdots + c_nx_n$. For clarity, we often abuse notation and use $[p]$ to denote the formula $\bigwedge[p]$ (i.e., conjunction of all formulas in the set $[p]$). Geometrically, $\mathcal{E}(\varphi)$ is the set of hyperplanes that form the boundary between points for which an atom does and does not hold, and $[p]$ is the intersection of the hyperplanes that p falls on.

4.1.3 The Template Constraint Matrix (TCM) Domain

The TCM domain [12] is a numerical abstract domain parameterized by a set of linear terms T called *templates* of the form $c_1x_1 + \cdots + c_nx_n$, where $c_i \in \mathbb{R}$, and $\mathit{Vars} = \{x_1, \ldots, x_n\}$ are real-valued variables. Throughout the thesis, we assume the set Vars is fixed. A set of templates T gives rise to the domain $\mathcal{A}_T$ where each $A \in \mathcal{A}_T$ is a vector of constants $\boldsymbol{k} = (k_1, \ldots, k_{|T|})$, with $k_i \in \mathbb{R} \cup \{\infty, -\infty\}$. In other words, the domain consists of polyhedra of a fixed shape determined by T. For example, suppose $T = \{2x + y, 2z\}$. Then an abstract state $(0, \infty) \in \mathcal{A}_T$ represents the formula $2x + y \leqslant 0 \wedge 2z \leqslant \infty$. By simplifying $2z \leqslant \infty$ to *true*, we get the formula $2x + y \leqslant 0$ in $\mathcal{L}$.

Without loss of generality, we assume that $T \neq \emptyset$. The top $(\top)$ and bottom $(\bot)$ elements of $\mathcal{A}_T$ are the vectors $(\infty, \ldots, \infty)$ and $(-\infty, \ldots, -\infty)$, respectively. We use $\sqcup$ and $\sqsubseteq$ to denote the join and the order operations in the TCM domain, respectively.

4.1.4 Symbolic Abstraction Over TCM domain

We now formalize the relationship between TCMs and $\mathcal{L}$ formulas. Let $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}_T$ be a Galois connection between the concrete domain $\mathcal{C}$ and the abstract domain $\mathcal{A}_T$. We use the notation of [18, 19] to define *symbolic versions* of the *abstraction* and *concretization* functions α and γ, namely, $\widehat{\alpha}$ and $\widehat{\gamma}$, between the domain representing $\mathcal{L}$ formulas and $\mathcal{A}_T$.

- $\widehat{\alpha} : \mathcal{L} \to \mathcal{A}_T$ maps a formula $\varphi \in \mathcal{L}$ to the best value $A \in \mathcal{A}_T$ such that $[\![\varphi]\!] \subseteq \gamma(A)$. Specifically,
$$\widehat{\alpha}(\varphi) = (k_1, \ldots, k_{|T|}),$$

where $\varphi' \equiv t_1 \leqslant k_1 \wedge \cdots \wedge t_{|T|} \leqslant k_{|T|}$ is the strongest formula such that $\varphi \Rightarrow \varphi'$, and $t \leqslant \infty \equiv \mathit{true}$ and $t \leqslant -\infty \equiv \mathit{false}$ for any template $t \in T$.

- $\widehat{\gamma} : \mathcal{A}_T \to \mathcal{L}$ maps an element $A \in \mathcal{A}_T$ to the formula $\varphi \in \mathcal{L}$ such that $\gamma(A) \subseteq [\![\varphi]\!]$. Specifically,

$$\widehat{\gamma}(A) = \begin{cases} \mathit{false}, & \exists k_i \in A \cdot k_i = -\infty \\ \bigwedge\{t_i \leqslant k_i \mid k_i \neq \infty\}, & \text{otherwise} \end{cases}$$

where $A = (k_1, \ldots, k_{|T|})$.

The goal of Symba is to compute $\widehat{\alpha}(\varphi)$, the best abstraction of a $\mathcal{L}$ formula in some TCM domain $\mathcal{A}_T$. Abstract post computations can be reduced to $\widehat{\alpha}$ [20, 18, 21], using which we can compute best abstract transformers for $\tau : \mathcal{L} \to \mathcal{L}$ described symbolically and conveniently as formulas.

4.2 Symba Formalized

$$\frac{}{\langle \emptyset, \bot, \top \rangle}\ \textsc{Init}$$

$$\frac{p \models \varphi \wedge \neg\widehat{\gamma}(U)}{\langle M, U, O\rangle \to \langle M \cup \{p\}, U \sqcup \widehat{\alpha}(p), O\rangle}\ \textsc{GlobalPush}$$

$$\frac{\begin{array}{c} U = (k_1, \ldots, k_n) \quad p_2 \models \varphi \quad [p_2] = [p_1] \quad t_i(p_1) < t_i(p_2) \\ \nexists p_3 \models \varphi \wedge t_i(p_2) \leqslant t_i(p_3) \wedge [p_2] \subset [p_3] \end{array}}{\langle M, U, O\rangle \to \langle M, U \sqcup (k_1, \ldots, k_{i-1}, \infty, k_{i+1}, \ldots, k_n), O\rangle}\ \textsc{Unbounded}(p_1 \in M, t_i \in T)$$

$$\frac{p_2, p_3 \models \varphi \quad t_i(p_1) < t_i(p_2) \leqslant t_i(p_3) \quad [p_1] = [p_2] \subset [p_3]}{\langle M, U, O\rangle \to \langle M \cup \{p_3\}, U \sqcup \widehat{\alpha}(p_3), O\rangle}\ \textsc{Unbounded-Fail}(p_1 \in M, t_i \in T)$$

$$\frac{O = (k_1, \ldots, k_n) \quad m = \max\{t_i(p') \mid p' \in M\} \quad \varphi \Rightarrow t_i \leqslant m}{\langle M, U, O\rangle \to \langle M, U, O \sqcap (k_1, \ldots, k_{i-1}, m, k_{i+1}, \ldots, k_n)\rangle}\ \textsc{Bounded}(t_i \in T)$$

Figure 4.1: Inference rules used by Symba.

We now formalize the symbolic abstraction algorithm Symba as a set of inference rules shown in Fig. 4.1.

Given a set of templates $T = \{t_1, \ldots, t_n\}$ and a formula φ in $\mathcal{L}$, Symba computes $\widehat{\alpha}(\varphi)$ in the TCM domain $\mathcal{A}_T$. The *state* of Symba is a tuple $\langle M, U, O \rangle$, where M is a set of models of φ; U is an under-approximation of $\widehat{\alpha}(\varphi)$ (i.e., $U \sqsubseteq \widehat{\alpha}(\varphi)$ is an invariant); and O is an over-approximation of $\widehat{\alpha}(\varphi)$ (i.e., $\widehat{\alpha}(\varphi) \sqsubseteq O$ is an invariant).

When Symba terminates, we know that $U = O = \widehat{\alpha}(\varphi)$. Initially, as defined by the rule Init, $M = \emptyset$, $U = \bot$, and $O = \top$. The rules GlobalPush, Unbounded, and Unbounded-Fail are used to weaken U until it is equal to $\widehat{\alpha}(\varphi)$, whereas Bounded strengthens O until it is equal to $\widehat{\alpha}(\varphi)$. Given a model p, we use $\widehat{\alpha}(p)$ to denote the best abstraction of p in $\mathcal{A}_T$, i.e., $(t_1 \leqslant t_1(p), \ldots, t_n \leqslant t_n(p))$.

GlobalPush finds a model of φ that is not captured by $\widehat{\gamma}(U)$ (i.e., lies outside the under-approximation) and adds it to U to weaken it. When the rule GlobalPush no longer applies, we know that $U = \widehat{\alpha}(\varphi)$. Note that applying this rule alone does not guarantee that U eventually reaches $\widehat{\alpha}(\varphi)$ for two reasons:

1. Since we are dealing with real variables, GlobalPush might keep finding models that approach the upper bound of one of the templates asymptotically.

2. GlobalPush cannot detect whether a template t is unbounded, as it will keep finding models p that increase the value $t(p)$, thus growing the under-approximation indefinitely.

To that end, the rules Unbounded and Unbounded-Fail are used to detect unbounded templates and help GlobalPush avoid asymptotic behavior. Unbounded takes as parameters a model $p_1 \in M$ and a template $t_i \in T$ and attempts to prove that t_i is unbounded as follows: First, it tries to find a point $p_2 \models \varphi$ such that $[p_1] = [p_2]$ and $t(p_1) < t(p_2)$. Then, it looks for a point p_3 such that $p_3 \models \varphi$, $[p_1] = [p_2] \subset [p_3]$ and $t(p_2) \leqslant t(p_3)$. If no such p_3 exists, then t is unbounded in φ. Otherwise, Unbounded-Fail adds p_3 to M. The intuition here is as follows: If we can find a model p_2, then we know that t can increase along the hyperplanes in $\mathcal{E}(\varphi)$. If no point p_3 exists, then we know that we can keep increasing t indefinitely without encountering any of the boundaries in $\mathcal{E}(\varphi)$ that are not in $[p_2]$, thus showing that t is unbounded. This is analogous to the technique used by the simplex method for showing that a dimension is unbounded in a convex polyhedron. We further discuss the intuition underlying Unbounded and prove its correctness in Sec. 4.3.

In addition to the aforementioned rules, the rule Bounded detects whether a model $p \in M$ exhibits the largest value for some template t, i.e., $\varphi \Rightarrow t \leqslant t(p)$, and strengthens the over-approximation accordingly. Note that the over-approximation is not required for the correctness of SYMBA, but its availability allows us to guarantee that SYMBA maintains a sound approximation O of $\widehat{\alpha}(\varphi)$ at every point of its execution. This makes SYMBA resilient to SMT solver failures and allows us to limit resource consumption when desired.

Example. We illustrate the applications of the rules on the 2-D example from Sec. 3.1 and shown in Fig. 3.1. Assume that after the initial call to GLOBALPUSH, $M = \{\mathsf{p}_1 = (2, 2)\}$, $\widehat{\gamma}(U) = y \leqslant 2 \wedge x + y \leqslant 4$, and $\widehat{\gamma}(O) = \mathit{true}$.

Applying UNBOUNDED-FAIL on $\mathsf{p}_1 \in M$ and $y \in T$ adds $\mathsf{p}_2 = (3, 3)$ to M. Next, BOUNDED is used to detect that p_2 exhibits an upper bound of y, and updates O so that $\widehat{\gamma}(O) \equiv y \leqslant 3$.

Assume that the second application of GLOBALPUSH adds point $\mathsf{p}_3 = (5, 3)$ to M. Applying UNBOUNDED$(\mathsf{p}_3, x + y)$ detects that $x + y$ is unbounded. At this point, $\widehat{\gamma}(U)$ becomes $y \leqslant 3$, making GLOBALPUSH inapplicable. Therefore, $\varphi \Rightarrow \widehat{\gamma}(U)$.

In what follows, we discuss and prove soundness of SYMBA, and define terminating sequences of rule application.

4.3 Soundness

We start by showing soundness of the UNBOUNDED rule.

A necessary and sufficient condition for proving that a given template t is unbounded within φ is the existence of a convex polyhedron φ_c, e.g., a half-line, such that t is unbounded in φ_c and $\varphi_c \Rightarrow \varphi$. Our solution addresses two problems:

1. How to restrict the space from which φ_c is drawn while maintaining completeness, i.e., ensuring that φ_c is found whenever t is unbounded in φ.

2. How to check that $\varphi_c \Rightarrow \varphi$.

The idea we use here is to restrict φ_c to formulas of the form

$$\bigwedge E \wedge t \geqslant k,$$

where $E \subseteq \mathcal{E}(\varphi)$ and $k \in \mathbb{R}$. This space of convex polyhedra is sufficient for completeness. For instance, consider the example from Fig. 3.1. To prove that the $x + y$ direction is unbounded, we find a point $\mathsf{p}_3 = (5, 3)$ that lies on the boundary $y = 3 \in \mathcal{E}(\varphi)$ and ask whether $\varphi_c \equiv y = 3 \wedge x + y \geqslant 8$ is contained in φ. Furthermore, we perform the containment check implicitly by checking whether there is a point in φ_c, along any direction that increases $x + y$, that intersects a boundary of φ. In our running example, such a point does not exist (see Fig. 3.1). Thus, $x+y$ is unbounded. For another example, consider the point $\mathsf{p_1} = (2, 2)$. Since $\mathsf{p_1}$ does not lie on any boundary, to check if $x + y$ is unbounded we ask whether $\varphi_c \equiv x + y \geqslant 4$ is contained in φ (i.e., we check whether increasing $x + y$ in φ_c does not encounter boundaries in φ). This is not the case, and the counterexample is the point $\mathsf{p_2}$, shown in Fig. 3.1, that lies on the boundaries $x = 3$ and $y = 3$.

Thm. 2 formalizes this construction using equivalence classes and states its correctness for proving that a template is unbounded in φ. To proceed with the proof, we first show that the following lemma holds.

Lemma 1. *Given a $\mathcal{L}$ formula φ_c defining a convex polyhedron (i.e., conjunction of linear inequalities), if $p \models \varphi_c$, $[p] \subseteq [p']$, $t(p) \leqslant t(p')$ and $\nexists p'' \models \varphi_c \cdot t(p) \leqslant t(p'') \wedge [p] \subset [p'']$, then $p' \models \varphi_c$.*

Proof. Formula $\varphi_c \wedge [p]$ can be written as a system of linear inequalities as follows:

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \tag{4.1a}$$

$$\mathbf{C}\boldsymbol{x} > \boldsymbol{d} \tag{4.1b}$$

By definition, p satisfies Eq. 4.1 and $[p]$ corresponds to Eq. 4.1a. After introducing slack variables corresponding to the inequality constraints, we can transform inequality constraints into equality constraints, i.e., $\sum_{j=1}^{n} c_{ij}x_j > d_i$ becomes $\sum_{j=1}^{n} c_{ij}x_j - s_i = d_i$.

$$\mathbf{A}\boldsymbol{x} + \mathbf{0}\boldsymbol{s} = \boldsymbol{b} \tag{4.2a}$$

$$\mathbf{C}\boldsymbol{x} - \mathbf{1}\boldsymbol{s} = \boldsymbol{d} \tag{4.2b}$$

$$\boldsymbol{s} > \mathbf{0} \tag{4.2c}$$

Here $\boldsymbol{s} = \mathbf{C}\boldsymbol{x} - \boldsymbol{d}$ is the vector of slack variables. Because $[p] \subseteq [p']$, p' must satisfy $[p]$ (Eq. 4.2a). Eq. 4.2b is always satisfied by definition. Suppose $p' \nvDash \psi_c$, then there exists some k such that $s_k > 0$ for p and $s_k \leqslant 0$ for p', which is equivalent to

$$\boldsymbol{c_k} \cdot \boldsymbol{p} - d_k > 0$$
$$\boldsymbol{c_k} \cdot \boldsymbol{p'} - d_k \leqslant 0$$

We now show that there exists a point $\boldsymbol{p''} = \alpha\boldsymbol{p} + (1-\alpha)\boldsymbol{p'}$ $(0 \leqslant \alpha < 1)$ in the convex hull of p and p' such that $\boldsymbol{c_k} \cdot \boldsymbol{p''} - d_k = 0$. Since $t(p) \leqslant t(p')$, it is easy to show that $t(p) \leqslant t(p'')$. Let $\boldsymbol{c_k} \cdot \boldsymbol{p} = d_k + \delta_1$, $\boldsymbol{c_k} \cdot \boldsymbol{p'} = d_k - \delta_2$ $(\delta_1 > 0,\ \delta_2 \geqslant 0)$, $\alpha = \frac{\delta_2}{\delta_1+\delta_2}$, we have $\boldsymbol{c_k} \cdot \boldsymbol{p''} = d_k$. This contradicts with the condition, as $\{\boldsymbol{c_k} \cdot \boldsymbol{x} = d_k\} \notin [p] \wedge \{\boldsymbol{c_k} \cdot \boldsymbol{x} = d_k\} \in [p'']$, i.e., $[p] \subset [p'']$. Therefore, $p' \models \psi_c$. $\square$

Theorem 2 (Soundness of UNBOUNDED). *Given a formula φ in logic $\mathcal{L}$ and a linear expression t over the variables of φ, then $\nexists k \in \mathbb{R} \cdot \varphi \Rightarrow t \leqslant k$ (i.e., t is unbounded) if and only if there exist $p_1, p_2 \models \varphi$ such that*

1. $t(p_1) < t(p_2)$
2. $[p_1] = [p_2]$
3. $\nexists p_3 \models \varphi \cdot t(p_2) \leqslant t(p_3) \wedge [p_2] \subset [p_3]$

Proof. $(\Leftarrow)$ We prove this direction by contradiction. First, let $p_1, p_2 \models \varphi$ be two models satisfying the three conditions of the theorem. Suppose there is a point $p^* \models \varphi$ such that $t(p^*)$ is the upper bound for t in φ. We show that there is always a point $p_2' \models \varphi$ such that $t(p_2') > t(p^*)$.

Pick a point p_2' such that $\boldsymbol{p_2'} = \boldsymbol{p_2} + \lambda(\boldsymbol{p_2} - \boldsymbol{p_1})$. It follows that $t(p_2') > t(p^*)$ when $\lambda > \frac{(\boldsymbol{p^*}-\boldsymbol{p_2})\cdot \boldsymbol{t}}{(\boldsymbol{p_2}-\boldsymbol{p_1})\cdot \boldsymbol{t}}$. The notation $\boldsymbol{p}$ denotes the vector representation $(p(x_1), \ldots, p(x_n))$ of the model $p : \mathit{Vars} \rightarrow \mathbb{R}$, where $\mathit{Vars} = \{x_1, \ldots, x_n\}$.

Let the formula φ' define a convex polyhedron such that $p_2 \models \varphi'$ and $\varphi' \Rightarrow \varphi$. Let $\varphi_c \equiv \psi' \wedge \bigwedge[p_2]$. We know that:

1. ψ_c defines convex polyhedron (by its definition).
2. $\nexists p_2'' \models \varphi_c \cdot t(p_2) \leqslant t(p_2'') \wedge [p_2] \subset [p_2'']$ (by condition 3 of the theorem).
3. $[p_2] \subseteq [p_2']$ (since p_2' is in the affine set of p_1 and p_2).
4. $t(p_2) \leqslant t(p_2')$ (since $\lambda \geqslant 0$).

Following the result of Lemma 1, p_2' is in φ_c which is also in φ. This contradicts the assumption that $t(p^*)$ is the least upper bound for t. Therefore, term t is unbounded in φ.

$(\Rightarrow)$ Given that t is unbounded in φ, we look for two models $p_1, p_2 \models \varphi$ that satisfy the required conditions. Pick $p_1, p_2 \models \varphi$ such that $[p_1] = [p_2]$, $t(p_2) > t(p_1)$, t is unbounded in $\varphi \wedge \bigwedge[p_1]$, and there does not exist an equivalence class $c \supset [p_1]$ such that t is unbounded in $\varphi \wedge \bigwedge c$. We know that such a class exists because t is unbounded in φ.

If there are no classes $c \supset [p_1]$, or for every $c \supset [p_1]$ we have $\varphi \wedge \bigwedge c \Rightarrow \mathit{false}$, then p_1 and p_2 satisfy the three conditions of the theorem. Otherwise, let $m = max_{p \models \varphi \wedge \psi} t(p)$, where $\psi \equiv \bigvee_{c \supset [p_1]} \bigwedge c$ (i.e., all classes larger than $[p_1]$). We know that m is defined (i.e., not ∞) because of our assumption on the class $[p_1]$. If $m < t(p_2)$, then p_1 and p_2 satisfy the three conditions of the theorem. Otherwise, since t is unbounded in $\varphi \wedge \bigwedge [p_1]$, we can find two models $p'_1, p'_2 \models \varphi$ such that $m < t(p'_1) < t(p'_2)$ and $[p_1] = [p'_1] = [p'_2]$. As a result, p'_1 and p'_2 satisfy the three conditions in the theorem. $\square$

In other words, if the Unbounded rule was applied, then $\varphi_c \equiv [p_2] \wedge t \geqslant t(p_2)$ is contained in φ. In the theorem, conditions 1 and 2 imply that t is unbounded within φ_c, and condition 3 implies that increasing t in φ_c does not encounter any boundaries of φ, i.e., $[p_2] \wedge t \geqslant t(p_2)$ is subsumed by φ. It follows from this theorem that Unbounded maintains the invariant $U \sqsubseteq \widehat{\alpha}(\varphi)$, since the best abstraction cannot have a least upper bound for t if it is unbounded.

Theorem 3 (Soundness of Symba)**.** *If* GlobalPush *does not apply, i.e.,* $\varphi \wedge \neg\widehat{\gamma}(U) \Rightarrow \mathit{false}$*, then* $U = \widehat{\alpha}(\varphi)$*.*

Proof. (sketch) Follows trivially from the invariant $U \sqsubseteq \widehat{\alpha}(\varphi)$. $\square$

4.4 Termination

We now discuss sufficient conditions for ensuring termination of Symba. For simplicity of presentation, we assume that T contains a single template t. We start by defining a *fairness* condition on the scheduling of Symba's rules that ensures termination.

A *fair scheduling* is an infinite sequence of actions $a_1, a_2, \ldots$, where

$$a_i \in \{\textsc{GlobalPush}, \textsc{Unbounded}, \textsc{Unbounded-Fail}\},$$

and the following conditions apply:

1. GlobalPush appears infinitely often, and
2. if a point p is added to M along the execution sequence, then both Unbounded(p, t) and Unbounded-Fail(p, t) eventually appear.

Condition 1 ensures that Symba does not get stuck in local maxima. Condition 2 ensures that we visit every local maximum by visiting every equivalence class, thus guaranteeing that either the least upper bound of t is found or it is proved unbounded.

Recall the 3-D example from Sec. 3, where $T = \{y\}$. Suppose our execution only applies the GLOBALPUSH rule. Then U might grow asymptotically towards the least upper bound of y, e.g., $y \leqslant 2, y \leqslant 2.1, y \leqslant 2.11$, etc., never reaching $y \leqslant 3$. Condition 2 forces computing models that lie on one or more of the boundaries $\mathcal{E}(\varphi)$, thus avoiding this asymptotic behaviour. But applying UNBOUNDED and UNBOUNDED-FAIL alone without applying GLOBALPUSH might get us stuck in local maxima. For example, on point $\mathsf{p_4}$ in Fig. 3.2, UNBOUNDED(-FAIL) are inapplicable. Condition 1 ensures that we eventually find a model outside the current under-approximation (see $\mathsf{p_5}$), thus escaping the local maximum.

A *k-sequence* for a template t is a sequence of points $p_1, \ldots, p_k$, where $\forall i \geqslant 1 \cdot p_i \models \varphi \wedge ([p_i] \subset [p_{i+1}]) \wedge t(p_i) \leqslant t(p_{i+1})$, and UNOUNDED-FAIL$(p_k, t)$ fails to apply. For example, in Fig. 3.2, $\mathsf{p_1}, \mathsf{p_2}, \mathsf{p_3}, \mathsf{p_4}$ is a $k-$sequence.

Since there are at most $2^{|Atoms(\varphi)|}$ equivalence classes, a k-sequence is of length at most $k = 2^{|Atoms(\varphi)|}$. Lemma 2 states that the last model p of a k-sequence always exhibits the largest value of t in its equivalence class $[p]$.

Lemma 2. *Let φ be a formula, and t be a template bounded in φ. Then, in every execution of* SYMBA*, the last element p in every k-sequence for t satisfies $t(p) = \max\{t(p_i) \mid p_i \models \varphi \wedge [p]\}$.*

Proof. According to the definition of a k-sequence, UNBOUNDED-FAIL(p, t) does not apply. Since t is bounded, there does not exist $p' \models \varphi \wedge [p]$ such that $t(p) < t(p')$. Thus, $t(p) = \max\{t(p_i) \mid p_i \models \varphi \wedge [p]\}$. □

We are now ready to prove termination of any fair execution of SYMBA. We assume that SYMBA terminates when GLOBALPUSH is no longer applicable, i.e., $\varphi \Rightarrow \widehat{\gamma}(U)$.

Theorem 4. SYMBA *terminates after a finite number of actions in any fair execution.*

Proof. We split the proof into two cases as follows:

Case 1: t is bounded within φ. Suppose SYMBA is non-terminating. Then, in any fair scheduling, there are infinitely many k-sequences. Following Lemma 2, there are infinitely many models p in the execution sequence such that $p \models \varphi$ and $t(p) = \max\{t(p_i) \mid p_i \models \varphi \wedge [p]\}$. We denote the set of such points by P. In any fair execution, GLOBALPUSH must appear after p is added to M. Therefore, there exists a point $p' \in P$ such that $t(p) < t(p')$. As a result, there is a sequence of points $p_1, p_2, \ldots$ in P such $t(p_1) < t(p_2) < t(p_3) < \cdots$. Hence, $\forall i, j \cdot i \neq j \Rightarrow [p_i] \neq [p_j]$. Since the number of equivalence classes is finite, SYMBA eventually finds the least upper bound of t and terminates.

Case 2: t is unbounded. Using the same argument as above, Symba eventually finds a point in an unbounded equivalence class (due to the finite number of equivalence classes) such that the three conditions in Thm. 2 hold. After that, GlobalPush becomes inapplicable. □

Chapter 5

Implementation

5.1 Tools and Technologies

Various tools and technologies have been used in the implementation of Symba. We start by reviewing some of them in this section.

5.1.1 LLVM

LLVM [14] is a compiler infrastructure developed by the University of Illinois with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. LLVM has a collection of core libraries which provide a modern source- and target-independent optimizer. These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR").

LLVM is a well documented framework designed with high modularity. It is easily extensible to build tools of various purposes, such as virtual machines and program analyzers. There are also a number of sub-projects based on it including the C language family frontend Clang [22].

5.1.2 Z3

Z3 [23] is a high-performance theorem prover being developed at Microsoft Research. It is targeted at solving problems that arise in software verification and software analysis. Z3 integrates support for a variety of theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. It can be used to prove theorems and find counter-examples for non-theorems. Z3 has been used in model checking and testing tools including Boogie [24], Pex [25], Yogi [26], etc.

5.1.3 APRON

APRON numerical abstract domain library [27] is designed for the static analysis of the numerical variables of a program by Abstract Interpretation. The APRON library is intended to be a common interface to various underlying libraries as well as abstract domains and to provide additional services that can be implemented independently. The latest version of APRON supports abstract domains including intervals, octagons, convex polyhedra and linear equalities. The library finds its applications both in compilation and optimization as well as verification and debugging.

5.1.4 UFO

UFO [13] is a framework and a tool developed by University of Toronto for verifying and falsifying safety properties of sequential C programs. The framework is built on top of LLVM and is targeted at researchers designing and experimenting with verification algorithms. It allows definition of different abstract post operators, refinement strategies and exploration strategies.

UFO comes with a number of instantiations ranging from predicate abstraction-based over-approximation driven (OD) to interpolation-based under-approximation driven (UD) and several combined OD/UD instantiations that use different forms of predicate abstraction to augment and strengthen interpolation-based analysis. We have implemented Symba as an abstract domain in UFO. Details will be given in the following sections.

5.2 Implementation

We have implemented Symba in C++, using the Z3 SMT solver for satisfiability queries. Our implementation accepts a formula φ and a set of templates T in the standard SMT-LIB2 [28] format (more details will be given in Sec. 6.1). It then computes the best abstraction $\widehat{\alpha}(\varphi)$ and returns the result.

5.2.1 Scheduling Policy

The policy is shown in Fig. 5.1. We maintain two vectors, namely, U and O, that represent the under- and over-approximation of $\widehat{\alpha}(\varphi)$. In contrast to the declarative rules in Fig. 4.1, we do not have a set of all visited models M explicitly. Instead, we

```
1: $\forall t \in T \cdot U[t] \leftarrow -\infty, O[t] \leftarrow \infty, L(t) \leftarrow \emptyset$
2: $Q \leftarrow \emptyset$
3:
4: function Join($p, T$)
5:     for all $t \in T$ such that $t(p) > U[t]$ do
6:         PushListAdd($L(t)$, $[p]$)
7:         $U[t] \leftarrow t(p)$
8:         Q.push($L(t)$)
9: function SymbaMain($\varphi, T$)
10:     while ($p \leftarrow$GlobalPush ()) succeeds do
11:         Join ($p, T$)
12:         while $ub < forcePush \wedge Q \neq \emptyset$ do
13:             $l \leftarrow$ Q.pop()
14:             while $l$ is not empty do
15:                 $c \leftarrow l$.pop()
16:                 Join (UnboundImpl($c, l.t$),$T$)
17:     return $U$
```

Figure 5.1: Deterministic implementation of scheduling policy.

create a push list $L(t)$ for each template t to store the active equivalence classes w.r.t. t, i.e., classes in which models are produced. Essentially, a push list captures a number of ongoing directions for exploring the upper or lower bounds of a particular template. To effectively manage the push list, one should remove all weaker classes from the list whenever a stronger equivalence class is added, in order to move forward on a k-sequence and avoid non-terminating behaviours (this is implemented by PushListAdd called on line 6). When a point p is sampled, we use it to update the under-approximation (line 7), and the directions (push lists) in which p has extended the frontier of U are scheduled for further investigation (line 8). The scheduling is achieved by maintaining a queue of push lists Q in a certain order. We have experimented with some heuristics for picking good push lists from Q to work on. Details can be found in Section 5.2.7.

In the SymbaMain function, we start by applying the GlobalPush rule to obtain an initial point p. We generate a k-sequence starting at p (for each $t \in T$) by applying UnboundedImpl until either Unbounded is applicable in which case the template is unbounded, or Unbounded-Fail is not applicable, in which case we apply GlobalPush to obtain a new initial point and start the process again. It is easy to check that this is a fair sequence, and therefore this process always terminates.

To evaluate different schedules, we added a parameter *forcePush* and forced a GlobalPush call after *forcePush* calls to UnboundedImpl (line 12). When *forcePush* is set to ∞, the scheduling policy is as described above.

5.2.2 Checking Unbounded Templates

Our implementation of Unbounded and Unbounded-Fail exploits the incremental (Push/Pop) interface that most SMT solvers supply. Moreover, instead of implementing the Bounded rule explicitly, we show how to update the over-approximation for *free*, as a side effect of applying the Unbounded rules.

Fig. 5.2 shows our implementation of the Unbounded rules. We assume that there

```
 1: function UnboundImpl($c \in \mathcal{P}(\mathcal{E}(\varphi)), t_i \in T$)
 2:     Push()
 3:     Assert($t_i > U[t_i]$)
 4:     if UNSAT then
 5:         $O[t_i] \leftarrow U[t_i]$; Pop(); return
 6:     Assert($\bigwedge c$)
 7:     if UNSAT then
 8:         RemoveStronger($L(t), c$); Pop(); return
 9:     Assert($\bigvee(\mathcal{E}(\varphi) \setminus c)$)
10:     if SAT then                                  ▷ Unbounded-Fail
11:         Pop(); return GetModel()
12:     else                                         ▷ Unbounded
13:         $O[t_i] \leftarrow \infty$
14:     Pop(); return
```

Figure 5.2: Implementation of Unbounded and Unbounded-Fail.

is a global SMT *context* in which the formula φ has been asserted. An active equivalence class c and a template t_i are passed in as parameters. $U(i)$ and $O(i)$ refer to the i-th element of the vector representing the abstract states U and O, respectively. SAT and UNSAT refer to the current state of the SMT context, and GetModel() returns a model satisfying the current state of the context if one exists. Push() and Pop() are used to store and restore the state of the context, respectively.

We start by incrementally asserting the conditions of Unbounded implicitly. Given c and t_i, we know that there is a previously sampled point $p_1 \models c$ such that $t_i(p_1) \leqslant U[t_i]$. First, in lines 3-8, we check if there exists a model $p_2 \models \varphi$ such that $t_i(p_2) > t_i(p_1)$ and $[p_2] = [p_1]$. We do this in two stages. We first check if there exists p_2 such that $t_i(p_2) > t_i(p_1)$. If not, we can update the over-approximation O accordingly (line 5). Otherwise, we check if there exists p_2' in the same equivalence class as p_1 (line 6). If such p_2 does not exist, we can remove all equivalence classes that are stronger than c from $L(t)$ since this direction has been proved to be a dead end (line 8). Given that p_2 exists, we check for the existence of p_3 in a stronger equivalence class (lines 9-13). If p_3 exists, we apply Unbounded-Fail; otherwise, we apply Unbounded.

5.2.3 Parallel Symba

Our non-deterministic description of Symba as a set of rules provides a natural way for implementing parallel versions of Symba. Our implementation of Symba allows running different threads of Symba, each one computing the best abstraction $\widehat{\alpha}(\varphi)$ w.r.t. a subset of the templates in T. Specifically, given a formula φ and a set of templates T, we first divide T into $T_1, \ldots, T_n$ mutually exclusive subsets of T. We then run n instances of

Symba in parallel, each one with a different set of templates T_i. After all instances return results, we can easily synthesize the best abstraction $\widehat{\alpha}(\varphi)$.

5.2.4 Dealing with Topologically Open Formulas

One limitation of Symba as well as other numerical abstract domains is that they expect topologically-closed formulas φ. But topologically-open formulas are inevitable when dealing with real-life programs. To ensure that the queries made by UFO are topologically-closed, we have to first convert topologically-open formulas into topologically-closed ones. There are two easy ways of doing this.

Approximation. The first way is to find a topologically-closed counterpart of the formula that over-approximates it, i.e., find φ' such that $\varphi \Rightarrow \varphi'$ and φ' is closed. It can be easily shown that if φ is in negation normal form (NNF), then replacing all the strict inequalities $(<, >)$ by their non-strict counterparts $(\leqslant, \geqslant)$ produces a topologically-closed formula φ' that subsumes φ.

The problem with this approach is that the resulting approximation φ' might lose too much precision. For example, suppose we have an open formula $\varphi \equiv x > 2 \wedge (x \leqslant 2 \vee y \geqslant 3)$. The best abstraction of φ over $\mathcal{A}_T$ with templates $T = \{x, y\}$ is $x > 2 \wedge y \geqslant 3$. But the best abstraction for the closed formula $\varphi' \equiv x \geqslant 2 \wedge (x \leqslant 2 \vee y \geqslant 3)$ is $x \geqslant 2$. In fact, we completely lose the information of y carried in φ.

To our knowledge, there is no easy way to avoid this precision loss other than converting the formula into disjunctive normal form (DNF) first. However, this step alone can be exponentially expensive.

Under Integer Assumption. Another workaround to this problem is to make the assumption that all variables are integer-valued. This assumption is valid in our case, as UFO works only on integer-valued programs and all data can be represented by integers. Using integers, we can replace atoms of the form $A < B$ by $A \leqslant B - 1$ and $A > B$ by $A \geqslant B + 1$ (assuming the formula is in NNF). As a result, we end up with a topologically-closed formula φ' in $\mathcal{L}$ that is equivalent to φ under integer assumption.

We have implemented this method in our experiments. Therefore, the results computed by Symba preserve the precision of the original formulas.

5.2.5 Integer Rounding

The *integer rounding* (IR) rule, shown in Fig. 5.3, is a simple optimization that improved the performance of Symba. Integer rounding extends the upper bounds of a template

$$\frac{U = (k_1, \ldots, k_n) \quad k_i \in \mathbb{R} \quad p \models \varphi \quad t_i(p) = \lceil k_i \rceil}{\langle M, U, O \rangle \to \langle M \cup \{p\}, U \sqcup \widehat{\alpha}(p), O \rangle} \text{IR}(t_i \in T)$$

Figure 5.3: Integer rounding rule.

t_i to the closest integral value, if possible. We have found that applying this rule after Unbounded-Fail enables faster convergence and decreases the number of SMT solver calls required. Details are shown in Chapter 6.

5.2.6 Constraints Reduction

An effective optimization technique is to limit $\mathcal{E}(\varphi)$ to a "relevant" subset when applying the Unbounded rule. In our experiments, we noticed that the set $\mathcal{E}(\varphi)$ of equality constraints can be quite large, which burdens the SMT solver. Removing irrelevant equality constraints decreases the size of the SMT queries. To find the set of "relevant" constraints, we define a relation $\propto$: $\mathit{Atoms}(\varphi) \times \mathit{Atoms}(\varphi)$ as follows: $P \propto P'$ if and only if

1. $\mathit{Vars}(P) \cap \mathit{Vars}(P') \neq \emptyset$, or
2. $\exists P'' \in \mathit{Atoms}(\varphi) \cdot P \propto P'' \wedge P'' \propto P'$,

where $\mathit{Vars}(P)$ is the set of variables appearing in P. This relation can be implemented using the *union find* algorithm. We maintain a disjoint-set data structure that is partitioned into a number of disjoint subsets with each subset containing a number of "related" variables and expressions. We pre-process all the constraints in $\mathcal{E}(\varphi)$ and templates in T to make sure that an equality constraint and a template appear in the same subset if and only if they are "related".

We then define *the equivalence class of* p *w.r.t.* t as

$$[p]_t = \{a \in \mathcal{E}(\varphi) \mid p \models a \wedge t \propto a\}.$$

Removing constraints that are not $\propto$-related to t corresponds to carrying out our algorithm on the projection of φ onto a lower-dimensional space, where the projection is guaranteed to have the same maximum value for t as φ; thus, correctness is not affected.

5.2.7 Priority Function

As mentioned in Section 5.2.1, we maintain a queue of push lists to manage the ongoing directions for exploration. In our experience, the sequence of exploration can have a drastic impact on the performance of Symba. To experiment with possible heuristics on the sampling sequence, we used a priority queue for push lists, and the priorities of the push lists are decided by a priority function.

One of the heuristics we tried is to take the variation trend of $U[t_i]$ (the maximum value of template t_i among all sampled points) into consideration. A pattern was observed in some examples: templates that grow faster also terminate faster, i.e., a bound is found or the template is proved unbounded. Therefore, we kept track of the value changes for each $U[t_i]$ and calculated the priority as follows:

$$\text{Priority}(t_i) = a \times (U[t_i] - U[t_i]') + b \times |L(t_i)|,$$

where a and b are integers, $U[t_i]'$ is the previous value of $U[t_i]$ and $L(t_i)$ is the push list for t_i. We also consider the size of the push lists in the function because this helps balance the distribution of the number of points sampled in all directions. By adjusting the weight factors a and b, we can easily control the sequence of templates to work on based on the variation trend of $U[t_i]$. The weight factors we used in our experiments are $a = 1$ and $b = 10$.

However, in our experiments, the effect of this priority function is still not fully understood. The performance of Symba is largely improved for some benchmarks, while decreased for some others.

Chapter 6

Evaluation

6.1 Benchmarks

6.1.1 Format

To ease the comparison of different symbolic abstraction algorithms and facilitate research in this field, we define a standardized benchmark format for symbolic abstraction problems. Each problem is encoded as a SMT-LIB2 assertion which is recognized by most analysis tools. Given a formula φ and a template T, the asserted expression is an implication $\varphi \Rightarrow \bigwedge\limits_{t_i \in T} t_i < k_i$, where k_i is some real constant.

```
(declare-const x Real)
(declare-const y Real)
(declare-const k1 Real)
(declare-const k2 Real)
(assert (=>
 (and (and (<= 1.0 y) (<= y 3.0))
      (or (and (<= 1.0 x) (<= x 3.0))
          (>= x 4.0)))
 (and (< y k1) (< (+ x y) k2))))
```

Figure 6.1: Problem file for the 2-D example.

Each SMT-LIB2 benchmark consists of a problem file and a result file. Recall the formula φ used in the 2-D example in Section 3.1. Here, $\varphi \equiv 1 \leqslant y \leqslant 3 \wedge (1 \leqslant x \leqslant 3 \vee x \geqslant 4)$, and the template $T = \{y, x + y\}$ are encoded as the problem file in Fig. 6.1. Fig. 6.2 is the solution file for the same example. It indicates that $1 \leqslant y \leqslant 3 \wedge 2 \leqslant x + y$ is the best symbolic abstraction of φ in $\mathcal{A}_T$. [1]

[1]Infinity is written as `1/0` in the solution file.

```
1: (y:REAL) : [1,3]
2: ((x:REAL)+(y:REAL)) : [2,1/0]
```

Figure 6.2: Solution file for the 2-D example.

6.1.2 Benchmark Generation

Our benchmark suite consists of a set of C programs from the Software Verification Competition (SV-COMP) [15]. The benchmarks cover a range of software, from Linux and Windows device drivers to models of SSH and sequentialized concurrent SystemC programs.[2] We narrowed the set of 2,000+ benchmarks down to 604 C programs that were not trivially discharged (proved correct or incorrect) by UFO.

We instrumented UFO to record abstract post queries, and collected 12K+ queries made by UFO on these C programs. Each abstract post query is represented by a formula encoding a set of initial states and a program fragment between two cutpoints (as in *large block encoding* [7, 29]). For the set of templates, we used all variables (as well as their negation) that are in scope at the destination cutpoint, effectively implementing a symbolic abstraction of the intervals domain. The post queries and the corresponding templates were combined as described in Sec. 6.1.1 and then converted into the SMT-LIB2 problem files through the Z3 C API using the function `z3_to_smtlib2_with_decl`.

From the generated symbolic abstraction queries, we focused on the hardest 295 benchmarks, which took Symba greater than 0.4s to process. Our experiments were conducted on a machine running Linux with an Intel i5 3.1GHz processor and 4GB of RAM. The benchmark set is available at `https://bitbucket.org/liyi0630/symba-bench`. Files are organized by directories named after the SV-COMP programs where the abstract post queries were originated. Results produced by Symba are also published for reference and comparison.

6.2 Experimental Evaluation

Our experimental evaluation is designed to evaluate the *efficiency* and *precision* of Symba, to assess the effectiveness of our optimizations (e.g., integer rounding), and to compare Symba against other symbolic abstraction techniques and also classical abstract transformer implementations.

We conducted two classes of experiments: (1) an evaluation of the effects of different optimizations on the efficiency of Symba, as well as comparison with other SMT-based

[2]We drew our benchmarks from the following SV-COMP categories: `ControlFlowIntegers`, `SystemC`, `ProductLines`, and `DeviceDrivers64`.

symbolic abstraction algorithms, and (2) a precision comparison of loop invariants generated by Symba's symbolic abstraction algorithm against traditional implementations of abstract transformers. We describe these below.

6.2.1 Evaluating Symba

Configurations of Symba

We evaluated several configurations of Symba:

1. SymbaIR(*forcePush*): Symba with integer rounding (IR), where *forcePush* represents the number of UnboundedImpl calls before GlobalPush is forced.
2. SymbaIROFF: Same as SymbaIR(∞), but with the integer rounding optimization turned off.
3. SymbaPAR(i): A parallel version of SymbaIR(∞), where i represents the number of templates per thread.

DNFBound

To compare Symba against other possible symbolic abstraction techniques, we implemented DnfBound, a symbolic abstraction algorithm which can be viewed as a fully symbolic implementation of the SMT-guided abstract interpretation algorithm described in [30]. Given a formula φ, DnfBound starts with an under-approximation $U = \bot$ of $\widehat{\alpha}(\varphi)$. It works by lazily building the *disjunctive normal form* (DNF) of φ, computing the best $\mathcal{A}_T$ abstraction of each disjunct (conjunction of linear inequalities), and joining it to the under-approximation until $\varphi \Rightarrow \widehat{\gamma}(U)$. When φ encodes a program fragment, each disjunct effectively encodes a path through the fragment. Our implementation of DnfBound uses Z3 to check subsumption and sample disjuncts of φ, and the APRON numerical abstract domain library to compute best abstractions of each disjunct.

Example. Recall the example from Sec. 3.2. We illustrate DnfBound on the same formula in Fig. 6.3, where the set of templates is now extended to $T = \{x, y, z, -x, -y, -z\}$, i.e., we would like to find the least upper and greatest lower bounds for all the three dimensions at the same time. Suppose the first point sampled is $\mathsf{p_1} = (1.3, 0.4, 1)$ which satisfies all constraints in $\mathit{Atoms}(\varphi)$ except $4y = 3x + 3$. Thus, a disjunct of φ satisfied by $\mathsf{p_1}$ is $\varphi_{d_1} = 0 \leqslant x \leqslant 3 \wedge 0 \leqslant z \leqslant 2 \wedge 2y \leqslant -x + 4$ (this is also a prime implicant of φ, shown in bold in Fig. 6.3a). Then we update the under-approximation to be $\widehat{\alpha}(\varphi_{d_1})$, a box hull surrounding the convex shape φ_{d_1} in this case,

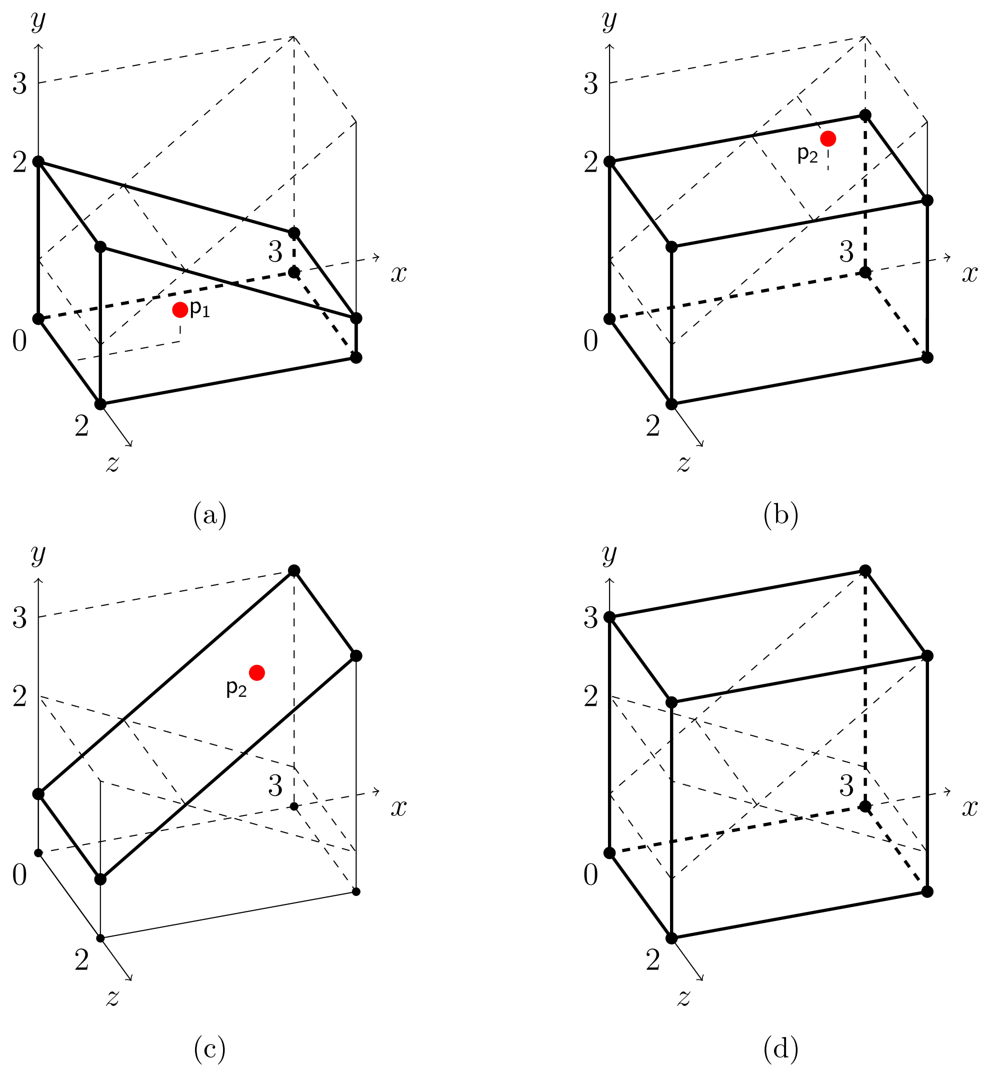

Figure 6.3: Illustration of DnfBound on a 3-D example.

i.e., $U = 0 \leqslant x \leqslant 3 \wedge y \leqslant 2 \wedge 0 \leqslant z \leqslant 2$ (shown in bold in Fig. 6.3b). After that, a point $\mathsf{p}_2 = (2.2, 2.4, 1)$ is picked outside of U. Obviously, it has to satisfy the disjunct $\varphi_{d_2} = 0 \leqslant x \leqslant 3 \wedge 0 \leqslant z \leqslant 2 \wedge 4y = 3x + 3$ (the hyperplane shown in bold in Fig. 6.3c), which corresponds to $U' = 0 \leqslant x \leqslant 3 \wedge 0.75 \leqslant y \leqslant 3 \wedge 0 \leqslant z \leqslant 2$. After joining U' with U, we get an under-approximation $U = 0 \leqslant x \leqslant 3 \wedge y \leqslant 3 \wedge 0 \leqslant z \leqslant 2$ that subsumes φ (shown in bold in Fig. 6.3d). Therefore, at this point, DnfBound terminates with the best abstraction U of φ in $\mathcal{A}_T$.

Z3Qelim

We also compared against the strongest post-condition computation (a harder problem) implemented using the highly efficient Z3 quantifier elimination algorithm Z3Qelim. We

	Configuration	Total Time(s)	# SMT Queries	# Solved	# GlobalPush	# UnboundedImpl
1	SymbaIR(1)	1,707	136,766	295	69,321	32,948
2	SymbaIR(3)	560	74,217	295	16,138	30,861
3	SymbaIR(8)	562	65,185	295	6,734	31,948
4	SymbaIR(13)	538	65,019	295	5,502	32,603
5	SymbaIR(∞)	569	69,785	295	5,491	34,978
6	SymbaIROFF	708	72,680	295	5,478	35,910
7	DnfBound	1,562	208	48		
8	Z3Qelim	669	-	39		
9	SymbaPAR(2)	634.97	-	295		
10	SymbaPAR(3)	522.63	-	295		
11	SymbaPAR(4)	460.48	-	295		

Table 6.1: Overall results on 295 SMT-LIB2 benchmarks.

start by constructing a formula

$$\varphi' \equiv \exists x_1, \ldots, x_n \in \mathit{Vars}(\varphi) \cdot \varphi \wedge \bigwedge \{z_i = t_i \mid t_i \in T\},$$

where each z_i is a fresh variable corresponding to t_i. Then applying quantifier elimination on φ' (eliminating x_i) corresponds to the best symbolic abstraction in the finite power set of T.

Results

Table 6.1 summarizes the results of running all the aforementioned algorithms and configurations on the 295 SMT-LIB2 benchmarks with a timeout of 100 seconds per benchmark. The average number of templates per benchmark is 69, with a minimum of 4 and a maximum of 380. The average number of variables per benchmark is 1,435, with a minimum of 32 and a maximum of 19,152.

The results of running SymbaIR(∞) are summarized in row 6 of Table 6.1. It successfully computed best abstractions for all 295 benchmarks in 569 seconds. In the process, it made ~70K SMT queries using 5,491 invocations of GlobalPush and 34,978 invocations of UnboundedImpl.

Rows 1-5 capture results of running SymbaIR(*forcePush*), where *forcePush* is 1, 3, 8, 13, and ∞, respectively. Total running times in all but the first case are comparable, around 560 seconds. However, when *forcePush* is 1, the total time goes up to ~1700 seconds due to the frequent calls to GlobalPush (after every call to UnboundedImpl). This is reflected in the total number of SMT queries (~130K), which is almost double the number of queries made for higher values of *forcePush* (~70K), and indicates the sensitivity of the scheduling policy and the importance of applying the GlobalPush rule sparingly.

Row 6 (SymbaIROFF) of Table 6.1 summarizes the effect of running Symba without the integer rounding rule. It requires the total of 708 seconds (compared to 569 seconds for SymbaIR(∞), which is the same configuration but with integer rounding). The extra computation time is spent in unnecessary SMT queries. While the total number of GlobalPush calls is almost the same in both configurations, the number of UnboundedImpl calls in SymbaIROFF is greater. Since we are dealing with integer programs, the vast majority of least upper bounds we compute are integers. Therefore, enabling the integer rounding heuristic often allows faster convergence without the need of calling the potentially expensive UnboundedImpl.

DnfBound (see row 7 of Table 6.1) was able to solve only 48 benchmarks. Interestingly, it made a small number of calls to Z3: we observed that the bottleneck is calling APRON, since the disjuncts can involve a large number of linear inequalities. To minimize the number of disjuncts, we used UNSAT cores produced by Z3 to compute *prime implicants* of the disjuncts. While this optimization often significantly reduces the size of a disjunct, it only slightly improved the overall performance of DnfBound. (The version of DnfBound reported here includes this optimization.) In comparison to DnfBound, all configurations of Symba including the least efficient ones solved all of the benchmarks. This is a strong indication of the power and utility of Symba's symbolic abstraction approach.

We have also tried using Z3 to compute the strongest post condition over the TCM domain by performing quantifier elimination (row 8). The results show that our benchmarks are quite difficult – Z3 was only able to solve 39 of them in the allotted time.

Finally, we compared our parallelized version SymbaPAR(i) with the sequentialized SymbaIR(∞). The results of running SymbaPAR(i) are summarized in rows 9-11. When the number of templates per thread is 2 ($i = 2$), i.e., $\lceil |T|/2 \rceil$ instances of Symba are running in parallel, the total time is longer than the sequentialized version (row 9). This implies that the overhead of pre-processing and threading has exceeded the benefits brought by the parallelization. When $i = 3$ and $i = 4$, the total running time is 523 and 460 seconds, respectively, which is shorter than all the other configurations. But the performance gain is far from linear in the number of threads running. Parallelism still does not work ideally for the current implementation of Symba.

Fig. 6.4 shows the number of benchmarks solved as we increase the timeout limit for various Symba configurations. Note that SymbaIR(∞) solves most of the benchmarks in under 3 seconds, with only a few taking longer than 10 seconds. Most of these "slow" examples were generated from `email*` benchmarks in the `ProductLines` category of SV-COMP. While the programs are relatively small ($\sim$5KLoC), most of the computation

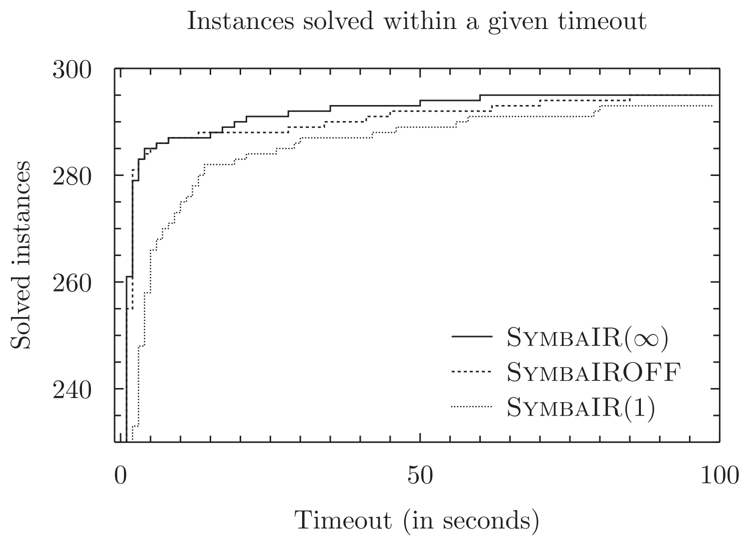

Figure 6.4: Number of benchmarks solved vs. timeout in seconds for several configurations of Symba.

AbsDom	# Completed	Time(s)	# Safety Proofs	Precision Gain(%)
SymbaIR(∞)	592	4,024	47	73
Intervals	604	121	0	0

Table 6.2: Overall results for loop invariant generation on 604 C benchmarks.

occurs within a single loop, making loop-free program fragments very large and resulting in the largest formulas in our benchmark suite, with 3K+ variables.

6.2.2 Symba for Invariant Generation

We have integrated Symba into the UFO verification and analysis framework, and used it to generate loop invariants. We applied UFO/Symba to the set of 604 C programs, of which 373 are safe and 231 are unsafe, with a total of ∼2.7MLoC, drawn from the SV-COMP benchmarks. We instantiated Symba with the set of interval templates.

We compared the precision of loop invariant generation using Symba against a *standard* intervals analysis, Intervals, where abstract post is performed over single instructions. Table 6.2 shows the aggregate results of this comparison. Within a timeout of 500 seconds, Symba computed loop invariants for 592 programs in 4,024 seconds. Compared with Intervals, Symba produced more precise loop invariants for 73% of the loops in the benchmark suite. For the remaining 27%, the generated invariants were equivalent.

Since these benchmarks were drawn from the Software Verification Competition, they contain assertions encoded as an error location. Symba's invariants proved the error location is unreachable for 47 out of 373 safe benchmarks, while invariants generated by

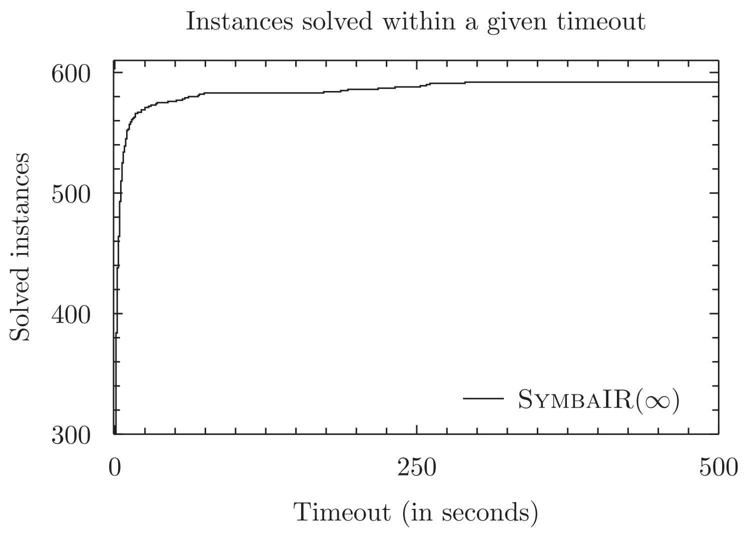

Figure 6.5: Number of C files analyzed using SYMBAIR(∞) as the abstract transformer vs. timeout in seconds.

INTERVALS could not prove any of the benchmarks safe.

In terms of total time, SYMBA required 4,024 seconds compared to 121 seconds for INTERVALS. SYMBA timed out for 12 benchmarks. To examine where most time is spent, consider Fig. 6.5 which shows the number of benchmarks analyzed by SYMBA as the timeout limit is increased. This figure shows that most benchmarks are solved within a 20 second timeout. A closer look at the benchmarks shows that the hardest programs to analyze come from two SV-COMP categories: (1) `email*` programs from `ProductLines`, which (as we discussed previously) exhibit very large loop-free fragments; and (2) Linux device drivers from `DeviceDrivers64`, which include over 140 templates per query. To address (1), we could avoid computing abstract post over large program fragments by adding artificial cutpoints in the program, e.g., as proposed for predicate abstraction in [31], thus shrinking the size of symbolic abstraction queries at the expense of precision. To address (2), we could utilize the over-approximation of the best abstraction maintained by SYMBA. That is, if a symbolic abstraction query exceeds a time limit, we could halt the algorithm prematurely and use the over-approximation as the result.

6.2.3 Symba in an Abstraction Refinement Loop

We have also integrated SYMBA in an abstraction refinement loop [11] where interpolants are used to iteratively strengthen invariants produced by abstract interpretation. Due to the unpredictability of the refinement process, it is not obvious how the increased precision afforded by SYMBA over INTERVALS will impact overall analysis results. Our results show that Symba generally decreases the number of refinements required, and can verify examples that Intervals cannot.

6.2.4 Summary

Our experiments indicate the power of our proposed symbolic abstraction technique and its SMT-based implementation for computing precise abstract post in the TCM domain over loop-free program fragments. For our first set of experiments, we evaluated different configurations of Symba, as well a path-based symbolic abstraction algorithm, on a large number of symbolic abstraction queries. Our experiments demonstrated the efficiency of Symba's approach and the importance of the integer rounding optimization and careful rule scheduling.

For loop invariant generation, our evaluation on 600+ C programs (with a total of ~2.7MLoC) shows more precise invariants for 73% of the loops when using Symba versus a traditional implementation of abstract post (with imprecision due to joins and abstract transformers over single instructions). The precision gain is reflected in the number of benchmarks proved correct by Symba's invariants (47 out of 373). Using a standard intervals domain, we could not prove any benchmark correct. Of course, due to the SMT-heavy nature of Symba, it took a larger amount of time than the intervals domain. We have shown, though, that the vast majority of programs were analyzed within 20 seconds, and most of the time was spent on harder examples with very large loop-free fragments and 100s of templates. We believe that the continuous improvements in the efficiency and scope of SMT solvers will pave the way for more efficient symbolic abstraction algorithms, and perhaps symbolic abstract domains that transcend linear numerical constraints, e.g., arrays or non-linear arithmetic.

Chapter 7

Related Work

7.1 Symbolic Abstraction of Numerical Domains

Sankaranarayanan et al. [12] proposed the TCM domain, which lies between the intervals and the more general polyhedra domain. They showed that basic domain operations of TCM can be posed as linear programming problems of polynomial time complexity. Their method only applies to straight-line code, i.e., transfer functions represented as conjunctions of inequalities. Monniaux [32] also presented an approach that finds the best transformer in TCM domain for straight-line code, using quantifier elimination techniques. In contrast to [12, 32], Symba uses SMT solvers to compute precise TCM post for arbitrary loop-free program fragments, thus making it more general and allowing it to avoid imprecision due to joins.

In [30], Monniaux and Gonnord proposed a technique that lazily enumerates program paths in loop-free fragments in order to avoid imprecise join operations. The idea is to compute abstract post over individual program paths and use an SMT solver to check if all paths are subsumed by the computed abstract state. We have implemented a similar algorithm (DnfBound) for the TCM domain and compared it with Symba in Section 5, showing scalability limitations of the former. Furthermore, Symba is a fully symbolic technique: The SMT solver is not only used to check if a given approximation is an abstraction of the formula, but it also used to compute the abstraction.

Recently, Thakur and Reps [18] proposed a generalization of Stålmarck's SAT solving method [33] to richer logics. The algorithm attempts to prove a formula φ unsatisfiable by iteratively refining an over-approximation of φ starting from $\top$ until arriving at $\bot$. They showed how the algorithm can be instantiated with abstract domains, such as polyhedra, and used to compute best abstractions of formulas in QF_LRA. Their approach is a general framework for symbolic abstraction that is applicable to a wide range of logics and abstract domains. In contrast, Symba is a domain-specific algorithm, designed with efficiency in mind, that exploits the state-of-the-art in SMT solving. Unfortunately, there is no available implementation that we could compare against, and no evaluation of their approach exists for computing precise abstract post over numerical domains.

7.2 Other Symbolic Abstraction Techniques

Symbolic abstraction algorithms have been proposed for non-numerical domains as well. Reps et al. [19] described an approach for computing the best abstract transfer function over finite-height, but possibly infinite-size, abstract domains. Both Symba and the approach in [19] sample models of a formula and use them to grow an under-approximation. In fact, [19] can be viewed as an iterative application of the GlobalPush rule. Convergence is guaranteed by requiring a finite-height domain. To deal with the infinite-height of TCM domains, Symba employs the additional Unbounded rule. Moreover, Symba maintains both an under- and an over-approximation, allowing it to produce sound but less precise results, if desired. Yorsh et al. [34] introduced a method similar to [19] for computing $\widehat{\alpha}$ over abstract domains that are used in shape analysis ("canonical abstraction" of logical structures [35]).

More recently, Thakur and Reps [21] proposed an algorithm that generalizes the symbolic abstraction algorithm of King and Søndergaard [36] from Boolean affine relations to the finite-height domains handled by [19]. In a similar fashion as Symba, it maintains an under- and an over-approximation, but does not compute best abstract transformers for domains with infinite height.

7.3 Predicate Abstraction

The use of decision procedures for computing abstract transformers was pioneered by Graf and Saïdi in their work on predicate abstraction [37]. They showed how theorem provers can be used to construct abstract transformers for abstract domains whose elements are arbitrary Boolean combinations of a finite set of predicates, e.g., linear inequalities. Cartesian predicate abstraction [38] is a less expensive (but less precise) predicate domain whose elements are restricted to conjunctions of predicates. Similar to Symba, software model checking techniques using predicate abstraction [7, 31, 39] encode loop-free program fragments as formulas and use SMT solvers to compute abstract post. Cartesian predicate abstraction can be viewed as a TCM domain with Boolean-valued (rather than real-valued) templates. Therefore, Symba is more general than Cartesian predicate abstraction in QF_LRA.

7.4 Classification and Machine Learning

A fundamental problem in machine learning is *classification*: given a set of positive and negative examples, find a classifier that predicts whether a given example is positive or negative. For example, using Support Vector Machines (SVMs) [40], one can compute linear inequalities separating positive and negative points in some space $\mathbb{R}^n$.

Symba can be viewed as a sophisticated classification algorithm, where positive and negative examples are models of φ and $\neg\varphi$, respectively. The goal is to find the best classifier, represented by a conjunction of linear inequalities (templates), that does not misclassify any of the positive examples (i.e., contains φ). Symba only samples positive examples (from φ) and keeps weakening a classifier (the under-approximation U) until it encompasses all positive examples. As Reps et al. point out in [19], weakening an under-approximation by sampling more points is analogous to the approach of the simple learning algorithm Find-S [41]. Find-S gradually weakens a classifier, starting from $\perp$, by iteratively taking into account more and more positive examples.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Numerical invariant generation is an important program analysis task with uses including program optimization, verification, and bug finding. Abstract interpretation with numerical abstract domains is one of the most studied and most efficient approaches for numerical invariant generation. This efficiency is often achieved by employing imprecise operations, such as join and abstract post. Moreover, expressive domains such as polyhedra are expensive, and efficient domains such as intervals can lack expressivity required for proving interesting program properties.

In this thesis, we have presented Symba: a novel approach that utilizes the power of SMT solvers to implement precise abstract transformers over large loop-free program segments encoded in linear real arithmetic. As a result, Symba does not suffer from imprecision incurred by abstract post over single instructions or basic blocks, and avoids imprecise joins by symbolically and efficiently enumerating program paths. Symba enables invariant generation in the general Template Constraint Matrix domain, which encompasses many popular domains including intervals, octagons, and octahedra.

We have implemented Symba in UFO, an analysis and verification tool, and applied it to a large number of C programs (a total of 2.7MLoC), ranging from Linux and Windows device drivers to models of SSH and sequentialized SystemC programs. In comparison with standard abstract transformers, Symba produced more precise loop invariants for 73% of the loops, allowing it to prove 47 (out of 373) programs safe; standard abstract interpretation could not prove any. Furthermore, our results indicate the efficiency of Symba's technique in comparison with other symbolic abstraction approaches, and highlight the power of our design choices.

8.2 Future Work

More efforts should be put into exploring various possible heuristics on the scheduling policy of Symba. A change in the priority function for choosing push lists could potentially alter the sequence of sampling drastically. It is still unclear whether or not an optimal sequence exists. From an engineering perspective, the implementation of Symba can be improved in terms of parallelism. Current results indicate that parallelization by splitting the set of templates is not as effective as we expected. We would like to experiment with rearranging and subdividing computation-intensive tasks such as SMT queries in different ways.

We believe that the continuous developments in SMT solving and decision procedures call for further investigation of their applications within an abstract interpretation context. In the future, we would like to extend our approach to other SMT theories. For example, it would be interesting to use the theory of arrays to reason about memory operations and compute invariants within quantified abstract domains.

So far, we have only focused on the TCM domain. A natural extension of our work is to apply our techniques to more precise domains, such as the polyhedron domain [5]. It would be interesting to examine the effects of such precision improvement on the performance of real verification tools like UFO.

Bibliography

[1] R. Floyd, "Assigning Meanings to Programs," *Mathematical Aspects of Computer Science*, vol. 19, no. 19-32, p. 1, 1967.

[2] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. of POPL'77*, pp. 238–252, 1977.

[3] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs," in *Proceedings of the Colloque sur la Programmation*, April 1976.

[4] A. Miné, "The Octagon Abstract Domain," *J. Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[5] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints among Variables of a Program," in *Proc. of POPL '78*, pp. 84–96, 1978.

[6] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS'04*, vol. 2988 of *LNCS*, pp. 168–176, Springer, March 2004.

[7] A. Gurfinkel, S. Chaki, and S. Sapra, "Efficient Predicate Abstraction of Program Summaries," in *Proc. of NFM'11*, vol. 6617 of *LNCS*, pp. 131–145, 2011.

[8] B. Cook, A. Podelski, and A. Rybalchenko, "Termination Proofs for System Code," in *Proc. of PLDI'06*, pp. 415–426, 2006.

[9] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing Software Verifiers from Proof Rules," in *Proc. of PLDI'12*, pp. 405–416, 2012.

[10] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *Proc. of SAT'12*, pp. 157–171, 2012.

[11] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Craig Interpretation," in *Proc. of SAS'12*, vol. 7460 of *LNCS*, pp. 300–316, 2012.

[12] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Scalable Analysis of Linear Systems using Mathematical Programming,” in *Proc. of VMCAI'05*, pp. 25–41, 2005.

[13] A. Albarghouthi, A. Gurfinkel, and M. Chechik, “UFO: A Framework for Abstraction- and Interpolation-Based Software Verification,” in *Proc. of CAV'12*, 2012.

[14] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proc. of CGO'04*, pp. 75–88, 2004.

[15] D. Beyer, “Competition On Software Verification - (SV-COMP),” in *Proc. of TACAS'12*, vol. 7214 of *LNCS*, pp. 504–524, 2012.

[16] K. G. Murty, *Linear programming.* Wiley, 1983.

[17] G. A. Kildall, “A Unified Approach to Global Program Optimization,” in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, (New York, NY, USA), pp. 194–206, ACM, 1973.

[18] A. V. Thakur and T. W. Reps, “A Method for Symbolic Computation of Abstract Operations,” in *Proc. of CAV'12*, pp. 174–192, 2012.

[19] T. Reps, M. Sagiv, and G. Yorsh, “Symbolic Implementation of the Best Transformer,” in *Proc. of VMCAI'04*, vol. 2937 of *LNCS*, 2004.

[20] T. Reps, M. Sagiv, and R. Wilhelm, “Static Program Analysis via 3-Valued Logic,” in *Proc. of CAV'04*, vol. 3114 of *LNCS*, pp. 15–30, 2004.

[21] A. V. Thakur, M. Elder, and T. W. Reps, “Bilateral Algorithms for Symbolic Abstraction,” in *Proc. of SAS'12*, pp. 111–128, 2012.

[22] “Clang: A C Language Family Frontend for LLVM.” `http://clang.llvm.org/`.

[23] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proc. of TACAS'08*, vol. 4963 of *LNCS*, pp. 337–340, 2008.

[24] R. DeLine and K. R. M. Leino, “BoogiePL: A Typed Procedural Language for Checking Object-oriented Programs,” tech. rep., Microsoft Research, 2005.

[25] N. Tillmann and W. Schulte, “Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution,” *IEEE Software*, vol. 23, no. 4, pp. 38–47, 2006.

[26] A. Nori, S. Rajamani, S. Tetali, and A. Thakur, “The Yogi Project: Software Property Checking via Static Analysis and Testing,” in *Proc. of TACAS’09*, vol. 5505 of *LNCS*, pp. 178–181, 2009.

[27] B. Jeannet and A. Miné, “Apron: A Library of Numerical Abstract Domains for Static Analysis,” in *Proc. of CAV’09*, pp. 661–667, 2009.

[28] C. Barrett, A. Stump, and C. Tinelli, “The SMT-LIB Standard: Version 2.0,” tech. rep., Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[29] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, “Software Model Checking via Large-Block Encoding,” in *Proc. of FMCAD’09*, pp. 25–32, 2009.

[30] D. Monniaux and L. Gonnord, “Using Bounded Model Checking to Focus Fixpoint Iterations,” in *Proc. of SAS’11*, LNCS, pp. 369–385, 2011.

[31] D. Beyer, M. E. Keremoglu, and P. Wendler, “Predicate Abstraction with Adjustable-Block Encoding,” in *Proc. of FMCAD’10*, pp. 189–197, 2010.

[32] D. Monniaux, “Automatic Modular Abstractions for Template Numerical Constraints,” *Logical Methods in Computer Science*, vol. 6, no. 3, 2010.

[33] M. Sheeran and G. Stålmarck, “A Tutorial on Stålmarck’s Proof Procedure for Propositional Logic,” *Formal Methods in System Design*, vol. 16, pp. 23–58, 2000.

[34] G. Yorsh, T. W. Reps, and S. Sagiv, “Symbolically Computing Most-Precise Abstract Operations for Shape Analysis,” in *Proc. of TACAS’04*, vol. 2988 of *LNCS*, pp. 530–545, 2004.

[35] S. Sagiv, T. W. Reps, and R. Wilhelm, “Parametric Shape analysis via 3-Valued Logic,” *ACM TOPLAS*, vol. 24, no. 3, pp. 217–298, 2002.

[36] A. King and H. Søndergaard, “Automatic Abstraction for Congruences,” in *Proc. of VMCAI’10*, pp. 197–213, 2010.

[37] S. Graf and H. Saïdi, “Construction of Abstract State Graphs with PVS,” in *Proc. of CAV’97*, vol. 1254, pp. 72–83, 1997.

[38] T. Ball, A. Podelski, and S. Rajamani, “Boolean and Cartesian Abstraction for Model Checking C Programs,” in *Proc. of TACAS’01*, vol. 2031, pp. 268–283, 2001.

[39] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *Proc. of POPL'02*, pp. 58–70, 2002.

[40] J. C. Platt, "Fast Training of Support Vector Machines Using Sequential Minimal Optimization," in *Advances in Kernel Methods*, pp. 185–208, Cambridge, MA, USA: MIT Press, 1999.

[41] T. M. Mitchell, *Machine Learning.* New York, NY, USA: McGraw-Hill, Inc., 1997.