Symbolic Optimization with SMT Solvers

Yi Li
University of Toronto
liyi@cs.toronto.edu

Aws Albarghouthi
University of Toronto
aws@cs.toronto.edu

Zachary Kincaid
University of Toronto
zkincaid@cs.toronto.edu

Arie Gurfinkel
Software Engineering Institute, CMU
arie@cmu.edu

Marsha Chechik
University of Toronto
chechik@cs.toronto.edu

Abstract

The rise in efficiency of Satisfiability Modulo Theories (SMT) solvers has created numerous uses for them in software verification, program synthesis, functional programming, refinement types, etc. In all of these applications, SMT solvers are used for generating satisfying assignments (e.g., a witness for a bug) or proving unsatisfiability/validity (e.g., proving that a subtyping relation holds). We are often interested in finding not just an arbitrary satisfying assignment, but one that optimizes (minimizes/maximizes) certain criteria. For example, we might be interested in detecting program executions that maximize energy usage (performance bugs), or synthesizing short programs that do not make expensive API calls. Unfortunately, none of the available SMT solvers offer such optimization capabilities.

In this paper, we present SYMBA, an efficient SMT-based optimization algorithm for *objective functions* in the theory of linear real arithmetic (LRA). Given a formula φ and an objective function t, SYMBA finds a satisfying assignment of φ that maximizes the value of t. SYMBA utilizes efficient SMT solvers as black boxes. As a result, it is easy to implement and it directly benefits from future advances in SMT solvers. Moreover, SYMBA can optimize a set of objective functions, reusing information between them to speed up the analysis. We have implemented SYMBA and evaluated it on a large number of optimization benchmarks drawn from program analysis tasks. Our results indicate the power and efficiency of SYMBA in comparison with competing approaches, and highlight the importance of its multi-objective-function feature.

Categories and Subject Descriptors G.1.6 [*Optimization*]: Constrained optimization; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Invariants

Keywords optimization; satisfiability modulo theories; invariant generation; symbolic abstraction; program analysis

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535857

1. Introduction

Over the past decade or so, we have witnessed an incredible improvement in the performance of *Satisfiability Modulo Theories* (SMT) solvers [8] and the range of logical theories they support. These advances made SMT solvers (e.g., Z3 [21], MathSAT [?], CVC [?], etc.) household names in the programming languages and verification communities, creating an explosion in the range of applications in which they are deployed, and paving the way for innovations that would not have been possible otherwise.

To mention a few, in *verification*, SMT solvers have been used for device driver verification [5, 34], checking complex verification conditions [22, 38, 48], and improving precision of invariant generation [33, 45]; in *testing and bug finding*, they have been instrumental in making symbolic execution [15, 26], fuzzing [28], and bounded model checking techniques [18, 24] practical; in *program synthesis*, they have been used to search for programs satisfying a given specification [30, 58]; in *functional programming*, they have been used to support strong typing guarantees with refinement types [12, 52].

In all of the aforementioned applications, SMT solvers are used for (1) generating satisfying assignments (e.g., a witness for a bug) or (2) proving unsatisfiability/validity (e.g., proving that a subtyping relation holds). To the best of our knowledge, none of the available SMT solvers support finding *optimal* satisfying assignments, i.e., satisfying assignments that minimize (or maximize) a given *objective function*. In this paper, we present SYMBA, an efficient SMT-based optimization algorithm for objective functions in the theory of *linear real arithmetic* (LRA). Given a formula φ and an objective function t, e.g., $x + y$, SYMBA computes the smallest constant k such that $\varphi \Rightarrow t \leqslant k$ [1]. Specifically, SYMBA finds a satisfying assignment of φ that exhibits the least upper bound k (maximum value) of an objective function t. In what follows, we start by arguing that such an algorithm has a wide array of applications in verification, bug finding, synthesis, and others. We then highlight SYMBA's salient features and describe its high-level operation.

Applications of Optimization We start by describing potential applications of SYMBA in the following domains:

- *Numerical invariant generation:* Numerical abstract domains, e.g., intervals [19] and octagons [41], are often used to generate numerical program invariants. The main ingredient in such an analysis is the *abstract transformer*: an operator that takes a set of initial states and an instruction (or a basic block), and com-

[1] Note that k is ∞ if t is unbounded in φ, and $-\infty$ if φ is unsatisfiable

putes a set of reachable states, often using convex optimization techniques. Since SYMBA can optimize over arbitrary LRA formulas (*non-convex optimization*), it can precisely compute abstract transformers over loop-free program fragments (encoded as formulas), without losing precision due to join or multiple applications of the transformer. This problem is known in the literature as *symbolic abstraction* and has been studied for a number of domains [44, 45, 51, 60, 61]. By designing optimization algorithms that exploit the power of SMT solvers, we enable efficient implementations of precise abstract transformers for a variety of numerical domains [54]. We discuss this application in more detail in Sec. 4, where we generate our benchmark suite from symbolic abstraction queries made by a program analysis tool [3].

A similar approach can be used for solving systems of Horn-like clauses in LRA [29, 31], which capture a large number of sequential and concurrent program verification tasks. Since Horn clauses are represented symbolically in SMT-LIB [13], applying traditional numerical abstract domains for solving them is not a straightforward endeavour, but SYMBA can be easily used to build a fixpoint solver for such clauses.

- *Counterexample generation:* In symbolic execution and bounded model checking, program executions are encoded succinctly as a formula. An SMT solver is then used to find a satisfying assignment of this formula that acts as a witness of an erroneous execution. By augmenting the encoding with arithmetic cost functions, we can use SYMBA to find counterexamples that maximize or minimize certain criteria. For example, we might be interested in finding performance bugs, e.g., execution traces with the highest energy or memory consumption. By assigning costs to program instructions and API calls, we can detect such executions using an optimizing SMT solver.
- *Program synthesis:* Program synthesis involves generating a program that satisfies a given specification. For example, in [30, 36], non-trivial bit-manipulating program snippets are synthesized from specifications by using SMT solvers to search through all possible combinations of bit-level operations (instructions). Similar to the counterexample generation described above, the goal is often to synthesize the shortest programs, or ones with the smallest cost. For example, when the synthesized snippet is part of a performance-critical code (e.g., as in superoptimization [40]), the size of the synthesized snippet and the operations it performs are crucial. By augmenting formulas given to the SMT solver with costs, we can instruct it to synthesize programs that minimize a given criterion.
- *Constraint programming:* In recent work [37], Köksal et al. proposed incorporating an SMT solver into an extension of Scala, allowing constraint manipulation as part of the language. One of the important constructs in their language is min/max, which returns the minimum/maximum satisfying assignment of a constraint w.r.t to an objective function, and enables elegant implementations of algorithms for problems like knapsack. Due to the unavailability of off-the-shelf solvers with an optimization feature, the authors use a simple binary search optimization algorithm restricted to bounded discrete domains. The authors of [37] also comment on the state-of-the-art of SMT solvers by saying: *"...we found that a number of features, if natively supported by solvers, could directly bring benefits to constraint programming. These include 1) support for enumeration of theory models and 2) solving constraints while minimizing/maximizing a given term."* SYMBA is thus an answer to point 2 for LRA terms (objective functions).
- *Interpolant generation:* Craig interpolation has proved to be a powerful technique for software verification based on predicate abstraction [35]. Interpolants are generated from unsatisfiability proofs. For linear real arithmetic, this can be performed by constructing a system of constraints whose solution is a proof of unsatisfiability (and an interpolant) [53]. As recently shown [2], the simplicity of the proof can be crucial to discovering the predicates required for a safe inductive invariant. Finding simple proofs boils down to finding an optimal solution of the system of constraints. Using SYMBA, this can be automated, without the need for the heuristics employed in [2].

***Symbolic Optimization with* SYMBA** Given a formula φ and a set of objective functions (*objectives* for short) $T = \{t_1, \ldots, t_n\}$, SYMBA computes the strongest formula

$$\mathsf{opt}_T(\varphi) \equiv t_1 \leqslant k_1 \wedge \ldots \wedge t_n \leqslant k_n,$$

such that $\varphi \Rightarrow \mathsf{opt}_T(\varphi)$, where $k_i \in \mathbb{R} \cup \{\infty, -\infty\}$. We call $\mathsf{opt}_T(\varphi)$ the *optimal solution* of T w.r.t. φ. In other words, SYMBA computes the least upper bound (maximum value) k_i of each objective t_i w.r.t. the satisfying assignments of φ.[2]

Note that we are optimizing a *set* of objective functions, as opposed to a single function as in traditional optimization problems. This allows SYMBA to reuse information computed for one objective in order to speed up the optimization of other objectives. This feature allows *incremental* implementations of SYMBA, where we can continuously ask for optimizing different objectives without losing previously computed information (in a style similar to the push/pop interface implemented by most SMT solvers). The incremental interface is useful, for example, when computing an abstract transformer for the intervals domain, where we need to find the maximum and minimum values of each live variable.

SYMBA maintains both an *under-* and an *over-approximation* of the optimal solution $\mathsf{opt}_T(\varphi)$. 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 optimal solution. 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* objectives.

SYMBA also maintains an over-approximation of $\mathsf{opt}_T(\varphi)$ (when it terminates, the two approximations are equivalent). Maintaining an over-approximation allows us to halt SYMBA at any point and still compute an over-approximation of the optimal solution (e.g., an over-approximation of the best abstract transformer). This makes SYMBA resilient to SMT solver failures and resource (e.g., time) depletion.

Another important feature of SYMBA is that it treats the underlying SMT solver as a black-box, using it to generate models and check validity. This makes SYMBA easy to implement, allowing us to take advantage of the existing highly-optimized SMT solvers without having to dive into their intricate implementations, and directly benefiting from future advances in SMT solving.

Our implementation of SYMBA uses the Z3 SMT solver. We have performed a thorough evaluation of SYMBA on a large set of realistic benchmarks drawn from program analysis tasks. We have also compared SYMBA against [56], which is, to the best of our knowledge, the only other proposed SMT-based optimization technique. The technique of [56] optimizes a single objective function, and is built into the MATHSAT SMT solver. For a fairer compari-

[2] Note that our goal is to find an optimal value for each objective independently (using a different satisfying assignment of φ), as opposed to a *pareto-optimal* satisfying assignment, i.e., a single assignment that optimizes all objective functions and cannot be improved upon.

son against SYMBA, we have also implemented a multi-objective-function extension of the approach in [56] in the Z3 SMT solver (using its available source code [1]) and experimented with various configurations. Our results demonstrate the efficiency and robustness of SYMBA against competing approaches, and highlight the effectiveness of its multi-objective-function feature and our proposed implementation optimizations.

Contributions We summarize our contributions as follows:

- SYMBA: a novel SMT-based optimization algorithm for objective functions in linear real arithmetic, with wide-ranging applications in program analysis, synthesis, etc. SYMBA utilizes efficient SMT solvers as black boxes. Thus, it is easy to implement and it directly benefits from future advances in SMT solving. Moreover, SYMBA can optimize a set of objective functions, reusing information between them to speed up the optimization task.
- An extensive evaluation of SYMBA against other proposed techniques in the literature on a large set of program analysis benchmarks. Our results indicate the power and efficiency of SYMBA in comparison with competing approaches, and highlight the importance of its multi-objective-function feature.
- An implementation of SYMBA, multiple implementations of the approach in [56], and a large set of optimization benchmarks drawn from program analysis tasks. Our source code, binaries, and benchmarks are available at:

`http://bitbucket.org/arieg/ufo`

Organization In Sec. 2, we demonstrate the operation of SYMBA on simple examples. In Sec. 3, we formalize SYMBA and prove its correctness. In Sec. 4, we describe our implementation and experimental evaluation. In Sec. 5 and 6, we compare SYMBA to related work and conclude the paper, respectively.

2. SYMBA by Example

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

2.1 A 2-dimensional Example

Consider the LRA 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 by the black boxes in Fig. 1. Suppose that our set of objectives 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 the minimum value for y, we can simply add $-y$ as an objective to T.) In this example, the optimal solution is

$$\mathsf{opt}_T(\varphi) \equiv y \leqslant 3 \wedge x + y \leqslant \infty,$$

since $x + y$ is unbounded in φ. Formulas of the form $t \leqslant \infty$ are treated as *true*, and $t \leqslant -\infty$ as *false*.

Initially, the under-approximation of $\mathsf{opt}_T(\varphi)$ is

$$U \equiv y \leqslant -\infty \wedge x + y \leqslant -\infty \equiv \textit{false},$$

and the over-approximation is

$$O \equiv y \leqslant \infty \wedge x + y \leqslant \infty \equiv \textit{true}.$$

SYMBA maintains the invariant $U \Rightarrow \mathsf{opt}_T(\varphi) \Rightarrow O$ (note that U is an under approximation of $\mathsf{opt}_T(\varphi)$, not necessarily φ, and similarly with O). 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 objectives. In this example, 3 is the upper bound for y, and $x + y$ is unbounded.

***First* GLOBALPUSH** SYMBA starts with a GLOBALPUSH by querying an SMT solver for a model of φ that is not a model of U (i.e., lies outside the under-approximation). Suppose the solver returns the point $\mathsf{p}_1 = (2, 2)$. The under-approximation U is the strongest formula of the form $t_1 \leqslant k_1 \wedge \ldots \wedge t_n \leqslant k_n$ that contains the set of points found by the solver (i.e., a *convex hull* expressed in terms of the objectives). So, the under-approximation is updated to $U \equiv y \leqslant 2 \wedge x + y \leqslant 4$ (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 Fig. 1.

UNBOUNDED *(p_1, y)* SYMBA now tries to prove that y is unbounded. First, we categorize points into boundary classes as follows: Let $\mathcal{E}(\varphi) = \{l = k \mid l \leqslant k \in \varphi\}$, 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. 1. The boundary 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, $[\mathsf{p}_1] = \emptyset$, 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 objective t is increasing along r, and r never hits any boundaries of φ (i.e., completely contained in φ).

UNBOUNDED 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. 1. Note that p_2 exhibits the upper bound of y; SYMBA detects that and strengthens the over-approximation O to $y \leqslant 3$. After witnessing the point p_2, 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 boundary class as p_3. Then, it tries to find a point p'' that has a boundary 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 Sec. 3.

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, $U \equiv O$, so SYMBA terminates with the optimal solution $y \leqslant 3$.

2.2 A 3-dimensional Example

We now illustrate the operation 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. 2. Suppose, for simplicity, that we would like to find the least upper bound only for y, i.e., $T = \{y\}$.

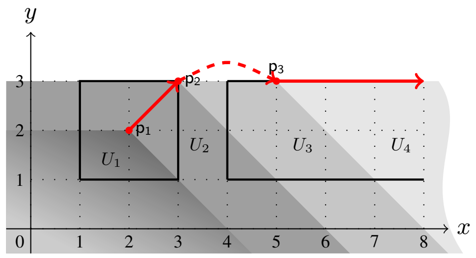


Figure 1. Illustration of SYMBA on a 2-D example.

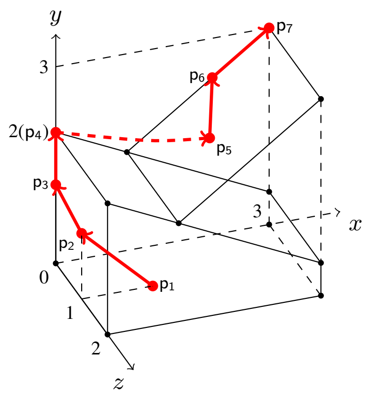


Figure 2. Illustration of SYMBA on a 3-D example.

***First* GLOBALPUSH** Similar to our previous example, SYMBA starts with $U \equiv$ *false* and $O \equiv$ *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, $U \equiv y \leqslant 0$.

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, $U \equiv y \leqslant 2$. 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 optimal solution $y \leqslant 3$.

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 (objective functions). In the following sections, we describe SYMBA formally and discuss our implementation and experimental results.

3. SYMBA: The Symbolic Optimization Algorithm

In this section, we provide definitions required for the rest of the paper and formalize SYMBA as a set of inference rules.

3.1 Definitions

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

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

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

We use $[\![\varphi]\!]$ to denote the set of all satisfying assignments (models) of φ. A *model* $p : \textit{Vars} \rightarrow \mathbb{R}$ of φ, denoted $p \models \varphi$, is a valuation of the variables of φ such that $\varphi(p) \equiv \textit{true}$, 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 *Atoms*(φ) to denote the set of all *Atoms* appearing in φ, and *Vars*(φ) to denote the set of all *Vars* appearing in φ.

Optimal Solutions Let φ be a formula in $\mathcal{L}$. Let $T = \{t_1, \ldots, t_n\}$ be a set of linear expressions, *objective functions*, where each t_i is of the form $c_1x_1 + \cdots + c_mx_m$, where $c_i \in \mathbb{R}$ and *Vars*$(\varphi) = \{x_1, \ldots, x_m\}$. The goal of SYMBA is to compute a vector $(k_1, \ldots, k_n)$, where each $k_i \in \mathbb{R} \cup \{\infty, -\infty\}$, such that for each t_i, $\varphi \Rightarrow t_i \leqslant k_i$ and there does not exist $k'_i < k_i$ where $\varphi \Rightarrow t_i \leqslant k'_i$. We say that $(k_1, \ldots, k_n)$ is the *optimal solution* of T w.r.t. φ, and denote it as $\mathsf{opt}_T(\varphi)$. We call each value k_i the *optimal value* (or the least upper bound) of t_i in φ.

Given such a vector $V = (k_1, \ldots, k_n)$, where $n = |T|$, we use $\mathsf{form}_T(V)$ to denote the formula $\bigwedge_{i \in [1,n]} t_i \leqslant k_i$. Given a model p of φ, we use p^T to denote the vector $(t_1(p), \ldots, t_{|T|}(p))$.

Given two vectors V_1 and V_2 of equal length, we use $\mathsf{min}(V_1, V_2)$ and $\mathsf{max}(V_1, V_2)$ to denote the pointwise minimum and maximum of the two vectors, respectively. We say $V_1 \leqslant V_2$ if each element of V_1 is less than or equal to its corresponding element in V_2, or if there exists a $-\infty$ in V_1. Intuitively, $V_1 \leqslant V_2$ *iff* $\mathsf{form}_T(V_1) \Rightarrow \mathsf{form}_T(V_2)$. Therefore, we say that V_2 is *weaker* than V_1 if $V_1 < V_2$ (or V_1 is *stronger* than V_2).

Combinations of Theories For clarity of presentation, we restrict ourselves to applying SYMBA to formulas in $\mathcal{L}$. It is important to note, however, that SYMBA is applicable to quantifier-free formulas over any combination of theories $\mathcal{T} \cup \mathcal{LRA}^{\leqslant}$, where $\mathcal{T}$ is an arbitrary combination of theories, and $\mathcal{LRA}^{\leqslant}$ is linear real arithmetic restricted to non-strict inequalities. The only restriction we require is that $\mathcal{T}$ and $\mathcal{LRA}^{\leqslant}$ have disjoint signatures. In other words, atomic formulas should be over $\mathcal{T}$ or $\mathcal{LRA}^{\leqslant}$, exclusively. For example, $\mathcal{T}$ can be the combination of the theories of bitvectors and arrays (perhaps for modelling program executions). The rest of our presentation can apply directly to SMT formulas over $\mathcal{T} \cup \mathcal{LRA}^{\leqslant}$ without any modification.

3.2 SYMBA Formalized

We now formalize the symbolic optimization algorithm SYMBA as a set of inference rules shown in Fig. 3.

Given a set of objectives $T = \{t_1, \ldots, t_n\}$ and a formula φ in $\mathcal{L}$, SYMBA computes $\mathsf{opt}_T(\varphi)$. 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 $\mathsf{opt}_T(\varphi)$ (i.e., $U \leqslant \mathsf{opt}_T(\varphi)$ is invariant); and O is an over-approximation of $\mathsf{opt}_T(\varphi)$ (i.e., $\mathsf{opt}_T(\varphi) \leqslant O$ is

$$\frac{}{\langle \emptyset, (-\infty,\dots,-\infty), (\infty,\dots,\infty)\rangle}\ \textsc{Init} \qquad \frac{p \models \varphi \wedge \neg\mathsf{form}_T(U)}{\langle M,U,O\rangle \to \langle M \cup \{p\}, \mathsf{max}(U, p^T), O\rangle}\ \textsc{GlobalPush}$$

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

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

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

Figure 3. Inference rules used by SYMBA.

invariant). Note that for clarity of presentation, we treated $\mathsf{opt}_T(\varphi)$, U, and O as formulas in Sec. 2, whereas here we treat them as vectors and use $\mathsf{form}_T(V)$ to convert a vector V to the formula it represents.

When SYMBA terminates, we know that $U = O = \mathsf{opt}_T(\varphi)$. Initially, as defined by the rule INIT, $M = \emptyset, U = (-\infty,\dots,-\infty)$, and $O = (\infty,\dots,\infty)$. The rules GLOBALPUSH, UNBOUNDED, and UNBOUNDED-FAIL are used to weaken U until it is equal to $\mathsf{opt}_T(\varphi)$, whereas BOUNDED strengthens O until it is equal to $\mathsf{opt}_T(\varphi)$.

GLOBALPUSH finds a model of φ that is not captured by $\mathsf{form}_T(U)$ (i.e., lies outside the under-approximation) and adds it to U to weaken it (using max). When the rule GLOBALPUSH no longer applies, we know that $U = \mathsf{opt}_T(\varphi)$. Note that applying this rule alone does not guarantee that U eventually reaches $\mathsf{opt}_T(\varphi)$ 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 objectives asymptotically.
2. GLOBALPUSH cannot detect whether an objective is unbounded, so it will keep finding models that increase the value of the unbounded objective indefinitely.

To that end, the rules UNBOUNDED and UNBOUNDED-FAIL are used to detect unbounded objectives and help GLOBALPUSH avoid asymptotic behavior. UNBOUNDED takes as parameters a model $p_1 \in M$ and an objective $t_i \in T$ 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 $p_3 \models \varphi$, $[p_2] \subset [p_3]$ 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. 3.3.

In addition to the aforementioned rules, the rule BOUNDED detects whether a model $p \in M$ exhibits the least upper bound of some objective t, 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 $\mathsf{opt}_T(\varphi)$ at every point of its execution. This makes SYMBA resilient to SMT solver failures and allows us to limit resource consumption when desired. That is, by prematurely terminating SYMBA during its execution, we can recover optimal values of some of the objective functions, as maintained by the over-approximation.

Example We illustrate the applications of the rules on the 2-D example from Sec. 2.1 and shown in Fig. 1. Assume that after the initial call to GLOBALPUSH, $M = \{\mathsf{p}_1 = (2,2)\}$, $\mathsf{form}_T(U) \equiv y \leqslant 2 \wedge x + y \leqslant 4$, and $\mathsf{form}_T(O) \equiv \mathit{true}$.

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

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, $\mathsf{form}_T(U)$ becomes $y \leqslant 3$, making GLOBALPUSH inapplicable. Therefore, $\varphi \Rightarrow \mathsf{form}_T(U)$, and $U = O = \mathsf{opt}_T(\varphi)$.

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

3.3 Soundness

We start by showing soundness of the UNBOUNDED rule.

A necessary and sufficient condition for proving that a given objective t is unbounded within φ is the existence of a convex polyhedron φ_c, e.g., a ray, 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. 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. 1). Thus, $x + y$ is unbounded. For another example, consider the point $\mathsf{p}_1 = (2, 2)$. Since 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 p_2, shown in Fig. 1, that lies on the boundaries $x = 3$ and $y = 3$.

Thm. 1 formalizes this construction using boundary classes and states its correctness for proving that an objective is unbounded in φ.

Theorem 1 (Soundness of UNBOUNDED). *Given a formula φ in $\mathcal{L}$ 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. Proofs are available in the appendix. □

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 $U \leqslant \mathsf{opt}_T(\varphi)$, since the optimal solution cannot have a least upper bound for t if it is unbounded (i.e., the least upper bound of t is ∞).

Theorem 2 (Soundness of SYMBA). *If* GLOBALPUSH *does not apply, i.e., $\varphi \wedge \neg\mathsf{form}_T(U) \Rightarrow false$, then $U = \mathsf{opt}_T(\varphi)$.*

Proof. Follows trivially from the invariant $U \leqslant \mathsf{opt}_T(\varphi)$. □

3.4 Termination

We now discuss sufficient conditions for ensuring termination of SYMBA. For simplicity of presentation, we assume that T contains a single objective 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 $a_i \in \{$GLOBALPUSH, UNBOUNDED, 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 a local maximum. Condition 2 ensures that we visit every local maximum by visiting every boundary 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. 2, 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 p_4 in Fig. 2, UNBOUNDED(-FAIL) are inapplicable. Condition 1 ensures that we eventually find a model outside the current under-approximation (see p_5), thus escaping the local maximum.

A *k-sequence* for an objective 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. 2, $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4$ is a k-sequence.

Since $[p_i]$ for a k-sequence strictly grows in size and the largest boundary class has size at most $|Atoms(\varphi)|$, a k-sequence is of length at most $k = |Atoms(\varphi)|$. Lemma 1 states that the last model p_k of a k-sequence always exhibits the largest value of t in its boundary class $[p_k]$.

Lemma 1. *Let φ be a formula, and t be an objective bounded in φ. Then, in every execution of* SYMBA*, the last element p_k in every k-sequence for t satisfies $t(p_k) = \max\{t(p) \mid p \models \varphi \wedge [p_k]\}$.*

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

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., $\varphi \Rightarrow \mathsf{form}_T(U)$.

Theorem 3. 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, infinitely many GLOBALPUSH creates infinitely many k-sequences. Following Lemma 1, there are infinitely many models p in the execution sequence such that $p \models \varphi$ and $t(p) = \max\{t(p') \mid p' \models \varphi \wedge [p]\}$. 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 boundary 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 boundary class (due to the finite number of boundary classes) such that the three conditions in Thm. 1 hold. After that, GLOBALPUSH becomes inapplicable. □

4. Implementation and Evaluation

4.1 Implementation

We have implemented SYMBA in C++, using the Z3 SMT solver [21] for satisfiability queries. Our implementation accepts a formula φ and a set of objectives T written in the standard SMT-LIB2 [7] format. It then computes the optimal solution $\mathsf{opt}_T(\varphi)$ and returns the result. We have made available the executable and benchmarks online.

Detecting Unbounded Objectives 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. 4 shows the procedure UNBOUNDEDIMPL: our implementation of the UNBOUNDED rules. We assume that there is a global SMT *context* in which the formula φ has been asserted. An active boundary class c and a objective t_i are passed in as parameters. $U[t_i]$ and $O[t_i]$ refer to the i-th element of the vectors U and O, respectively. SAT and UNSAT refer to the current state of the SMT context, and GETMODEL() returns a model satisfying the current

```
1:  function UNBOUNDEDIMPL($c \in \mathcal{P}(\mathcal{E}(\varphi)), t_i \in T$)
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:          POP(); return
9:      ASSERT($\bigvee(\mathcal{E}(\varphi) \setminus c)$)
10:     if SAT then                                ▷ UNBOUNDED-FAIL
11:         POP(); return GETMODEL()
12:     else                                       ▷ UNBOUNDED
13:         $U[t_i] \leftarrow \infty$
14:     POP(); return
```

Figure 4. Implementation of UNBOUNDED(-FAIL).

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 boundary class as p_1 (line 6). If no such p'_2 exists, then neither UNBOUNDED nor UNBOUNDED-FAIL applies. Given that p'_2 exists, we check for the existence of p_3 in a stronger boundary class (lines 9-13). If p_3 exists, we apply UNBOUNDED-FAIL; otherwise, we apply UNBOUNDED.

Scheduling Policy Our implementation is a scheduling of SYMBA's rules (Fig. 3) that satisfies the fairness conditions (in Sec. 3.4).

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 UNBOUNDED(-FAIL) until either UNBOUNDED is applicable (in which case the objective 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 variations of the scheduling policy described above, we instrumented our implementation with a parameter *balance* $\in (0, 100]$ which ensures that UNBOUNDEDIMPL does not take more than *balance*% of the total execution time. Specifically, during execution, if UNBOUNDEDIMPL has taken more than *balance*% of the elapsed time, SYMBA switches to applying the GLOBALPUSH rule until the time taken by UNBOUNDEDIMPL so far is less than *balance*% of the elapsed time. Intuitively, when *balance* is 100, the deterministic schedule described above is in effect.

Optimizations Another effective optimization 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 $\propto$: $Atoms(\varphi) \times Atoms(\varphi)$ as follows: $P \propto P'$ if and only if

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

where $Vars(P)$ is the set of variables appearing in P.

We then define *the boundary 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.

STATISTIC	AVG.	MAX.	MIN.	STD.
# of Variables	882	19,170	40	1,647
# of Objectives	56	386	20	15
# of Nodes in DAG	7,278	127,987	1,121	10,619

Table 1. Aggregate statistics of our 1,065 benchmarks.

4.2 Experimental Evaluation

Our experimental evaluation is designed to compare SYMBA against other symbolic optimization techniques, and to assess the effectiveness of our different implementation heuristics.

We conducted two classes of experiments: (1) a comparison with existing SMT-based optimization techniques [56]; and (2) an evaluation of the effects of different implementation heuristics, as well as information reuse among multiple objectives, on the efficiency of SYMBA. We describe these in detail below.

Benchmarks As discussed in Sec. 1, one possible application of optimization is computing abstract transformers for numerical abstract domains. We have incorporated SYMBA into the UFO program analysis and verification framework [3] and used it as an abstract transformer (abstract post operator) for the family of *Template Constraint Matrix* (TCM) domains [54]. A TCM domain is parameterized by a set of *templates* $T = \{t_1, \ldots, t_n\}$, which are linear expressions over program variables. Given an abstract state φ_{pre} describing a set of initial (pre) states and a loop-free program fragment encoded as a formula φ_{lf}, the best (most precise) abstract transformer for a TCM domain computes the strongest formula $\bigwedge t_i \leqslant k_i$ that is implied by $\varphi_{pre} \wedge \varphi_{lf}$. Thus, we can use SYMBA to compute the best abstract transformer by simply computing $\mathsf{opt}_T(\varphi_{pre} \wedge \varphi_{lf})$. Note that the TCM domains subsume a number of popular domains, including intervals, octagons, octahedra, etc. For instance, by setting T to all live variables and their negation at the destination program location, then we get an intervals domain, since the result of SYMBA can be interpreted as the minimum and maximum value of each program variable after executing the program fragment denoted by φ_{lf}.

We generated our benchmarks from a set of C programs used in the 2013 Software Verification Competition (SV-COMP) [10]. The programs cover a range of software, from Linux and Windows device drivers to models of SSH and sequentialized concurrent SystemC programs.[3] We narrowed the set down to 604 C programs that were not trivially discharged (proved correct or incorrect) by UFO. We instrumented UFO to record abstract post queries in SMT-LIB2 format, and collected 10K+ 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* [11, 32]). For the set of objectives, we used all variables (as well as their negation) that are in scope at the destination cutpoint (i.e., an intervals domain). From the generated queries, we selected the hardest 1,065 benchmarks for evaluation (which took SYMBA more than 0.5s to process). Table 1 shows the average, maximum, minimum, and standard deviation, of the number of variables, objective functions, and nodes in the DAG representation of the formulas in our benchmarks. We conducted all of our experiments on a machine running Linux with an Intel i5 3.1GHz processor and 4GB of RAM.

[3] We drew 2,000+ programs from the following SV-COMP categories: `ControlFlowIntegers`, `SystemC`, `ProductLines`, and `DeviceDrivers64`.

Comparing with Existing Tools To the best of our knowledge, the work of Sebastiani and Tomasi [56] is the only other SMT technique that addresses the problem of finding optimal assignments for LRA objective functions. At a high level, the technique works as follows:

A Sample a satisfiable disjunct φ_d from a given formula φ using an SMT solver.

B Since φ_d is a conjunction of atoms, the linear arithmetic atoms in φ_d represent a convex polyhedron. So, use any *linear programming* (LP) solver to find the optimal value of a given objective function within the given disjunct.

C Check, using an SMT solver, whether the result is optimal for all of φ. If not, go back to step **A** and sample a new disjunct. The process is guaranteed to terminate since there are finitely many disjuncts.

We have acquired a binary of the implementation described in [56] from the authors. Their tool is called OPT-MATHSAT, as it is built in the MATHSAT SMT solver [**?**]. There are two issues threatening the validity of a direct comparison between SYMBA and OPT-MATHSAT: (1) OPT-MATHSAT accepts a single objective function, meaning that we have to call it multiple times per benchmark, each time with a different objective. Since we do not have access to its source code, we cannot tell if multiple calls to OPT-MATHSAT incur significant pre-processing overhead. (2) Our implementation of SYMBA uses Z3 as its underlying SMT solver, whereas OPT-MATHSAT uses MATHSAT.

In order to avoid these issues and establish a fairer comparison, we have implemented two versions of the *linear search* (LS) algorithm proposed in [56] and implemented in OPT-MATHSAT:

1. LS(OPT-Z3): By accessing the available Z3 source code [1], we modified the linear arithmetic solver of Z3 to allow optimization of LRA objectives in the satisfying assignments. A DPLL(T) [25] solver like Z3 lazily finds propositionally satisfiable disjuncts (conjunctions of atoms) from a given formula φ, and uses a (theory) T-solver to decide the satisfiability of the atoms (in our case linear constraints) [23]. We instrumented Z3's linear arithmetic solver such that it does not terminate immediately after the first satisfying assignment is found, but finds an optimal satisfying assignment for a given objective function. This was implemented using the standard incremental simplex solving procedure under exact (rational) representation. We call the modified tool OPT-Z3. LS(OPT-Z3) is an implementation of LS in Z3 that uses OPT-Z3 as its LP solver. This is analogous to the implementation described in [56], where a modified version of MATHSAT's linear arithmetic solver is used as the LP solver.
2. LS(LP) uses an off-the-shelf LP solver as its convex optimization engine. We have chosen two well-known open source LP libraries: the GNU Linear Programming Kit (GLPK) [39] and the Sequential Object-oriented Simplex (SOPLEX) [62] [4].

Both versions of LS use Z3 for sampling disjuncts and checking optimality (i.e., steps **A** and **C** above), but different LP solvers for finding optimal values (step **B**). Unlike OPT-MATHSAT, both LS(OPT-Z3) and LS(LP) accept multiple objective functions, and optimize them simultaneously. Specifically, in step **B**, they make multiple calls to the LP solver to find an optimal value for each objective function.

[4] We used GLPK v4.51 custom-built. The solver implements the primal two-phase simplex method based on floating point arithmetic. We used SOPLEX v1.7.1, which uses floating point arithmetic.

SYMBA ***Configurations*** We use the following SYMBA configurations:

1. SYMBA(X): SYMBA with different scheduling policies, where X specifies the value of *balance*.
2. SYMBAONEOBJ: Same as SYMBA(100), but optimizes a single objective at a time. That is, execution is restarted from scratch for each objective function.
3. SYMBA$(X)^{\text{OPT-Z3}}$: Same as SYMBA(X), but uses Z3 with the modified linear arithmetic solver OPT-Z3 (we describe this in more detail below).

Results: **SYMBA** ***vs. Other Techniques*** Fig. 5(a) shows the results of running SYMBA(100) vs. OPT-MATHSAT on the 1,065 SMT-LIB2 benchmarks with a timeout of 100 seconds per benchmark. Each point on the graph represents a benchmark. The axes correspond to the CPU time (measured in seconds—log scale) taken by SYMBA(100) (x-axis) and OPT-MATHSAT (y-axis). The points above the diagonal represent problems where SYMBA is faster. Points at the top right corner are cases where both OPT-MATHSAT and SYMBA(100) cannot complete the benchmark in the allotted 100 seconds[5]. OPT-MATHSAT has 13 timeouts vs. 19 timeouts by SYMBA(100). Our results clearly show that SYMBA(100) outperforms OPT-MATHSAT on our set of benchmarks in most cases. The average and maximum speed up of SYMBA(100) vs. OPT-MATHSAT are 2.2x and 10.4x, respectively.

We now compare SYMBA against our multi-objective-function implementation of the linear search algorithm employed by OPT-MATHSAT. Fig. 5(b) compares the execution times of SYMBA(100) vs. LS(OPT-Z3). The results clearly demonstrate the superior performance of LS(OPT-Z3): most benchmarks are solved in less time by LS(OPT-Z3), often by an order of magnitude. LS(OPT-Z3) has 7 timeouts.

To understand the reasons behind these performance differences, we took a closer look at the benchmarks where SYMBA(100) is significantly slower than LS(OPT-Z3). We noticed two recurring problems for SYMBA on these benchmarks: (1) the majority of the time is spent in the UNBOUNDEDIMPL function, indicating the expensive nature of UNBOUNDEDIMPL calls and ineffectiveness of our scheduling strategy when *balance*=100; (2) applications of GLOBALPUSH make very small expansions to the under-approximation U, indicating that we prefer *guided* applications of GLOBALPUSH. To address point (1), we ensured that UNBOUNDEDIMPL does not take more than 40% of the execution time by setting *balance* to 40. To address point (2), we considered applying GLOBALPUSH using OPT-Z3 (described above) instead of Z3. That is, instead of GLOBALPUSH asking Z3 for a point that lies outside the under-approximation U, we made GLOBALPUSH supply OPT-Z3 with one of the objective functions, and whenever the Z3's DPLL(T) solver finds a satisfiable disjunct with a point outside the under-approximation, it finds a satisfying assignment that maximizes that objective function within that disjunct. This causes the GLOBALPUSH rule to produce models that are *farther away* from the current under-approximation, expediting convergence. We call this configuration SYMBA$(40)^{\text{OPT-Z3}}$.

Fig. 5(c) compares the execution times of SYMBA$(40)^{\text{OPT-Z3}}$ vs. LS(OPT-Z3). The results now show that SYMBA$(40)^{\text{OPT-Z3}}$ outperforms LS(OPT-Z3) on 81% of the benchmarks, with a 5.0x maximum speedup and a 1.4x average speedup per benchmark. Moreover, SYMBA$(40)^{\text{OPT-Z3}}$ solves all 1,065 benchmarks without timing out. We have also run the two other configurations of lin-

[5] Since OPT-MATHSAT accepts a single objective at a time, we invoked it multiple times per benchmark, giving it a timeout of 100 seconds per objective. If the total time (for all objectives) taken for a benchmark is more than 100 seconds, OPT-MATHSAT is considered to have timed out.

ear search, LS(GLPK) and LS(SoPlex). They exhibit similar behaviour to LS(Opt-Z3), but are slightly slower. We cross-checked all results produced by different tools (when they do not timeout) and all of them match.

Results:* Symba*'s Configurations Table 2 summarizes the results of running all the aforementioned algorithms and configurations on the same set of benchmarks with a timeout of 100 seconds per benchmark. The results of running Symba(100) are summarized in row 1 of Table 2. Symba(100) was able to solve 1,046 out of 1,065 benchmarks in 3,841 seconds. In the process, it made $\sim$395K SMT queries using 24,387 invocations of GlobalPush and 164,156 invocations of UnboundedImpl.

Rows 2-3 capture the results of running Symba(X), where X is 60 and 20, respectively. When X is 60 (time spent in UnboundedImpl is restricted to 60% of the total time) the number of GlobalPush calls goes up by about 400%. Time is spent in making unguided discovery rather than big leap towards the goal. This even affects UnboundedImpl, the number of calls slightly increases since more points are sampled. When X is 20, it was only able to solve 766 benchmarks, for which the number of calls to GlobalPush goes above 130K while the number of calls to UnboundedImpl drops to $\sim$42K. Our experiments show that 100 is the best value for *balance* when running Symba(X)[6]. Conversely, when running Symba$(X)^{\text{Opt-Z3}}$, we found that we greatly benefit from a lower *balance* value (*balance*=40 gives us best performance), since there the GlobalPush rule can discover unbounded objectives, alleviating the pressure on UnboundedImpl.

SymbaOneObj (see Row 4 of Table 2) was able to solve 1,045 problems in 6,867 seconds. SymbaOneObj uses the same configuration as Symba(100) except that it finds solutions for multiple objectives independently, without reusing models amongst different objectives (as Symba does). Using SymbaOneObj causes the number of SMT queries to go up by 15% and the number of GlobalPush calls to increase by 300%. Optimizing multiple objective functions simultaneously ensures that all objectives benefit from the sampled models and potentially avoids repeating expensive SMT calls.

Summary The experiments compare our proposed SMT-based symbolic optimization algorithm with existing techniques and highlight the effectiveness of various implementation heuristics and optimizations. We compared Symba with OPT-MathSAT as well as two LP based implementations of its algorithm on a large set of benchmarks generated from program analysis tasks. The results demonstrate the power of Symba's approach. A configuration that employs both efficient scheduling policy and convex optimization outperforms them all and solves all the benchmarks. Our experiments also demonstrate the importance of Symba's multi-objective-function capability.

5. Related Work

Our work intersects with different areas of research. In this section, we compare Symba with (1) other optimization techniques in SAT and SMT solvers; (2) optimization techniques employed within the context of abstract interpretation [20]; (3) linear programming techniques; and (4) classification techniques from the machine learning community.

Optimization in SAT/SMT Within the SMT and SAT solving arena, numerous forms of optimization have been proposed (e.g., MAX-SAT/SMT in Yices [**?**]). The closest to Symba is the recent of work of Sebastiani and Tomasi [56]. Similar to Symba, they propose an SMT-based solution for optimizing objective functions in LRA, and to the best of our knowledge, this is the only other work that addresses this problem. As explained in Sec. 4, this approach works by sampling a satisfiable disjunct (convex polyhedron) from a given formula using the SMT solver as a black box. Then, using any LP solver, it finds the optimal value of an objective function in that disjunct. It then checks if the optimal value is globally optimal (again, using the SMT solver); otherwise, more disjuncts are sampled. Effectively, this approach lazily builds the DNF of a formula until the disjunct with the optimal value of a given objective function is found. Compared to [56], Symba does not require an off-the-shelf LP solver (or access to a modified theory solver within the SMT solver [56]) Instead, Symba offers a simple and elegant optimization algorithm that can be easily implemented on top of existing SMT solvers, without using any external tools. Moreover, Symba can simultaneously optimize multiple objective functions. Symba also maintains an over-approximation of the optimal solution, allowing us to prematurely terminate it and still retrieve optimal values for a subset of the objective functions. As we show in Sec. 4, the multi-objective-function feature can also be implemented within the approach of [56]. Both Symba and [56] can apply to formulas over mixed theories.

In [17], Cimatti et al. proposed a theory of *costs* for augmenting an SMT solver with pseudo-boolean (PB) constraints [9]. At a high level, their theory allows associating a cost with individual Boolean constraints. The goal then is to find a satisfying assignment that minimizes/maximizes the total cost. Thus, the theory of costs enables encoding *weighted MAX-SAT/SMT* problems – in fact, the two problems are equivalent. It is easy to see that Symba subsumes the weighted MAX-SMT problem. For example, we can associate with each Boolean constraint B_i a cost c_i, and add a constraint $\mathsf{ite}(B_i, c_i = k_i, c_i = 0)$, where $k_i \in \mathbb{R}$. Then, our objective is to minimize or maximize the sum of all costs $c_1 + \ldots + c_n$.

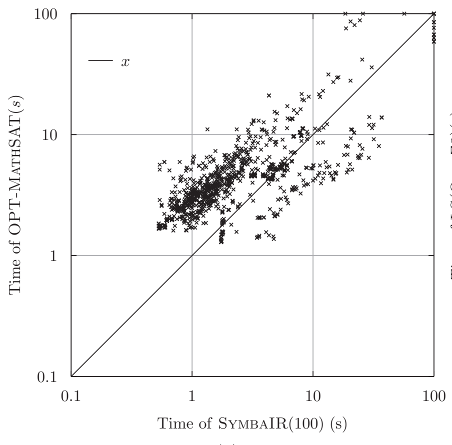
Another form of optimization in the SMT framework is that of Nieuwenhuis and Oliveras [46]. Their optimization technique extends the traditional DPLL(T) algorithm for SMT solving with extra rules for strengthening the theory T midway during execution, e.g., a theory solver for linear arithmetic might be strengthened to find only assignments with values less than 10. This allows guiding the solver towards satisfying assignments that maximize or minimize certain objective functions. The authors show how to implement weighted MAX-SMT in their general framework. It is unclear to us if their theory strengthening approach can be used to implement an optimization procedure for LRA objective functions (as in this paper). Moreover, unlike Symba, their approach is not easily implementable, as it requires deep modifications to an SMT solver.

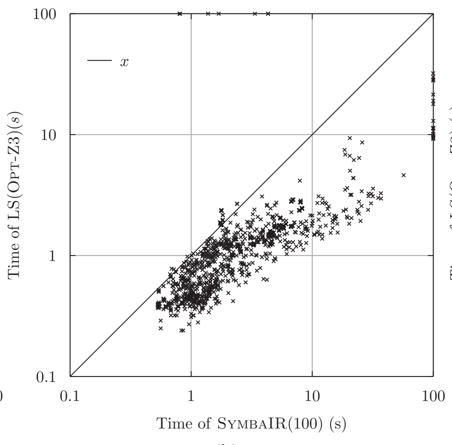
In more recent work [16], Chaganty et al. consider the problem of finding *most likely* satisfying assignments in the presence of probabilistic constraints. There, an SMT solver is used to handle axioms, i.e., constraints with 0 or 1 probabilities, and a *relational solver* (e.g., [47]) is used to handle other probabilistic constraints. It is important to note that the relational solver can be replaced by a weighted MAX-SMT solver (as noted in [16]). Thus, Symba can be directly applicable to solving such problems. We leave this interesting direction for future work.

Optimization in Abstract Interpretation Numerical abstract domains have been an active subject of research due to their importance in different program analyses. As discussed earlier, an important operation in such domains is the abstract transformer (post), which can often be phrased as an optimization problem over formulas in LRA. As a result, optimization has been a subject of interest within the program analysis community.

For example, Monniaux [43] proposed an algorithm for computing best abstract transformers of *Template Constraint Matrix* (TCM) domains [54] over loop-free code. As discussed in Sec. 4,

[6] We omit Symba(40) and Symba(80) from the table as they exhibit similar performance to Symba(60).





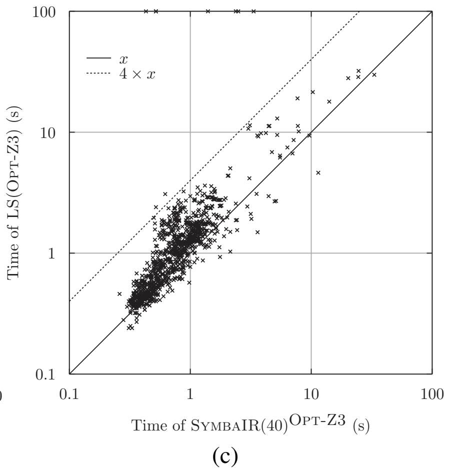


Figure 5. Performance comparison between (a) SYMBA(100) vs. OPT-MATHSAT, (b) SYMBA(100) vs. LS(OPT-Z3), and (c) SYMBA(40)$^{\text{OPT-Z3}}$ vs LS(OPT-Z3).

	CONFIGURATION	TOTAL TIME(s)	# SMT QUERIES	# SOLVED	# GLOBALPUSH	# UNBOUNDEDIMPL
1	SYMBA(100)	3,841	394,579	1,046	24,387	164,156
2	SYMBA(60)	5,720	577,068	1,015	120,112	179,278
3	SYMBA(20)	2,716	231,906	766	132,051	42,227
4	SYMBAONEOBJ	6,867	445,181	1,045	83,421	162,796
5	SYMBA(40)$^{\text{OPT-Z3}}$	1,087	84,814	1,065	7,007	51,898
6	OPT-MATHSAT	5,992	-	1,052	-	-
7	LS(OPT-Z3)	1,521	20,829 [a]	1,058	-	-
8	LS(GLPK)	3,098	20,854	1,063	-	-
9	LS(SOPLEX)	2,791	20,920	1,065	-	-

[a] We do not count calls to LP solver (including OPT-Z3) in step **B**.

Table 2. Overall results for different SYMBA and LS configurations, as well as OPT-MATHSAT, on the 1065 SMT-LIB2 benchmarks.

TCM domains are parameterized by a set of templates T, or objective functions. Loop-free program segments, along with a set of initial states, can be encoded as a formula φ in LRA. Then, the problem of computing the *best* abstract transformer becomes that of computing $\mathsf{opt}_T(\varphi)$. This problem is known in the program analysis community as *symbolic abstraction*. The approach in [43] uses quantifier elimination, which is admissible in LRA. In later work, Monniaux and Gonnord [45] attack the problem from a different angle, using an SMT solver to sample disjuncts φ_d of φ, and, using off-the-shelf solvers, to find the optimal solution $\mathsf{opt}_T(\varphi_d)$ of each disjunct. Surprisingly, this is done in the same manner as proposed by Sebastiani and Tomassi [56] for LRA optimization in SMT (discussed above).

Recently, Thakur and Reps [60] proposed a generalization of Stålmarck's SAT solving method [57] to richer logics. The algorithm attempts to prove a formula φ unsatisfiable by iteratively refining an over-approximation of φ starting from *true* until arriving at *false*. They showed how the algorithm can be instantiated with abstract domains, such as polyhedra, and used to compute best abstractions of formulas in LRA within the given abstract domain. Thus, by instantiating their algorithm with a TCM domain, we can compute $\mathsf{opt}_T(\varphi)$ for φ in LRA. Their approach is a general framework for symbolic abstraction that is applicable to a wide range of logics and abstract domains.

In contrast to these techniques, our goal with SYMBA is to bring LRA optimization to an SMT setting, leveraging the power and generality of SMT solvers, and making optimization directly usable by researchers who are already familiar with, and actively using, SMT solvers. Additionally, with SYMBA, we do not only find the optimal value of a given objective function, but also the satisfying assignment that results in such a value (found in the set of models M). This is a crucial requirement when, e.g., searching for optimal counterexample witnesses, where we need a trace of the counterexample (i.e., an optimal satisfying assignment) and not the optimal value of the given objective function.

Linear Programming In the field of linear programming, the optimization problem over conjunctions and disjunctions of convex polyhedra (as in our setting) has been known as *Linear Disjunctive Programming* (LDP) [4]. Later, *Linear Generalized Disjunctive Programming* (LGDP) [50] was proposed; there, Boolean variables are used to explicitly model discrete decisions. LDP and LGDP are equivalent to SYMBA when it is applied to formulas over LRA and propositional variables. Most notable approaches for solving LGDP problems carefully convert the problem to a *Mixed Integer Linear Programming* (MILP) problem, e.g., [55], and use existing MILP solvers to solve it. In comparison to such techniques, SYMBA can handle formulas over arbitrary theories (in combination with LRA), and can simultaneously optimize multiple objective functions. Additionally, SYMBA uses infinite precision rational arithmetic employed by SMT solvers, whereas LGDP solvers tend to use floating point arithmetic, potentially losing precision.

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) [49], 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 (objective functions), that does not misclassify any of the positive examples (i.e., is implied by 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 [51], weakening an under-approximation by sampling more points is analogous to the approach of the simple learning algorithm Find-S [42]. Find-S gradually weakens a classifier, starting from *false*, by iteratively taking into account more and more positive examples.

6. Conclusions and Future Work

We proposed SYMBA, an efficient SMT-based optimization algorithm for objective functions stated in the theory of linear real arithmetic. SYMBA utilizes efficient SMT solvers as black boxes, making it easy to implement without requiring modifications to existing intricate SMT solver implementations, and enabling it to directly benefit from future advances in SMT solving. We have evaluated SYMBA on benchmarks drawn from abstract transformer computations for numerical abstract domains. Our thorough experimental evaluation indicates the advantages of our approach over other proposed techniques.

We see many avenues for future work. First, the most natural next step is extending SYMBA to integer arithmetic objective functions. We believe this can be done by augmenting SYMBA's rules with new ones that introduce Gomory cuts (cutting planes) [**?**] in order to prune infeasible solutions. Another interesting direction is handling non-linear objective functions, in order to model complex cost functions. From an engineering perspective, we would also like to study efficient parallel implementations of SYMBA's rules.

References

[1] Z3 Source Code Repository. `http://z3.codeplex.com/`.

[2] A. Albarghouthi and K. L. McMillan. Beautiful Interpolants. In *Proc. of CAV'13*, LNCS, 2013.

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

[4] E. Balas. Disjunctive programming: Properties of the convex hull of feasible points. *Discrete Applied Mathematics*, 89(1–3):3 – 44, 1998.

[5] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proc. of CAV'01*, volume 2102 of *LNCS*, pages 260–264, 2001.

[6] C. Barrett and C. Tinelli. CVC3. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 298–302, July 2007.

[7] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[8] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[9] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institute für Informatik, 1995.

[10] D. Beyer. Competition On Software Verification - (SV-COMP). In *Proc. of TACAS'12*, volume 7214 of *LNCS*, pages 504–524, 2012. URL `http://sv-comp.sosy-lab.org/`.

[11] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *Proc. of FMCAD'09*, pages 25–32, 2009.

[12] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic Subtyping with an SMT Solver. *J. Funct. Program.*, 22(1): 31–105, 2012.

[13] N. Bjorner, K. McMillan, and A. Rybalchenko. Program Verification as Satisfiability Modulo Theories. In *Proc. of SMT'12 workshop*, 2012.

[14] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *Proc. of CAV'08*, pages 299–303, 2008.

[15] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of OSDI'08*, pages 209–224, 2008.

[16] A. Chaganty, A. Lal, A. Nori, and S. Rajamani. Combining Relational Learning with SMT Solvers using CEGAR. In *Proc. of CAV'13*, LNCS, 2013.

[17] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *Proc. of TACAS'10*, pages 99–113, 2010.

[18] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 168–176, March 2004.

[19] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proc. of the Colloque sur la Programmation*, 1976.

[20] 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*, pages 238–252, 1977.

[21] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, LNCS, pages 337–340, 2008.

[22] R. DeLine and K. R. M. Leino. BoogiePL: A Typed Procedural Language for Checking Object-oriented Programs. Technical report, Microsoft Research, 2005.

[23] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] S. Falke, F. Merz, and C. Sinz. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM - (Competition Contribution). In *Proc. of TACAS'13*, pages 623–626, 2013.

[25] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Dpll(t): Fast decision procedures. In *Proc. of CAV'04*, LNCS, pages 175–188, 2004.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of PLDI'05*, pages 213–223, 2005.

[27] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proc. of POPL'10*, pages 43–56, 2010.

[28] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM*, 55(3):40–44, 2012.

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

[30] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-Free Programs. In *Proc. of PLDI'11*, pages 62–73, 2011.

[31] A. Gupta, C. Popeea, and A. Rybalchenko. Solving Recursion-Free Horn Clauses over LI+UIF. In *Proc. of APLAS'11*, pages 188–203, 2011.

[32] A. Gurfinkel, S. Chaki, and S. Sapra. Efficient Predicate Abstraction of Program Summaries. In *Proc. of NFM'11*, volume 6617 of *LNCS*, pages 131–145, 2011.

[33] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program Analysis via Satisfiability Modulo Path Programs. In *Proc. of POPL'10*, pages 71–82, 2010.

[34] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of POPL'02*, pages 58–70, 2002.

[35] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *Proc. of POPL'04*, pages 232–244, 2004.

[36] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-Guided Component-Based Program Synthesis. In *Proc. of ICSE'10*, pages 215–224, 2010.

[37] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as Control. In *Proc. of POPL'12*, pages 151–164, 2012.

[38] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. of LPAR'10*, pages 348–370, 2010.

[39] A. Makhorin. The GNU Linear Programming Kit. http://www.gnu.org/software/glpk/, 2000.

[40] H. Massalin. Superoptimizer: a Look at the Smallest Program. *SIGARCH Comput. Archit. News*, 15(5):122–126, Oct. 1987.

[41] A. Miné. The Octagon Abstract Domain. *J. Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

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

[43] D. Monniaux. Automatic modular abstractions for linear constraints. In *Proc. of POPL'09*, pages 140–151, 2009.

[44] D. Monniaux. Automatic Modular Abstractions for Template Numerical Constraints. *Logical Methods in Computer Science*, 6(3), 2010.

[45] D. Monniaux and L. Gonnord. Using Bounded Model Checking to Focus Fixpoint Iterations. In *Proc. of SAS'11*, LNCS, pages 369–385, 2011.

[46] R. Nieuwenhuis and A. Oliveras. On sat modulo theories and optimization problems. In *SAT*, pages 156–169, 2006.

[47] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: scaling up statistical inference in markov logic networks using an rdbms. *Proc. of VLDB*, 4 (6):373–384, Mar. 2011.

[48] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV*, 2013.

[49] J. C. Platt. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. In *Advances in Kernel Methods*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-19416-3.

[50] R. Raman and I. Grossmann. Modelling and computational techniques for logic based integer programming. *Computers and Chemical Engineering*, 18(7):563 – 578, 1994. ISSN 0098-1354.

[51] T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*, 2004.

[52] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid Types. In *Proc. of PLDI'08*, pages 159–169, 2008.

[53] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. *J. Symb. Comput.*, 45(11):1212–1233, 2010.

[54] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable Analysis of Linear Systems using Mathematical Programming. In *Proc. of VMCAI'05*, pages 25–41, 2005.

[55] N. W. Sawaya and I. E. Grossmann. A cutting plane method for solving linear generalized disjunctive programming problems. *Computers Ad Chemical Engineering*, 29(9):1891 – 1913, 2005. ISSN 0098-1354.

[56] R. Sebastiani and S. Tomasi. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ Cost Functions. In *Proc. of IJCAR'12*, pages 484–498, 2012.

[57] M. Sheeran and G. Stååmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Formal Methods in System Design*, 16: 23–58, 2000.

[58] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proc. of ASPLOS'06*, pages 404–415, 2006.

[59] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. of ICFP'11*, pages 266–278, 2011.

[60] A. V. Thakur and T. W. Reps. A Method for Symbolic Computation of Abstract Operations. In *Proc. of CAV'12*, pages 174–192, 2012.

[61] A. V. Thakur, M. Elder, and T. W. Reps. Bilateral Algorithms for Symbolic Abstraction. In *Proc. of SAS'12*, pages 111–128, 2012.

[62] R. Wunderling. *Paralleler und Objekt-orientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

7. Appendix

Lemma 2. *Given an $\mathcal{L}$ formula φ_c defining a convex polyhedron (conjunction of linear constraints), 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:

$$\mathbf{A}\vec{x} = \vec{b} \tag{1a}$$

$$\mathbf{C}\vec{x} \geqslant \vec{d} \tag{1b}$$

By definition, p satisfies the system and Eq. 1a is a subset of $[p]$ according to the definition of boundary class. Because $[p] \subseteq [p']$, p' must satisfy Eq. 1a. Suppose $p' \not\models \varphi_c$. Then there exists some k such that $\vec{c_k} \cdot \vec{p} > d_k$ and $\vec{c_k} \cdot \vec{p'} < d_k$

Note that $\vec{c_k} \cdot \vec{p} \neq d_k$ since $\vec{c_k} \cdot \vec{p'} \neq d_k$ and $[p] \subseteq [p']$. We now show that there exists a point $\vec{p_k} = \alpha\vec{p} + (1-\alpha)\vec{p'}$ $(0 < \alpha < 1)$ in the convex hull of p and p' s.t. $\vec{c_k} \cdot \vec{p_k} = d_k$. Since $t(p) \leqslant t(p')$, it is easy to show that $t(p) \leqslant t(p_k)$. Let $\vec{c_k} \cdot \vec{p} = d_k + \delta_1$, $\vec{c_k} \cdot \vec{p'} = d_k - \delta_2$ (where $\delta_1 > 0$ and $\delta_2 > 0$), and $\alpha = \frac{\delta_2}{\delta_1 + \delta_2}$. We have $\vec{c_k} \cdot \vec{p_k} = d_k$. We know that $[p] \subset [p_k]$ because p_k is in the convex hull of p and p' and $\{\vec{c_k} \cdot \vec{x} = d_k\} \notin [p] \wedge \{\vec{c_k} \cdot \vec{x} = d_k\} \in [p_k]$.

Suppose $p_k \not\models \varphi_c$, i.e., there exists j s.t. $\vec{c_j} \cdot \vec{p_k} < d_j$, we then repeat the above process. Each point p_i found satisfies $t(p) \leqslant t(p_i) \wedge [p] \subset [p_i]$. We are guaranteed to find a point p'' in the convex hull s.t. it is also within φ_c, because in each step we satisfy at least one more inequality. This contradicts the condition $\nexists p'' \models \varphi_c \cdot t(p) \leqslant t(p'') \wedge [p] \subset [p'']$. Therefore, $p' \models \varphi_c$. □

Proof of Thm. 1 ($\Leftarrow$) 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 $p^* \models \varphi$ s.t. $t(p^*)$ is the upper bound for t in φ. We show that there is always a point $p_2' \models \varphi$ s.t. $t(p_2') > t(p^*)$.

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

Let the formula φ' define a convex polyhedron s.t. $p_2 \models \varphi'$ and $\varphi' \Rightarrow \varphi$. Let $\varphi_c \equiv \varphi' \wedge \bigwedge[p_2]$. We know the following:

1. φ_c defines a 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 2, 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 $p_1, p_2 \models \varphi$ s.t. $[p_1] = [p_2]$, $t(p_2) > t(p_1)$, and $[p_1]$ is the most restrictive boundary class within which t is unbounded (i.e., t is unbounded in $\varphi \wedge \bigwedge[p_1]$, and there does not exist a boundary class $c \supset [p_1]$ s.t. t is unbounded in $\varphi \wedge \bigwedge c$). We know that such a class exists because t is unbounded in φ (otherwise t is bounded in every boundary class and φ is bounded). In other words, since (1) there are infinitely many models of φ with increasing values of t and (2) finitely many boundary classes, there has to be a boundary class $[p_1]$ s.t. t is unbounded in $\varphi \wedge \bigwedge[p_1]$ and there doesn't exist a class $c \supset [p_1]$ where t is unbounded in $\varphi \wedge \bigwedge c$.

If there are no classes $c \supset [p_1]$, or for every $c \supset [p_1]$, $\varphi \wedge \bigwedge c \Rightarrow$ *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 stronger 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 $p_1', p_2' \models \varphi$ s.t. $m < t(p_1') < t(p_2')$ and $[p_1] = [p_1'] = [p_2']$. As a result, p_1' and p_2' satisfy the three conditions in the theorem. $\square$