Synthesizing Invariants for Polynomial Programs by Semidefinite Programming

HAO WU, Institute of Software, Chinese Academy of Sciences& University of CAS, China
QIUYE WANG, Institute of Software, Chinese Academy of Sciences& University of CAS, China
BAI XUE, Institute of Software, Chinese Academy of Sciences& University of CAS, China
NAIJUN ZHAN, Institute of Software, Chinese Academy of Sciences& University of CAS, China
LIHONG ZHI, Institute of System Science, Chinese Academy of Sciences& University of CAS, China
ZHI-HONG YANG, Central South University, China

Constraint-solving-based program invariant synthesis involves taking a parametric template, encoding the invariant conditions, and attempting to solve the constraints to obtain a valid assignment of parameters. The challenge lies in that the resulting constraints are often non-convex and lack efficient solvers. Consequently, existing works mostly rely on heuristic algorithms or general-purpose solvers, leading to a trade-off between completeness and efficiency.

In this paper, we propose two novel approaches to synthesize invariants for polynomial programs using semidefinite programming (SDP). For basic semialgebraic templates, we apply techniques from robust optimization to construct a hierarchy of SDP relaxations. These relaxations induce a series of sub-level sets that under-approximate the set of valid parameter assignments. Under a certain non-degenerate assumption, we present a weak completeness result that the synthesized sets include almost all valid assignments. Furthermore, we discuss several extensions to improve the efficiency and expressiveness of the algorithm. We also identify a subclass of basic semialgebraic templates, called masked templates, for which the non-degenerate assumption is violated. Regarding masked templates, we present a substitution-based method to strengthen the invariant conditions. The strengthened constraints again admit a hierarchy of SDP approximations. Both of our approaches have been implemented, and empirical results demonstrate that they outperform the state-of-the-art methods.

Additional Key Words and Phrases: program verification, invariant synthesis, sum-of-squares relaxations, semidefinite programming

1 INTRODUCTION

The dominant approach to program verification is *Floyd-Hoare-Naur's inductive assertion method* [26, 32, 53], which is based on Hoare logic [32]. The central concept of Hoare logic is the *Hoare tripe*, represented as

$$\{P\}C\{Q\},$$

where C is a segment of program to be verified, P is the precondition and Q is the postcondition. The program C is said to be *partially correct* with respect to specifications P and Q if, assuming the precondition P holds before executing C and the program C terminates, then the postcondition Q will hold upon completion of C.

Authors' addresses: Hao Wu, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences& and University of CAS, Beijing, China; Qiuye Wang, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences& and University of CAS, Beijing, China, wangqye@ios.ac.cn; Bai Xue, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences& and University of CAS, Beijing, China, xuebai@ios.ac.cn; Naijun Zhan, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences& and University of CAS, Beijing, China, znj@ios.ac.cn; Lihong Zhi, Mathematics Mechanization Research Center, Institute of System Science, Chinese Academy of Sciences& and University of CAS, Beijing, China, lzhi@mmrc.iss.ac.cn; Zhi-Hong Yang, School of Mathematics and Statistics, Central South University, Changsha, China, zhihongyang2020@outlook.com.

2021. 0164-0925/2021/10-ART $15.00
https://doi.org/

In Hoare logic, an *invariant* is an assertion associated with a particular program location, and it holds true whenever the location is reached during program execution. An *inductive invariant* is a specific type of invariant that satisfies two key properties: it holds true at the first visit to the program location, and it continues to hold (is preserved) during subsequent visits to that location. The difference between invariants and inductive invariants is discussed in detail in [63]. For the purpose of this paper, we will solely focus on inductive invariants, and for simplicity, we will refer to them simply as "invariants", unless otherwise stated.

Identifying and generating invariants, as well as *termination analysis* [10, 12, 14, 17, 76], stands as a crucial aspect of Hoare-style program verification. The effectiveness of the verification process heavily relies on the ability to discover appropriate invariants that accurately capture the behavior and properties of the program throughout its execution. Though this has been shown to be undecidable in general [51], many efforts have been put into this area, resulting in various invariant synthesis techniques, including the approaches based on Criag interpolation [28, 45], abstract interpretation [59, 60], recurrence analysis [34, 38], and most recently, machine learning techniques [67, 74].

There is a class of work in this direction that employs the method of constraint-solving. Approaches falling under this category are commonly referred to as *constraint-solving-based (or template-based) invariant synthesis*. The general paradigm of these approaches is as follows: The algorithm takes a user-specified parametric formula as input, encodes the invariant conditions into constraints, and attempts to solve these constraints to obtain a valid invariant.

In this paper, we focus on synthesizing polynomial invariants for polynomial programs. In other words, we consider templates of the form $I(\boldsymbol{a}, \boldsymbol{x}) \leq 0$ where $I(\boldsymbol{a}, \boldsymbol{x})$ is a polynomial with unknown parameters $\boldsymbol{a}$. We will also explore the scenario where the template is a parametric (basic) semialgebraic set. In such cases, the conditions for the template to be an invariant can be expressed as a quantified first-order logic formula. It is worth noting that if both the program and the specifications (precondition and postcondition) are polynomial, then the formula is decidable, as per Tarski's theorem [70]. However, the resulting constraints are typically non-convex, posing challenges for finding valid assignments of parameters. Existing works mostly rely on general-purpose solvers [13, 29, 37, 73] or heuristics [1, 18, 46] to tackle these constraints.

Our Contributions. We propose two innovative approaches to synthesize invariants for polynomial programs using SDP.

Our first approach takes a more general perspective of the synthesis problem by characterizing the valid set (i.e., the collection of all valid assignments of parameters), rather than seeking a single valid assignment. To achieve this, we employ the technique from [43], which computes a series of polynomials $h^{(d)}(\boldsymbol{a})$ for $d \in \mathbb{N}$ such that any $\boldsymbol{a}_0$ satisfying $h^{(d)}(\boldsymbol{a}_0) \leq 0$ is a valid assignment. Under the non-degenerate assumption that the valid set contains at least one interior point, our approach produces a non-empty under-approximation of the valid set for sufficiently large d. Furthermore, we show that our approach is capable of handling more complex program structures, such as nested loops, and (basic) semialgebraic templates. We also present a binary search framework to improve the approximation precision in computation.

In scenarios where the non-degenerate assumption is violated, such as when the given basic semialgebraic template contains equalities, our first approach may not be able to generate a non-empty under-approximation. To overcome this limitation, we introduce a new approach tailored for a specific subclass of basic semialgebraic templates that include equalities, referred to as "masked templates". This approach addresses the problem of masked templates by strengthening the invariant conditions through variable substitution. By employing this technique, the strengthened constraints allow for a hierarchy of sum-of-squares approximations, similar to our first approach.

Both approaches have been implemented and tested on two sets of benchmarks, depending on the form of the target invariants. When compared with state-of-the-art constraint-solving-based and learning-based methods, both of our approaches demonstrate advantages in terms of effectiveness and efficiency.

In summary, our main contributions are:

(1) We present a weak complete algorithm based on SDP to synthesize invariants from polynomial (or basic semialgebraic) templates.
(2) We introduce the definition of masked templates and propose an algorithm to strengthen the invariant conditions with masked templates, allowing for the resulting constraints to be efficiently solved by SDP.
(3) We implement both of our approaches and showcase their superior performance when compared to state-of-the-art methods

The rest of this paper is organized as follows: In Section 2, we introduce some basic notions and algebraic tools that will be used. Section 3 presents the technique in [43] and applies it to our invariant synthesis problem. Section 4 discusses additional techniques to enhance the efficiency and expressiveness. Section 5 introduces the definition of masked templates and proposes a new approach based on variable substitution. We report the experimental results in Section 6 and discuss related work in Section 7. Finally, Section 8 concludes the paper.

2 PRELIMINARIES

The following basic notions will be used throughout the rest of this paper: $\mathbb{R}$, $\mathbb{R}^+$ and $\mathbb{N}$ respectively stand for the set of real numbers, the set of positive real numbers and the set of non-negative integers. We use boldface letters to denote vectors (such as $\boldsymbol{x}, \boldsymbol{y}$) and vector-valued functions (such as $\boldsymbol{f}(\boldsymbol{x})$, $\boldsymbol{g}(\boldsymbol{x})$). If not explicitly stated otherwise, the comparison between vectors is elementwise (i.e, for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, $\boldsymbol{x} \geq \boldsymbol{y}$ means $x_1 \geq y_1 \wedge x_2 \geq y_2 \wedge \cdots \wedge x_n \geq y_n$). $U(\boldsymbol{x}_0, \delta)$ denotes the δ neighbor of $\boldsymbol{x}_0$, i.e., $U(\boldsymbol{x}_0, \delta) = \{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}_0\| \leq \delta\}$, where $\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^n x_i^2}$ is the l_2-norm. $\mathbb{R}[\cdot]$ denotes the ring of polynomials in variables given by the argument, $\mathbb{R}^d[\cdot]$ denotes the set of polynomials of degree less than or equal to d in variables given by the argument, where $d \in \mathbb{N}$. For convenience, we do not explicitly distinguish a polynomial $p \in \mathbb{R}[\boldsymbol{x}]$ and the function it introduces. A basic semialgebraic set $\mathcal{K} \subseteq \mathbb{R}^n$ is of the form $\{\boldsymbol{x} \mid p_1(\boldsymbol{x}) \triangleleft 0, \ldots, p_m(\boldsymbol{x}) \triangleleft 0\}$, where $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ and $\triangleleft \in \{\leq, <\}$. A semialgebraic set is of the form $\bigcup_{i=1}^n \mathcal{K}_i$, where $\mathcal{K}_i$ are basic semialgebraic sets. Let $\mathcal{A} \subseteq \mathbb{R}^n, \mathcal{B} \subseteq \mathbb{R}^m$ be two semialgebraic sets, a mapping $f : \mathcal{A} \to \mathcal{B}$ is called a semialgebraic function if its graph $\{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{A} \times \mathcal{B} \mid f(\boldsymbol{x}) = \boldsymbol{y}\}$ is a semialgebraic set in $\mathbb{R}^{n+m}$.

2.1 Program Model

In this paper, we focus on synthesizing invariants for loops of the form presented in Code 1. The loop consists of a loop guard $\boldsymbol{g}(\boldsymbol{x}) \leq 0$ and a switch-case loop body, where the branch conditionals $\boldsymbol{c}_i(\boldsymbol{x}) \leq 0$ are tested in parallel. If more than one branch conditionals are satisfied, the program will nondeterministically choose a satisfied branch. Program variables are assumed to take real values. The goal is to prove the correctness of the program, i.e., for any state satisfying the precondition ($\boldsymbol{x} \in$ *Pre*), if the loop terminates, the final state must satisfy the postcondition ($\boldsymbol{x} \in$ *Post*). Here *Pre* and *Post* are basic semialgebraic sets defined by $\boldsymbol{q}_{pre}(\boldsymbol{x}) \leq 0$ and $\boldsymbol{q}_{post}(\boldsymbol{x}) \leq 0$ respectively.

We make two assumptions in our model:

Assumption 1. *Throughout the execution of the program, the program state $\boldsymbol{x}$ remains within a known hyper-rectangle $C_{\boldsymbol{x}} \subseteq \mathbb{R}^n$. In our algorithm, we consider $C_{\boldsymbol{x}}$ to be of the form $\{\boldsymbol{x} \in \mathbb{R}^n \mid x_1^2 - N^2 \leq 0, \ldots, x_n^2 - N^2 \leq 0\} = [-N, N]^n$, where $N \in \mathbb{N}$ is a constant.*

Code 1 The Program Model

```
// Program variables: $\boldsymbol{x} \in \mathbb{R}^n$
// Precondition: $Pre = \{\boldsymbol{x} \mid \boldsymbol{q}_{pre}(\boldsymbol{x}) \le 0\}$
while ($\boldsymbol{g}(\boldsymbol{x}) \le 0$) {
    case ($\boldsymbol{c}_1(\boldsymbol{x}) \le 0$) : $\boldsymbol{x} = f_1(\boldsymbol{x})$;
    case ($\boldsymbol{c}_2(\boldsymbol{x}) \le 0$) : $\boldsymbol{x} = f_2(\boldsymbol{x})$;
    ...
    case ($\boldsymbol{c}_k(\boldsymbol{x}) \le 0$) : $\boldsymbol{x} = f_k(\boldsymbol{x})$;
}
// Postcondition: $Post = \{\boldsymbol{x} \mid \boldsymbol{q}_{post}(\boldsymbol{x}) \le 0\}$
```

Assumption 2. *We treat the comparison operators "strictly less than" (e.g., $p(\boldsymbol{x}) < 0$) and "less than or equal to" (e.g., $p(\boldsymbol{x}) \le 0$) as indistinguishable. Consequently, throughout our algorithm, we consider $<, >$ to be equivalent to $\le, \ge$ respectively. Hence, when referring to basic semialgebraic sets, we assume that they are closed, i.e., defined by non-strict polynomial inequalities.*

Assumption 1 corresponds to the Archimedean condition for quadratic modules in Putinar's Positivstellensatz (introduced later in Section 2.3) and serves as a necessary condition for our completeness result in Section 3. This assumption is reasonable in most cases, as many real-world programs have natural bounds for program variables. Additionally, in practical programming languages like C, variables are typically assigned types and have known value ranges.

Assumption 2 is made due to the use of numerical methods in our approach. For numerical solvers, it is unnecessary and unrealistic to distinguish between $\le$ and $<$. From another perspective, our model still encompasses a broad set of programs that utilize floating-point numbers: given two adjacent floating-point numbers $a_1 < a_2$, then $x < a_2$ is equivalent to $x \le a_1$. However, the difference between floats and reals is not a focus of the current paper, so we simply replace $<, >$ with $\le, \ge$.

Both Assumptions 1,2 have been made in related invariant synthesis works based on Putinar's Positivstellensatz [1, 13, 29], either explicitly or implicitly, for the same reasons mentioned above.

Remark 1. *To better illustrate our main ideas, we concentrate on unnested loops following the structure in Code 1. As for more complex program structures, such as nested loops or programs represented by control flow graphs, our approach can be easily extended to handle them with minor modifications (see Section 4.2).*

2.2 Invariants

Now we give the formal definition of invariants.

Definition 1 (Invariant). *$Inv \subseteq \mathbb{R}^n$ is an invariant of the program in Code 1 if it satisfies the following conditions, also called the invariant conditions,*

(1) $\boldsymbol{x} \in Pre \implies \boldsymbol{x} \in Inv$;
(2) $\boldsymbol{x} \in Inv \wedge \boldsymbol{g}(\boldsymbol{x}) \le 0 \wedge \boldsymbol{c}_i(\boldsymbol{x}) \le 0 \implies f_i(\boldsymbol{x}) \in Inv$, *for* $i = 1, \dots, k$;
(3) $\boldsymbol{x} \in Inv \wedge \neg(\boldsymbol{g}(\boldsymbol{x}) \le 0) \implies \boldsymbol{x} \in Post$.

The existence of an invariant implies the correctness of the loop. However, directly searching for a satisfying *Inv* within the entire space of all subsets of $\mathbb{R}^n$ could be challenging. To address this issue, one common approach is to impose constraints on the invariants *Inv* to adhere to specific types of parametric formulas. In this paper, we primarily focus on polynomial templates, which are defined as follows. The discussion on basic semialgebraic templates and general semialgebraic templates is provided in Section 4.3.

Definition 2 (Polynomial Template). *A polynomial template is a polynomial $I(\boldsymbol{a}, \boldsymbol{x}) : C_{\boldsymbol{a}} \times \mathbb{R}^n \mapsto \mathbb{R}$ in $\mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$, where $C_{\boldsymbol{a}}$ is a hyper-rectangle in $\mathbb{R}^{n'}$ and $\boldsymbol{a} = (a_1, a_2, \ldots, a_{n'}) \in C_{\boldsymbol{a}}$ are referred to as parameters. Given a parameter assignment $\boldsymbol{a}_0 \in \mathbb{R}^{n'}$, the instantiation of the invariant Inv w.r.t. $\boldsymbol{a}_0$ is the set $\{\boldsymbol{x} \in C_{\boldsymbol{x}} \mid I(\boldsymbol{a}_0, \boldsymbol{x}) \leq 0\}$, where $C_{\boldsymbol{x}} = [-N, N]^n$ for some user-defined $N \in \mathbb{N}$.*

When a polynomial template $I(\boldsymbol{a}, \boldsymbol{x})$ is fixed, the invariant conditions can be expressed as quantified formulas in first-order logic:

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ \boldsymbol{q}_{pre}(\boldsymbol{x}) \leq 0 \implies I(\boldsymbol{a}, \boldsymbol{x}) \leq 0, \tag{1}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \wedge \boldsymbol{g}(\boldsymbol{x}) \leq 0 \wedge \boldsymbol{c}_i(\boldsymbol{x}) \leq 0 \implies I(\boldsymbol{a}, \boldsymbol{f}_i(\boldsymbol{x})) \leq 0, \quad i = 1, \ldots, k, \tag{2}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \wedge \bigwedge_{j=1}^{i-1} g_j(\boldsymbol{x}) \leq 0 \wedge g_i(\boldsymbol{x}) \geq 0 \implies \boldsymbol{q}_{post}(\boldsymbol{x}) \leq 0, \quad i = 1, \ldots, m_g, \tag{3}$$

where $\boldsymbol{g} = (g_1, g_2, \ldots, g_{m_g})$. When the loop guard is a single inequality, i.e., $\boldsymbol{g} = g_1$, Constraint (3) can be simplified into

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \wedge g_1(\boldsymbol{x}) \geq 0 \implies \boldsymbol{q}_{post}(\boldsymbol{x}) \leq 0.$$

It is worth noting that we use $g_i(\boldsymbol{x}) \geq 0$ rather than $g_i(\boldsymbol{x}) > 0$ to represent the negation of $g_i(\boldsymbol{x}) \leq 0$ in Constraint (3). This is a consequence of Assumption 2.

We are interested in finding assignments of $\boldsymbol{a} \in C_{\boldsymbol{a}}$ satisfying Constraints (1)-(3). The related concepts are formalized below:

Definition 3 (Valid and Valid Set). *Given a program as presented in Code 1 and a polynomial template $I(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$, a parameter assignment $\boldsymbol{a}_0 \in C_{\boldsymbol{a}}$ is valid if it satisfies constraints (1)-(3), meaning that the set $\{\boldsymbol{x} \mid I(\boldsymbol{a}_0, \boldsymbol{x}) \leq 0\}$ is an invariant of the program. The valid set, denoted by R_I, represents the collection of all valid parameter assignments for the polynomial template $I(\boldsymbol{a}, \boldsymbol{x})$.*

Remark 2. *The reason for the assumption $\boldsymbol{a} \in C_{\boldsymbol{a}}$ is similar to that of $\boldsymbol{x}$. However, when $I(\boldsymbol{a}, \boldsymbol{x})$ is homogeneous in $\boldsymbol{a}$, we can take $C_{\boldsymbol{a}}$ to be $[-1, 1]^{n'}$ without loss of generality. This is because the parameters $\boldsymbol{a}$ can be scaled by any positive constant without changing the invariant candidate they define. In practice, we often set the templates to be homogeneous in $\boldsymbol{a}$. For example, when the program variables are (x_1, x_2), a polynomial template could be $I = a_1x_1^2 + a_2x_1x_2 + a_3x_2^2 + a_4x_1 + a_5x_2 + a_6$, which represents any polynomial in (x_1, x_2) of degree not exceeding 2.*

2.3 Putinar's Positivstellensatz and Sum-of-Squares Relaxation

The sum-of-squares relaxation is a well-established technique in polynomial optimization. Its basic idea is to approximate a non-convex polynomial optimization problem by a sequence of convex optimization problems, which can be efficiently solved as SDPs. In this section, we introduce the fundamental concepts related to this technique and demonstrate its application to our synthesizing problem. For more comprehensive technical details, we suggest referring to [42, 56].

A polynomial $p(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ is said to be a *sum-of-squares* polynomial if it can be expressed as $p(\boldsymbol{x}) = \sum_{i=1}^{m} p_i(\boldsymbol{x})^2$, where $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ and $m \in \mathbb{N}$. Similar to $\mathbb{R}[\cdot]$ and $\mathbb{R}^d[\cdot]$, we use $\Sigma[\cdot]$ and $\Sigma^d[\cdot]$ to denote the set of sum-of-squares polynomials and sum-of-squares polynomials of degree less than or equal to d, respectively. Given polynomials $p_1, \ldots, p_m$, the set

$$Q(p_1, p_2, \ldots, p_m) = \{\sigma_0 + \sum_{i=1}^{m} \sigma_i p_i \mid \sigma_0, \sigma_i \in \Sigma[\boldsymbol{x}]\}$$

is called the *quadratic module* generated by $p_1, p_2, \ldots, p_m$. A quadratic module Q is called *Archimedean*, or satisfies *Archimedean condition*, if $N - \|\boldsymbol{x}\|^2 \in Q$ for some constant $N \in \mathbb{N}$.

Putinar's theorem basically states that, if a polynomial $p(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ is strictly positive over a compact basic semialgebraic set $\mathcal{K}$, then $p(\boldsymbol{x})$ admits a representation using the defining polynomials of $\mathcal{K}$. We now present the theorem formally:

Theorem 1 (Putinar's Positivstellensatz [42, 57]). *Given a basic semialgebraic set $\mathcal{K} = \{\boldsymbol{x} \mid p_1(\boldsymbol{x}) \geq 0, \ldots, p_m(\boldsymbol{x}) \geq 0\}$, if $Q(p_1, \ldots, p_m)$ is Archimedean and polynomial $p \in \mathbb{R}[\boldsymbol{x}]$ is strictly positive over $\mathcal{K}$, then $p \in Q(p_1, \ldots, p_m)$.*

The Archimedean condition is not very restrictive. For example, one can check that it is satisfied when $\{p_1(\boldsymbol{x}), \ldots, p_m(\boldsymbol{x})\}$ includes polynomials $x_i^2 - N^2 \leq 0$, for $1 \leq i \leq n$ and some $N \in \mathbb{N}$. This explains why we assume $\boldsymbol{x} \in C_{\boldsymbol{x}}$ and $\boldsymbol{a} \in C_{\boldsymbol{a}}$ for rectangles $C_{\boldsymbol{x}}$ and $C_{\boldsymbol{a}}$.

In the following, we use a simple example to demonstrate the power of Putinar's Positivstellensatz. Suppose we are given an optimization problem of the form

$$\begin{aligned} & min \quad \boldsymbol{c}^T \boldsymbol{a} \\ & s.t. \quad \forall \boldsymbol{x}. \bigwedge_{i=1}^{m} p_i(\boldsymbol{x}) \leq 0 \implies l(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \end{aligned} \tag{4}$$

where $\boldsymbol{c} \in \mathbb{R}^n$ are known constants, $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ are polynomials such that $Q(p_1, \cdots, p_m)$ is Archimedean, and $l(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$ is *linear* in $\boldsymbol{a}$.

By applying Putinar's Positivstellensatz, one can transform the Program (4) into

$$\begin{aligned} & min \quad \boldsymbol{c}^T \boldsymbol{a} \\ & s.t. \quad -l(\boldsymbol{a}, \boldsymbol{x}) = \sigma_0(\boldsymbol{x}) + \sum_{i=1}^{m} \sigma_i(\boldsymbol{x}) \cdot (-p_i(\boldsymbol{x})), \\ & \qquad \sigma_0, \sigma_i \in \Sigma[\boldsymbol{x}], \end{aligned} \tag{5}$$

where σ_0, σ_i are sum-of-squares polynomials with unknown coefficients.

Parrilo [55] showed that Program (5) can be approximated by solving a series of sum-of-squares relaxations. The idea is to impose restrictions on the degrees of the unknown sum-of-squares polynomials (or alternatively, restrict the maximum degree of the constraints). Given $d \in \mathbb{N}$, the d-th relaxation is defined as follows:

$$\begin{aligned} & min \quad \boldsymbol{c}^T \boldsymbol{a} \\ & s.t. \quad -l(\boldsymbol{a}, \boldsymbol{x}) = \sigma_0(\boldsymbol{x}) + \sum_{i=1}^{m} \sigma_i(\boldsymbol{x}) \cdot (-p_i(\boldsymbol{x})), \\ & \qquad \sigma_0, \sigma_i \in \Sigma^{2d}[\boldsymbol{x}]. \end{aligned} \tag{6}$$

where the decision variables consist of the parameters $\boldsymbol{a}$ as well as the unknown coefficients in $\sigma_{(\cdot)}$. When the degree bound d is fixed, Program (6) can be transformed into an SDP that can be efficiently solved in polynomial time, for example, using interior-point methods. As the degree bound d increases and approaches infinity, the optimal value of Program (6) converges to the optimal value of Program (5). Therefore, the series of sum-of-squares relaxations of the form Program (6) provides increasingly accurate approximations to the original non-convex Program (4).

For each d, the dual problem of Program (6) is called the moment problem and was first studied by Lasserre[41]. For this reason, this series of relaxations is referred to as the Moment-SOS hierarchy, or Lasserre's hierarchy[42].

Remark 3. *When $I(\boldsymbol{a}, x)$ is a polynomial template and is linear in $\boldsymbol{a}$, the invariant condition Constraint (1) conforms to the form of the constraint in Program (4), allowing for finding a satisfying assignment of $\boldsymbol{a}$ using the sum-of-squares relaxations discussed above. However, Constraints (2) and (3) can not be handled in the same manner. This is because the parameters $\boldsymbol{a}$ occur before the implication symbol. If we apply Putinar's Positivstellensatz, the resulting relaxation will include bilinear terms, which arise from the product of $\boldsymbol{a}$ and coefficients in $\sigma_{(\cdot)}$. As a result, the relaxation is no longer an SDP and poses additional challenges.*

3 SYNTHESIZING INVARIANTS FROM POLYNOMIAL TEMPLATES

In this section, we take a more general perspective of the synthesis problem by characterizing the valid set R_I, rather than just synthesizing a single valid assignment from R_I. To this end, we introduce the technique in [43] to construct a hierarchy of sum-of-squares relaxations for the valid set R_I. By solving these relaxations, we obtain a sequence of polynomials $h^{(d)}(\boldsymbol{a})$ for $d \in \mathbb{N}$ whose sub-level sets are under-approximations of R_I. Therefore, the invariant synthesis problem is reduced to a simple problem of deciding whether $h^{(d)}(\boldsymbol{a}) \leq 0$ has a solution over $\boldsymbol{a} \in C_{\boldsymbol{a}}$, which can be handled by modern symbolic solvers. Furthermore, we show that these under-approximations have some desired properties, including soundness, convergence, and weak completeness. Finally, we provide an example to illustrate the effectiveness of our approach.

3.1 Under-approximating Sets Defined with Quantifiers

In general, Lasserre's paper [43] deals with quantified constraints of the form

$$\forall \boldsymbol{x}. \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \implies l(\boldsymbol{a}, \boldsymbol{x}) \leq 0, \tag{7}$$

where $p_i(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$ and $l(\boldsymbol{a}, \boldsymbol{x})$ is a semialgebraic function (for now, we can consider $l(\boldsymbol{a}, \boldsymbol{x})$ as a polynomial in $\mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$). We assume that $p_i(\boldsymbol{a}, \boldsymbol{x})$ include the polynomials that define the hyperrectangles $C_{\boldsymbol{x}}$ and $C_{\boldsymbol{a}}$, ensuring that $\boldsymbol{Q}(p_1, \ldots, p_m)$ is Archimedean. Compared to Constraint (4), we have $p_i(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$ instead of $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$. Consequently, the invariant conditions (1)-(3) all conform to this form. The goal of [43] is to under-approximate the following set defined with a quantifier:

$$R = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid \forall \boldsymbol{x}. \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \implies l(\boldsymbol{a}, \boldsymbol{x}) \leq 0\}. \tag{8}$$

Let us define $\mathcal{K}_{\boldsymbol{a}} = \{\boldsymbol{x} \in C_{\boldsymbol{x}} \mid \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \leq 0\}$ and assume $\mathcal{K}_{\boldsymbol{a}}$ is non-empty for every $\boldsymbol{a} \in C_{\boldsymbol{a}}$, then we can express R as a sub-level set:

$$R = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid \bar{J}(\boldsymbol{a}) \leq 0\}, \tag{9}$$

where $\bar{J}(\boldsymbol{a}) = \sup_{\boldsymbol{x} \in \mathcal{K}_{\boldsymbol{a}}} l(\boldsymbol{a}, \boldsymbol{x})$. According to [43, Lemma 1], the function $\bar{J}(\boldsymbol{a})$ is upper semi-continuous in $C_{\boldsymbol{a}}$ (i.e., for all $\boldsymbol{a}_0 \in C_{\boldsymbol{a}}$, $\limsup_{\boldsymbol{a} \to \boldsymbol{a}_0} \bar{J}(\boldsymbol{a}) \leq \bar{J}(\boldsymbol{a}_0)$ holds). Therefore, we can under-approximate the set R by approximating $\bar{J}(\boldsymbol{a})$ from above. Namely, for any function $h(\boldsymbol{a}) \geq \bar{J}(\boldsymbol{a})$ over $C_{\boldsymbol{a}}$, we have $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid h(\boldsymbol{a}) \leq 0\} \subseteq R$. So the problem is reduced to the following optimization problem:

$$\begin{aligned} \min \quad & \int_{C_{\boldsymbol{a}}} h(\boldsymbol{a}) \mathrm{d}\mu(\boldsymbol{a}) \\ s.t. \quad & \forall \boldsymbol{x}. \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \leq 0 \implies h(\boldsymbol{a}) - l(\boldsymbol{a}, \boldsymbol{x}) \geq 0 \end{aligned} \tag{10}$$

where $\mu(\boldsymbol{a})$ is the scaled Lebesgue measure over C_a, i.e., $\int_{C_{\boldsymbol{a}}} \mathrm{d}\mu(\boldsymbol{a}) = 1$. Minimizing $\int_{C_{\boldsymbol{a}}} h(\boldsymbol{a})\mathrm{d}\mu(\boldsymbol{a})$ is the same as minimizing $\int_{C_{\boldsymbol{a}}} |h(\boldsymbol{a}) - \bar{J}(\boldsymbol{a})|\mathrm{d}\mu(\boldsymbol{a})$, i.e., the gap between h and $\bar{J}$ over $C_{\boldsymbol{a}}$.

By applying Putinar's Positivstellensatz, we can also construct a series of sum-of-squares relaxations for Program (10). Fix $d \in \mathbb{N}$, we set $h(\boldsymbol{a})$ to be a polynomial of degree d, i.e., $h(\boldsymbol{a}) = \sum_{\boldsymbol{\beta}} \mathrm{h}_{\boldsymbol{\beta}} \boldsymbol{a}^{\boldsymbol{\beta}}$ where $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_{n'})$ with $\sum_{j=1}^{n'} \beta_j \leq d$ and $\mathrm{h}_{\boldsymbol{\beta}} \in \mathbb{R}$ are unknown coefficients in $\mathbb{R}$. The d-th relaxation is given by

$$\begin{aligned} \min \quad & \sum_{\boldsymbol{\beta}} \gamma_{\boldsymbol{\beta}} \mathrm{h}_{\boldsymbol{\beta}} \\ s.t. \quad & h(\boldsymbol{a}) - l(\boldsymbol{a}, \boldsymbol{x}) = \sigma_0(\boldsymbol{a}, \boldsymbol{x}) + \sum_{i=1}^{m} \sigma_i(\boldsymbol{a}, \boldsymbol{x}) \cdot (-p_i(\boldsymbol{a}, \boldsymbol{x})), \\ & \sigma_0, \sigma_i \in \Sigma^{2\lceil d/2 \rceil}[\boldsymbol{a}, \boldsymbol{x}], \end{aligned} \tag{11}$$

where $\gamma_{\boldsymbol{\beta}} = \frac{1}{\mu(C_a)} \int_{C_{\boldsymbol{a}}} \boldsymbol{a}^{\boldsymbol{\beta}} \mathrm{d}\mu(\boldsymbol{a})$. Notice that, in the d-th relaxation, d corresponds to the degree of $h(\boldsymbol{a})$ instead of the degree of unknown sum-of-squares polynomials.

Given feasible assignments $\mathrm{h}^*_{\boldsymbol{\beta}}$, the d-th polynomial approximation of $\bar{J}(\boldsymbol{a})$ can be obtained as $h^{(d)}(\boldsymbol{a}) = \sum_{\boldsymbol{\beta}} \mathrm{h}^*_{\boldsymbol{\beta}} \boldsymbol{a}^{\boldsymbol{\beta}}$, and the corresponding under-approximation of R is $R^{(d)} = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid h^{(d)}(\boldsymbol{a}) \leq 0\}$. Moreover, we have the following theorem.

Theorem 2. [*] *[43, Theorem 5] Assume that R has nonempty interior and $\mathcal{K}_{\boldsymbol{a}}$ is non-empty for every $\boldsymbol{a} \in C_{\boldsymbol{a}}$, then $h^{(d)}(\boldsymbol{a})$ converges to $\bar{J}(\boldsymbol{a})$ (from above) as d goes to ∞, i.e.,*

$$\lim_{d \to \infty} \int_{C_{\boldsymbol{a}}} |h^{(d)}(\boldsymbol{a}) - \bar{J}(\boldsymbol{a})| \mathrm{d}\boldsymbol{a} = 0.$$

3.2 Encoding the Invariant Conditions

In this part, we show how to encode the invariant conditions using the approach introduced above. For simplicity, we assume that $\boldsymbol{q}_{pre}, \boldsymbol{q}_{post}, \boldsymbol{g}, \boldsymbol{c}_i$ are polynomials (instead of basic semialgebraic sets) and use q_{pre}, q_{post}, g, and c_i instead. Furthermore, we set $C_{\boldsymbol{x}} = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_1^2 - N^2 \leq 0, \ldots, x_n^2 - N^2 \leq 0\}$ (in our experiments, $N = 100$), and $C_{\boldsymbol{a}} = \{\boldsymbol{x} \in \mathbb{R}^{n'} \mid a_1^2 - 1 \leq 0, \ldots, a_{n'}^2 - 1 \leq 0\}$.

Similar to Equation (9), we want to express the valid set R_I as a sub-level set. Following the definition of $\bar{J}$, let us define $\bar{J}_1(\boldsymbol{a}), \cdots, \bar{J}_{k+2}(\boldsymbol{a})$ for constraints (1)-(3) respectively (recall that k is the number of branches). For example, $\bar{J}_1(\boldsymbol{a}) = \sup_{\boldsymbol{x} \in \mathcal{K}_{\boldsymbol{a},1}} I(\boldsymbol{a}, \boldsymbol{x})$ with $\mathcal{K}_{\boldsymbol{a},1} = \{\boldsymbol{x} \in C_{\boldsymbol{x}} \mid q_{pre}(\boldsymbol{x}) \leq 0\}$. Additionally, we define

$$J(\boldsymbol{a}) = \max\{\bar{J}_1(\boldsymbol{a}), \cdots, \bar{J}_{k+2}(\boldsymbol{a}), M\}, \tag{12}$$

where $M < 0$ is a constant (in our experiments, $M = -10$). Then we have

$$R_I = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid J(\boldsymbol{a}) \leq 0\}. \tag{13}$$

Here the constant M is introduced to ensure $J(\boldsymbol{a}) > -\infty$, otherwise the derived relaxation will be unbounded when $\mathcal{K}_{\boldsymbol{a},i} = \emptyset$ for all $1 \leq i \leq k+2$. In this way, the condition $\mathcal{K}_{\boldsymbol{a}} \neq \emptyset$ in Theorem 2 can be dropped.

Similar to Programs (10) and (11), a series of under-approximations of R_I can be obtained by solving the following hierarchy of sum-of-square relaxations. Fix $d \in \mathbb{N}$, the d-th relaxation $\mathbf{M}_d$ is

[*] The proof of Theorem 2 is based on the moment approach.

given by

$$
\begin{aligned}
&\mathbf{M}_d : \min \quad \sum_{\boldsymbol{\beta}} \gamma_{\boldsymbol{\beta}} \mathrm{h}_{\boldsymbol{\beta}} \\
s.t. \quad h(\boldsymbol{a}) - I(\boldsymbol{a},\boldsymbol{x}) &= \sigma_0(\boldsymbol{a},\boldsymbol{x}) + \sigma_1(\boldsymbol{a},\boldsymbol{x}) \cdot (-q_{pre}(\boldsymbol{a},\boldsymbol{x})) \\
&\quad + \sum_{j=1}^{n} \sigma_j^{\boldsymbol{x}}(\boldsymbol{a},\boldsymbol{x}) \cdot (N^2 - x_i^2) + \sum_{j=1}^{n'} \sigma_j^{\boldsymbol{a}}(\boldsymbol{a},\boldsymbol{x}) \cdot (1 - a_i^2) && (14) \\
h(\boldsymbol{a}) - I(\boldsymbol{a}, f_i(\boldsymbol{x})) &= \dot{\sigma}_{i,0}(\boldsymbol{a},\boldsymbol{x}) + \dot{\sigma}_{i,1}(\boldsymbol{a},\boldsymbol{x}) \cdot (-I(\boldsymbol{a},\boldsymbol{x})) + \dot{\sigma}_{i,2}(\boldsymbol{a},\boldsymbol{x}) \cdot (-c_i(\boldsymbol{a},\boldsymbol{x})) \\
&\quad + \sum_{j=1}^{n} \dot{\sigma}_{i,j}^{\boldsymbol{x}}(\boldsymbol{a},\boldsymbol{x}) \cdot (N^2 - x_i^2) + \sum_{j=1}^{n'} \dot{\sigma}_{i,j}^{\boldsymbol{a}}(\boldsymbol{a},\boldsymbol{x}) \cdot (1 - a_i^2) \quad \text{for } i = 1, \ldots, k && (15) \\
h(\boldsymbol{a}) - q_{post}(\boldsymbol{x}) &= \mathring{\sigma}_0(\boldsymbol{a},\boldsymbol{x}) + \mathring{\sigma}_1(\boldsymbol{a},\boldsymbol{x}) \cdot (-I(\boldsymbol{a},\boldsymbol{x})) + \mathring{\sigma}_2(\boldsymbol{a},\boldsymbol{x}) \cdot g(\boldsymbol{x}) \\
&\quad + \sum_{j=1}^{n} \mathring{\sigma}_j^{\boldsymbol{x}}(\boldsymbol{a},\boldsymbol{x}) \cdot (N^2 - x_i^2) + \sum_{j=1}^{n'} \mathring{\sigma}_j^{\boldsymbol{a}}(\boldsymbol{a},\boldsymbol{x}) \cdot (1 - a_i^2) && (16) \\
h(\boldsymbol{a}) - M &= \hat{\sigma}_0(\boldsymbol{a},\boldsymbol{x}) + \sum_{j=1}^{n} \hat{\sigma}_j^{\boldsymbol{x}}(\boldsymbol{a},\boldsymbol{x}) \cdot (N^2 - x_i^2) + \sum_{j=1}^{n'} \hat{\sigma}_j^{\boldsymbol{a}}(\boldsymbol{a},\boldsymbol{x}) \cdot (1 - a_i^2) && (17) \\
&\sigma_{(\cdot)}^{(\cdot)}, \dot{\sigma}_{(\cdot)}^{(\cdot)}, \mathring{\sigma}_{(\cdot)}^{(\cdot)}, \hat{\sigma}_{(\cdot)}^{(\cdot)} \in \Sigma^{2\lceil d/2 \rceil}(\boldsymbol{a},\boldsymbol{x})
\end{aligned}
$$

where, according to the definition of $C_{\boldsymbol{a}}$,

$$
\gamma_{\boldsymbol{\beta}} = \frac{1}{2^{n'}} \int_{C_{\boldsymbol{a}}} \boldsymbol{a}^{\boldsymbol{\beta}} \mathrm{d}\mu(\boldsymbol{a}) = \begin{cases} 0 & \text{if } \beta_i \text{ is odd for some } i, \\ \prod_{i=1}^{n} (\beta_i + 1)^{-1} & \text{otherwise.} \end{cases}
$$

If $\mathbf{M}_d$ is solvable for some $d \in \mathbb{N}$, let $\mathrm{h}_{\boldsymbol{\beta}}^*$ denote the optimal solutions of $\mathbf{M}_d$. Then the under-approximation of R_I is obtained as $R_I^{(d)} = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid h^{(d)}(\boldsymbol{a}) \leq 0\} \subseteq R_I$, where $h^{(d)}(\boldsymbol{a}) = \sum_{\boldsymbol{\beta}} \mathrm{h}_{\boldsymbol{\beta}}^* \boldsymbol{a}^{\boldsymbol{\beta}}$. If $\mathbf{M}_d$ does not admit an optimal solution, then we denote $h^{(d)}(\boldsymbol{a}) = 1$, which implies $R_I^{(d)} = \emptyset$.

Finally, to synthesize a valid assignment of parameters $\boldsymbol{a}$, we only need to find a solution of $\boldsymbol{a} \in C_{\boldsymbol{a}}$ such that $h^{(d)}(\boldsymbol{a}) \leq 0$. The new synthesis problem is much easier than the original invariant synthesis problem. In most cases, it can be efficiently solved by modern symbolic solvers.

Remark 4. *The technique in [43] can be viewed as an approach to construct "higher-level" sum-of-squares relaxations. Instead of synthesizing one valid assignment of $\boldsymbol{a}$, the new relaxations try to characterize the requirement for $\boldsymbol{a}$ to be valid. From the perspective of invariant synthesis, what we obtain is not a single invariant but a* ***cluster*** *of invariants of similar shapes.*

3.3 Convergence, Soundness, and Weak Completeness

Following Theorem 2, the under-approximations $R_I^{(d)}$ can be proved to have many desired properties.

Theorem 3 (Soundness). *For any $d \in \mathbb{N}$, $R_I^{(d)}$ is an under-approximation of the valid set R_I, i.e. $R_I^{(d)} \subseteq R_I$.*

Proof. If program $\mathbf{M}_d$ is unsolvable, then $R_I^{(d)} = \emptyset \subseteq R_I$.

If program $\mathbf{M}_d$ is solvable and $h^{(d)}(\boldsymbol{a})$ is obtained, by Theorem 2, we have $h^{(d)}(\boldsymbol{a}) > J(\boldsymbol{a})$ over $C_{\boldsymbol{a}}$, therefore $R_I^{(d)} \subseteq R_I$. □

Theorem 4 (Convergence). *Provided that the set $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid J(\boldsymbol{a}) = 0\}$ has Lebesgue measure zero, then we have*

$$\lim_{d\to\infty} \mu(R_I \setminus R_I^{(d)}) = 0,$$

Proof. Very similar to [43, Theorem 3]. □

Theorem 5 (Weak Completeness). *If the set $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid J(\boldsymbol{a}) = 0\}$ has Lebesgue measure zero and the valid set R_I contains an interior point, solving program $\mathbf{M}_d$ will give a non-empty under-approximation $R_I^{(d)}$ of the valid set R_I for some $d \in \mathbb{N}$ large enough.*

Proof. Since R_I contains an interior point, we know that R_I has a positive Lebesgue measure. Therefore, by Theorem 4, $R_I^{(d)}$ has a positive Lebesgue measure when d is large enough, which implies that $R_I^{(d)}$ is a non-empty under-approximation of the valid set R_I. □

Now we discuss the conditions in Theorem 4 and 5.

The assumption that $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid J(\boldsymbol{a}) = 0\}$ has Lebesgue measure zero basically states that the set of zero points of $J(\boldsymbol{a})$ should be negligible. Note that the zero points of $J(\boldsymbol{a})$ are also zero points of $\bar{J}_i(\boldsymbol{a})$. When the assumption is violated, there must exists some $\bar{J}_i(\boldsymbol{a})$ such that $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid \bar{J}_i(\boldsymbol{a}) = 0\}$ has a positive Lebesgue measure. Let us assume that the set of zero points of $\bar{J}_i(\boldsymbol{a})$ has a positive Lebesgue measure for some i. Similar to the following Lemma 1, we can prove that $\bar{J}_1(\boldsymbol{a}), \dots, \bar{J}_{k+2}(\boldsymbol{a})$ and $J(\boldsymbol{a})$ are all semialgebraic functions. By [9, Lemma 2.5.2], there exists a nonzero polynomial $f \in \mathbb{R}[\boldsymbol{a}, y]$ such that $f(\boldsymbol{a}, \bar{J}_i(\boldsymbol{a})) = 0$ for every $\boldsymbol{a} \in C_{\boldsymbol{a}}$. Since the set of zero points of $\bar{J}_i(\boldsymbol{a})$ is contained in the set of zero points of $f(\boldsymbol{a}, 0)$, the set $\{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid f(\boldsymbol{a}, 0) = 0\}$ also has a positive Lebesgue measure. As $f(\boldsymbol{a}, 0)$ is a polynomial, $f(\boldsymbol{a}, 0)$ must be constant zero. In other words, $f(\boldsymbol{a}, y)$ contains y as a factor, which is relatively rare in theory. In practice, we are not ware of any program that violates this assumption.

Lemma 1. *$\bar{J}(\boldsymbol{a})$ as defined in Equation (9) is a semialgebraic function.*

Proof. The graph of the function $\bar{J}(\boldsymbol{a})$ is

$$\begin{aligned}\{(\boldsymbol{a}, y) \in \mathbb{R}^{n'+1} \mid (\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \le 0 \implies l(\boldsymbol{a}, \boldsymbol{x}) \le y) \wedge \\ (\forall \epsilon \in \mathbb{R}^+, \exists \boldsymbol{x} \in C_{\boldsymbol{x}}.\ \bigwedge_{i=1}^{m} p_i(\boldsymbol{a}, \boldsymbol{x}) \le 0 \wedge l(\boldsymbol{a}, \boldsymbol{x}) + \epsilon > y)\},\end{aligned}$$

which is a semialgebraic set in $\mathbb{R}^{n'+1}$ by Tarski-Seidenberg principle [9, Proposition 2.2.4]. □

The other assumption, $\mathbb{R}_I$ containing an interior point, ensures that there exists a region with a positive Lebesgue measure where $J(a)$ take negative values. This assumption also suggests the existence of a "robust" invariant $I(\boldsymbol{x}, \boldsymbol{a_0})$ such that $I(\boldsymbol{x}, \boldsymbol{a_0} + \boldsymbol{\epsilon})$ is still an invariant for sufficiently small $\boldsymbol{\epsilon} \in \mathbb{R}^{n'}$.

We end this section with an illustrative example.

Example 1. *Let us consider the discrete-time dynamical system presented in Code 2. The star symbol $(*)$ means that the loop guard condition can be violated at any time, so the postcondition should always hold. In practical computation, this can be achieved by simply setting the loop guard $\boldsymbol{g}(\boldsymbol{x}) = 0$ (the negation of $0 \le 0$ is written as $0 \ge 0$, according to Assumption 2).*

We use the following polynomial template to search for ellipsoid-shaped invariants centered at the origin:

$$Inv(\hat{a}, \hat{b}, x, y) = x^2 + \hat{a}y^2 + \hat{b},$$

Code 2 A Simple Discrete-Time Dynamical System

```
// Precondition: $\{x^2 + y^2 - 1 \leq 0\}$
while (*) {
    $x = 0.9(x - 0.01y)$;
    $y = 0.9(y + 0.01x)$;
}
// Postcondition: $\{x^2 + (y - 2)^2 - 0.25 \geq 0\}$
```

where $\hat{a}$ and $\hat{b}$ are parameters with range $(\hat{a}, \hat{b}) \in [-10, 10]^2$.

First, we rescale the range of parameters to $[-1, 1]^2$ by introducing new parameters $a = \hat{a}/10$ and $b = \hat{b}/10$. Then, by applying our method, we obtain a series of sum-of-squares relaxations of the form Constraints (14)-(17) in the new parameters a, b.

We have shown that, for each $d \in \mathbb{N}$, the sub-level set $\{(a, b) \in [-1, 1]^2 \mid h^{(d)}(a, b) \leq 0\}$ is an under-approximation of the valid set R_I. However, when d is too small, $h^{(d)}(a, b) \leq 0$ could have no solutions, meaning that the under-approximation is an empty set. In this case, the under-approximation is useless and we go on trying relaxations of larger degrees. For example, solving the relaxation of degree 3,4,5 and 6 will give following polynomials, respectively:

$$\begin{aligned}
h^{(3)}(a, b) &= -1.074940b^3 - 2.262917ab^2 + \cdots + 3.051822a + 2.847483,\\
h^{(4)}(a, b) &= -1.167411b^4 - 0.192411ab^3 + \cdots + 2.971745a + 2.464498,\\
h^{(5)}(a, b) &= 0.813646b^5 + 1.996911ab^4 + \cdots + 2.833340a + 2.312266,\\
h^{(6)}(a, b) &= 2.688766b^6 + 0.095530ab^5 + \cdots + 2.893498a + 1.915259.
\end{aligned}$$

In Figure 1, we plot the sub-level sets of the above four polynomials. One can see that $\{(a, b) \in [-1, 1]^2 \mid h^{(d)}(a, b) \leq 0\}$ is empty for $d = 3$ and non-empty for the rest. In most cases, larger d usually means better under-approximations.

For each obtained polynomial $h^{(d)}(a, b)$, we use symbolic constraint solvers such as Z3[22] to check whether $h^{(d)}(a, b) \leq 0$ is solvable over $[-1, 1]^2$. These tasks are much easier than the quantified constraints for invariants and can be answered quickly when the polynomial is not too complex. For example, Z3 finds a value $(a_0, b_0) = (0.6915578, -1)$ such that $h^{(4)}(a_0, b_0) \leq 0$, which implies that

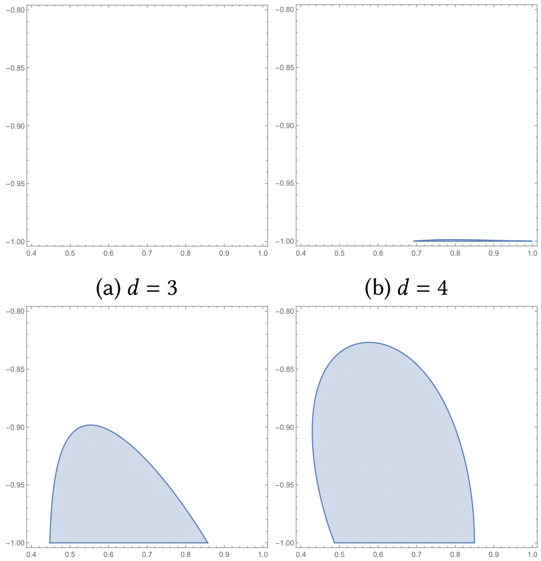
$$x^2 + 6.915578y^2 - 10 \leq 0$$

is an invariant of Code 2.

Note that even though this example seems simple enough, many existing invariant synthesizing tools do not support this nonlinear template or fail to synthesize a suitable invariant [16, 33, 38]. On the other hand, directly applying symbolic solvers such as Z3 or Redlog[24] also fails to produce a satisfying assignment of parameters in an hour.

4 EXTENSIONS

In this section, we discuss several extensions of our approach. In Section 4.1, we propose a binary search algorithm to improve the approximation precision over $C_{\boldsymbol{a}}$. In Section 4.2 and 4.3, we extend our approach to allow for more complex program structures and invariant templates, providing a more flexible and expressive framework for synthesizing invariants.



(a) $d = 3$

(b) $d = 4$

(c) $d = 5$

(d) $d = 6$

x-axis: a from 0.4 to 1.0; y-axis: b from -1.0 to -0.8

Fig. 1. Sub-level sets $\{(a, b) \in [-1, 1]^2 \mid h^{(d)}(a, b) \leq 0\}$ for $d = 3, 4, 5, 6$.

4.1 Binary Search Strategy

In Section 3, our goal is to under-approximate the valid set R_I. To achieve this, we start by selecting a hyper-rectangle $C_{\boldsymbol{a}}$ that contains potential parameter choices. The under-approximation procedure involves finding polynomials $h^{(d)}(\boldsymbol{a})$ to approximate the upper semi-continuous function $J(\boldsymbol{a})$ over $C_{\boldsymbol{a}}$ from above. As we have shown, increasing the relaxation degree d can improve the precision of the approximation. However, we can also enhance the approximation by adjusting the size of $C_{\boldsymbol{a}}$. In this subsection, we propose a binary search algorithm, which entails iteratively partitioning $C_{\boldsymbol{a}}$ to improve the approximation precision.

In the binary search scheme, we start with an initial partition $C_{\boldsymbol{a}} = C'_{\boldsymbol{a}} \cup C''_{\boldsymbol{a}}$, where $C'_{\boldsymbol{a}}$ and $C''_{\boldsymbol{a}}$ are disjoint. It is straight forward to see that the valid set R_I w.r.t. $C_{\boldsymbol{a}}$ is the union of the valid set R'_I and R''_I w.r.t. $C'_{\boldsymbol{a}}$ and $C''_{\boldsymbol{a}}$ respectively. Therefore, under-approximations of R_I can be obtained by taking the union of under-approximations of R'_I and R''_I. Notice that the objective value of program $\mathbf{M}_d$, denoted by v, is actually the rescaled integral of the polynomial $h^{(d)}(\boldsymbol{a})$ over $C_{\boldsymbol{a}}$. As $h^{(d)}(\boldsymbol{a})$ approximates $J(\boldsymbol{a})$ from the above, the optimal value v measures the approximation gap to some extents. Furthermore, since a feasible approximation $h^{(d)}(\boldsymbol{a})$ over $C_{\boldsymbol{a}}$ is still feasible over C'_a and C''_a, there will always be $\Delta v = v - \frac{1}{2}(v' + v'') \geq 0$, where v' and v'' are the respective objective values obtained by solving $\mathbf{M}_d$ over $C'_{\boldsymbol{a}}$ and $C''_{\boldsymbol{a}}$. Consequently, one may use the difference Δv to measure the improvements induced by each partitioning step. The binary search scheme stops either when the size of the current $C_{\boldsymbol{a}}$ becomes sufficiently small or when the improvement Δv is below a predefined threshold. The overall scheme is summarized as Algorithm 1.

Algorithm 1: Binary Search

```
Input  : a hyper-rectangle $C_{\boldsymbol{a}} \subseteq \mathbb{R}^{n'}$ and thresholds $\epsilon_d, \epsilon_v > 0$
Output : a polynomial $h \in \mathbb{R}[\boldsymbol{a}]$
$C \leftarrow C_{\boldsymbol{a}}$;
$(v, h) \leftarrow$ Solve $\mathbf{M}_d$ with respect to $C$ ; ▹ $v$: optimal value; $h$: polynomial $h^{(d)}(\boldsymbol{a})$
while $\mathit{Diameter}(C) > \epsilon_d$ do ▹ Diameter returns the length of the longest side
    $(C_l, C_r) \leftarrow$ Bisection$(C)$ ; ▹ Bisection bisects the rectangle from the longest side
    $(v_l, h_l) \leftarrow$ Solve $\mathbf{M}_d$ with respect to $C_l$;
    $(v_r, h_r) \leftarrow$ Solve $\mathbf{M}_d$ with respect to $C_r$;
    $\Delta v = v - \frac{1}{2}(v_l + v_r)$;
    if $\Delta v < \epsilon_v$ then break; ▹ stop when the improvement is small
    if $v_l < v_r$ then
        $(C, v, p) \leftarrow (C_l, v_l, h_l)$;
    else
        $(C, v, p) \leftarrow (C_r, v_r, h_r)$;
    end
end
return $h$
```

Code 3 A Simple Nested Loop

```
// Program variables: $\boldsymbol{x} \in \mathbb{R}^n$
// Precondition: $\mathit{Pre} = \{\boldsymbol{x} \mid \boldsymbol{q}_{pre}(\boldsymbol{x}) \le 0\}$
// Invariant (outer): $I_1(\boldsymbol{a}, \boldsymbol{x})$ with $\boldsymbol{a} \in R^{n'_1}$
while ($g_1(\boldsymbol{x}) \le 0$) {
    $\boldsymbol{x} = f_1(\boldsymbol{x})$;
    // Invariant (inner): $I_2(\boldsymbol{b}, \boldsymbol{x})$ with $\boldsymbol{b} \in R^{n'_2}$
    while ($g_2(\boldsymbol{x}) \le 0$) {
        $\boldsymbol{x} = f_2(\boldsymbol{x})$;
    }
}
// Postcondition: $\mathit{Post} = \{\boldsymbol{x} \mid \boldsymbol{q}_{post}(\boldsymbol{x}) \le 0\}$
```

4.2 Nested Loops

In Section 3, we have focused on unnested conditional loops of the form Code 1. In fact, our approach can be extended to nested loops or even control flow graphs without any substantial changes.

To illustrate the main idea, let us consider a simple nested loop Code 3. Synthesizing invariants for Code 3 is more challenging compared to Code 1 because it involves two invariants: $I_1(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$ for the outer while loop and $I_2(\boldsymbol{b}, \boldsymbol{x}) \in \mathbb{R}[\boldsymbol{b}, \boldsymbol{x}]$ for the inner while loop. Similar to the unnested cases, we assume that $\boldsymbol{a} \in C_{\boldsymbol{a}}, \boldsymbol{b} \in C_{\boldsymbol{b}}$ for some known hyper-rectangle $C_{\boldsymbol{a}}, C_{\boldsymbol{b}}$. The goal is to find valid assignments of the parameters $\boldsymbol{a}$ and $\boldsymbol{b}$ satisfying the following constraints:

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ \boldsymbol{q}_{pre}(\boldsymbol{x}) \leq 0 \implies I_1(\boldsymbol{a},\boldsymbol{x}) \leq 0, \tag{18}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I_1(\boldsymbol{a},\boldsymbol{x}) \leq 0 \wedge g_1(\boldsymbol{x}) \leq 0 \implies I_2(\boldsymbol{b}, f_1(\boldsymbol{x})) \leq 0, \tag{19}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I_2(\boldsymbol{b},\boldsymbol{x}) \leq 0 \wedge g_2(\boldsymbol{x}) \leq 0 \implies I_2(\boldsymbol{b}, f_2(\boldsymbol{x})) \leq 0, \tag{20}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I_2(\boldsymbol{b},\boldsymbol{x}) \leq 0 \wedge g_2(\boldsymbol{x}) \geq 0 \implies I_1(\boldsymbol{a},\boldsymbol{x}) \leq 0, \tag{21}$$

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ I_1(\boldsymbol{a},\boldsymbol{x}) \leq 0 \wedge g_1(\boldsymbol{x}) \geq 0 \implies \boldsymbol{q}_{post}(\boldsymbol{x}) \leq 0. \tag{22}$$

Here constraints (19)-(21) play two roles: for the inner loop, they encode the conditions of I_2 to be an invariant with I_1 serving as both precondition and postcondition; for the outer loop, they collectively encode the inductive condition of I_1.

It is not hard to see, constraints (18)-(22) still adhere to the form of Constraint (7), albeit parameters $\boldsymbol{a}$ are replaced by $(\boldsymbol{a},\boldsymbol{b})$. As a result, our approach in Section 3 remains applicable for nested loops. Furthermore, if the conditions stated in Theorem 5 are met, our approach will find a non-empty under-approximation of the valid set for parameters $(\boldsymbol{a},\boldsymbol{b})$. In general, our approach can be applied to programs represented by control flow graphs.

4.3 Semialgebraic Templates

In this subsection, we discuss the extensions of our approach to deal with basic semialgebraic templates and (general) semialgebraic templates. First, we briefly show that techniques in Section 3 can be directly applied to the cases when templates are basic semialgebraic (instead of only polynomial) without substantial changes. After that, we discuss how to deal with general semialgebraic templates. The formal definitions of basic semialgebraic templates and (general) semialgebraic templates are given as follows:

Definition 4 (Basic Semialgebraic Template). *A basic semialgebraic template is a finite collection of polynomials $I_r(\boldsymbol{a},\boldsymbol{x}) : C_{\boldsymbol{a}} \times \mathbb{R}^n \mapsto \mathbb{R}$ in $\mathbb{R}[\boldsymbol{a},\boldsymbol{x}]$, where $C_{\boldsymbol{a}}$ is a hyper-rectangle in $\mathbb{R}^{n'}$. Given a parameter assignment $\boldsymbol{a}_0 \in \mathbb{R}^{n'}$, the instantiation of the invariant Inv w.r.t. $\boldsymbol{a}_0$ is the set $\{\boldsymbol{x} \mid \bigwedge_r I_r(\boldsymbol{a}_0,\boldsymbol{x}) \leq 0\}$.*

Definition 5 (Semialgebraic Template). *A (general) semialgebraic template is a finite collection of polynomials $I_{t,r}(\boldsymbol{a},\boldsymbol{x}) : C_{\boldsymbol{a}} \times \mathbb{R}^n \mapsto \mathbb{R}$ in $\mathbb{R}[\boldsymbol{a},\boldsymbol{x}]$, where $C_{\boldsymbol{a}}$ is a hyper-rectangle in $\mathbb{R}^{n'}$. Given a parameter assignment $\boldsymbol{a}_0 \in \mathbb{R}^{n'}$, the instantiation of the invariant Inv w.r.t. $\boldsymbol{a}_0$ is the set $\{\boldsymbol{x} \mid \bigvee_t \bigwedge_r I_{t,r}(\boldsymbol{a}_0,\boldsymbol{x}) \leq 0\}$.*

As for basic semialgebraic templates, we show that this case can be reduced to the polynomial case with minor modifications. When the template $I(\boldsymbol{a},\boldsymbol{x})$ is replaced by $\bigwedge_r l_r(\boldsymbol{a},\boldsymbol{x}) \leq 0$ in Constraints (1)-(3), the new constraints are of the from

$$\forall \boldsymbol{x}. \bigwedge_{i=1}^{m} h_i(\boldsymbol{a},\boldsymbol{x}) \leq 0 \implies \bigwedge_r l_r(\boldsymbol{a},\boldsymbol{x}) \leq 0. \tag{23}$$

Immediately, one can see the above constraint can be rewritten into multiple constraints of the form Constraint (4), i.e.,

$$\forall \boldsymbol{x}. \bigwedge_{i=1}^{m} h_i(\boldsymbol{a},\boldsymbol{x}) \leq 0 \implies l_r(\boldsymbol{a},\boldsymbol{x}) \leq 0, \quad \text{for each } r. \tag{24}$$

Consequently, the derived sum-of-square relaxations will be much like Program (11) but include more constraints. After that, all other results can be derived similarly.

As for general semialgebraic templates, unfortunately, simply rewriting the constraints no longer works due to the combination of constraints. To address this issue, we resort to the *lifting* technique introduced in [44] to reduce the problem to the basic semialgebraic case.

Given a semialgebraic template as in Definition 5, the synthesis problem amounts to solving the following quantified constraints for $\boldsymbol{a}$:

$$\forall \boldsymbol{x}.(\boldsymbol{a},\boldsymbol{x}) \in \mathcal{K} \implies s(\boldsymbol{a},\boldsymbol{x}) \leq 0$$

where $\mathcal{K}$ is a known semialgebraic and $s(\boldsymbol{a},\boldsymbol{x}) = \min_t \max_r I_{t,r}(\boldsymbol{a},\boldsymbol{x})$ can be proved to be a semi-algebraic function. Since the graph of every semi-algebraic function is the projection of a basic semialgebraic set in the lifted space [44, p.148], we know that the graph

$$\{(\boldsymbol{a},\boldsymbol{x},s(\boldsymbol{a},\boldsymbol{x})) \mid (\boldsymbol{a},\boldsymbol{x}) \in \mathcal{K}\} \subset \mathbb{R}^{n+n'+1}$$

is the projection of a basic semialgebraic set $\hat{\mathcal{K}} \subseteq \mathbb{R}^{n+n'+1+u}$, called the *lifting* of $\mathcal{K}$, for some $u \in \mathbb{N}$, i.e. ,

$$\{(\boldsymbol{a},\boldsymbol{x},v) \mid (\boldsymbol{a},\boldsymbol{x}) \in \mathcal{K}, v = s(\boldsymbol{a},\boldsymbol{x})\} = \{(\boldsymbol{a},\boldsymbol{x},v,\boldsymbol{w}) \mid \exists \boldsymbol{w} \in \mathbb{R}^u.(\boldsymbol{a},\boldsymbol{x},v,\boldsymbol{w}) \in \hat{\mathcal{K}}\}.$$

Let us define a function $\hat{f} : \hat{\mathcal{K}} \to \mathbb{R}$ such that $\hat{f}(\boldsymbol{a},\boldsymbol{x},v,\boldsymbol{w}) = v$, then the valid set R_I can be expressed as

$$R_I = \{\boldsymbol{a} \in C_{\boldsymbol{a}} \mid \forall(\boldsymbol{x},v,\boldsymbol{w}) \in \hat{\mathcal{K}}.\hat{f}(\boldsymbol{a},\boldsymbol{x},v,\boldsymbol{w}) \leq 0\},$$

which conforms to the form of Constraint (4), provided that the lifting $\hat{\mathcal{K}}$ is computable (regarding how to compute such a lifting, readers may refer to [44]). In this way, the technique in Section 3.1 can be applied. However, the completeness result is difficult to obtain and needs stronger assumptions in the general semialgebraic case.

In practice, our algorithm is less efficient for general semialgebraic templates compared to polynomial and basic semialgebraic templates. The main reason lies in the lifting process. Applying lifting dramatically increases either the degree of defining polynomials or the number of parameters, sometimes even both.

5 SYNTHESIZING INVARIANTS FROM MASKED TEMPLATES

Masked templates are a special subclass of basic semialgebraic templates defined by parametric polynomial equations and some known polynomial inequations. Regarding these templates, our approach in Section 3 will fail to produce a non-trivial under-approximation of the valid set. To address this problem, we propose a new SDP-based approach. The basic idea is to strengthen the invariant constraints by exploiting the specific structure of parametric equations.

5.1 Masked Templates

Recall that the completeness result of our approach in Section 3 relies on the assumption that the valid set R_I contains an interior point. Unfortunately, whether this assumption holds depends on both the template being used and the program being verified. For instance, consider the following example.

Example 2. *Code 4 is taken from the benchmark set [58], where the purple part represents some unknown polynomial in x, r of degree 2. In other words, we are given a basic semialgebraic template of the form $\{y - I(\boldsymbol{a},x,r) = 0, x \geq 0\}$, where $I(\boldsymbol{a},x,r) = a_1x^2 + a_2xr + a_3r^2 + a_4x + a_5r + a_6$ with parameters $\boldsymbol{a} = (a_1,\ldots,a_6)$. Unfortunately, in this case, the valid set R_I does not contain an interior point because the following constraint*

$$\forall(x,y,r) \in C_{\boldsymbol{x}}.\ y \geq 0 \wedge x = y/2 \wedge r = 0 \implies y = I(\boldsymbol{a},x,r) \wedge x \geq 0$$

implies that $a_1 = a_6 = 0$ and $a_4 = 2$.

Code 4 freire1

```
// Precondition: $\{y \geq 0, x = y/2, r = 0\}$
// Invariant: $\{y = poly[(x, r), 2], x \geq 0\}$
while ($x \geq r$) {
    $x = x - r$;
    $r = r + 1$;
}
// Postcondition: $\{r^2 + r \geq y, r^2 - r \leq y\}$
```

Based on our experience, we find that the assumption tends to hold for programs abstracted from dynamical systems, while it tends to fail for algorithmic programs. The reason is that, invariants of algorithmic programs usually represent some strict symbolic relations. To meet this challenge, we propose a new SDP-based algorithm to synthesize invariants for a class of templates involving equations, which we refer to as masked templates. The related definitions are formulated in Definition 6 and 7.

Before presenting the definitions, we fix some notations. Given a set $J = \{j_1, \dots, j_m\} \subseteq \{1, \dots, n\}$ with $m \leq n$, we denote $\boldsymbol{x}_J = (x_{j_1}, \dots, x_{j_m})$ the projection of $\boldsymbol{x}$ onto indexes in J. Recall that $\boldsymbol{f}_i : \mathbb{R}^n \longrightarrow \mathbb{R}^n, \boldsymbol{x} \mapsto (f_{i,1}(\boldsymbol{x}), \dots, f_{i,n}(\boldsymbol{x}))$ is the assignment function of the i-th branch, we use $\boldsymbol{f}_{i,J} : \mathbb{R}^n \longrightarrow \mathbb{R}^m, \boldsymbol{x} \mapsto (f_{i,j_1}(\boldsymbol{x}), \dots, f_{i,j_m}(\boldsymbol{x}))$ to denote the projection of $\boldsymbol{f}_i$ onto indexes in J.

Definition 6 (Core Set and Core Variables). *Given a program as in Code 1, a nonempty proper subset $J \subsetneq \{1, \dots, n\}$ is called a core set if, for any $i = 1, \dots, k$, $f_{i,j}(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}_J]$ for all $j \in J$. Furthermore, if a variable is in $\boldsymbol{x}_J$, then we call it a core variable, otherwise a non-core variable.*

Definition 7 (Masked Template). *Given a core set $J = \{j_1, \dots, j_m\} \subsetneq \{1, \dots, n\}$, a masked template is a finite collection of polynomials $I_r(\boldsymbol{a}, \boldsymbol{x}_J) : C_{\boldsymbol{a}} \times \mathbb{R}^{n-m} \mapsto \mathbb{R}$ in $\mathbb{R}[\boldsymbol{a}, \boldsymbol{x}]$ and polynomials $I_t(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$, for $r \in R \subsetneq \{1, \dots, n\} \backslash J$, $t \in T \subseteq \{1, \dots, n\}$. Moreover, $I_r(\boldsymbol{a}, \boldsymbol{x}_J)$ is linear in $\boldsymbol{a}$ for every $r \in R$. Given a parameter assignment $\boldsymbol{a}_0 \in \mathbb{R}^{n'}$, the instantiation of the invariant Inv w.r.t. $\boldsymbol{a}_0$ is the set $\{\boldsymbol{x} \mid \bigwedge_{r \in R} x_r = I_r(\boldsymbol{a}_0, \boldsymbol{x}_J) \wedge \bigwedge_t I_{t \in T}(\boldsymbol{x}) \leq 0\}$.*

Definition 6 suggests that the program variables can be partitioned into two nonempty parts, core variables and non-core variables. For a core variable x_j, the assignment $f_{i,j}(\boldsymbol{x})$ can be expressed as a polynomial $p(\boldsymbol{x}_J) \in \mathbb{R}[\boldsymbol{x}_J]$ and is independent of non-core variables. For a non-core variable x_r, we allow $f_{i,r}(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$. In Code 4, it is easy to see that variables x, r are core variables and variable y is a non-core variable. In a masked template, a parametric equation must be of the form $x_r = I_r(\boldsymbol{a}, \boldsymbol{x}_J)$, meaning that a non-core variable can be expressed as a polynomial of core variables.

Remark 5. *In practice, core variables usually correspond to local variables (such as x, r) of programs, while non-core variables correspond to input arguments (such as y) taken by programs. The motivation for the definition of masked templates is that, for most algorithmic programs, the invariants include two parts: (i) equations of the form in which a quantity (usually a non-core variable) is equivalent to an expression involving core variables; (ii) inequalities that are derived from conditionals and monotonicity. Besides, the name of masked templates is inspired from the so-called "masked programs" in [29].*

5.2 Strengthening the Invariant Conditions

In this part, we show how to strengthen the invariant conditions for masked templates based on variable substitution. The resulting constraints have desired properties and can be solved by standard techniques presented in Section 2.3.

Given a masked template as in Definition 7, we explicitly write down the invariant conditions:

$$\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ \boldsymbol{q}_{pre}(\boldsymbol{x}) \le 0 \implies \bigwedge_{r\in R} x_r = I_r(\boldsymbol{a}, \boldsymbol{x}_J) \wedge \bigwedge_{t\in T} I_t(\boldsymbol{x}) \le 0 \tag{25}$$

$$\begin{aligned}\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ &\bigwedge_{r\in R} x_r = I_r(\boldsymbol{a}, \boldsymbol{x}_J) \wedge \bigwedge_{t\in T} I_t(\boldsymbol{x}) \le 0 \wedge \boldsymbol{g}(\boldsymbol{x}) \le 0 \wedge \boldsymbol{c}_i(\boldsymbol{x}) \le 0 \\ &\qquad \implies \bigwedge_{r\in R} f_{i,r}(\boldsymbol{x}) = I_r(\boldsymbol{a}, \boldsymbol{f}_{i,J}(\boldsymbol{x}_J)) \wedge \bigwedge_{t\in T} I_t(\boldsymbol{f}(\boldsymbol{x})) \le 0, \quad i = 1, \ldots, k,\end{aligned} \tag{26}$$

$$\begin{aligned}\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ &\bigwedge_{r\in R} x_r = I_r(\boldsymbol{a}, \boldsymbol{x}_J) \wedge \bigwedge_{t\in T} I_t(\boldsymbol{x}) \le 0 \wedge \bigwedge_{j=1}^{i-1} g_i(\boldsymbol{x}) \le 0 \wedge g_i(\boldsymbol{x}) \ge 0 \\ &\qquad \implies \boldsymbol{q}_{post}(\boldsymbol{x}) \le 0, \quad i = 1, \ldots, m_{\boldsymbol{g}}.\end{aligned} \tag{27}$$

As mentioned in Remark 3, since parameters $\boldsymbol{a}$ occur in the left-hand-side of the implications, we can not directly apply Putinar's Positivstellensatz to obtain sum-of-squares relaxations.

However, after a simple variable substitution procedure, the above constraints can be converted into a desired form. The substitution is based on the following observation in first order logic. Given a formula $\big(y = f(x) \wedge A(x, y)\big) \implies B(x, y)$, if we remove $y = f(x)$ from the left and replace all occurrences of y in $B(x, y)$ by $f(x)$, then a strengthened formula is obtained, i.e.,

$$A(x, y) \implies B(x, f(x)) \models \big(y = f(x) \wedge A(x, y)\big) \implies B(x, y),$$

where f is a function and A, B are formulas in variables x, y.

Using this idea, Constraints (26) and (27) can be strengthened into

$$\begin{aligned}\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ &\bigwedge_{t\in T} I_t(\boldsymbol{x}) \le 0 \wedge \boldsymbol{g}(\boldsymbol{x}) \le 0 \wedge \boldsymbol{c}_i(\boldsymbol{x}) \le 0 \\ &\implies \mathrm{Sub}_R\Bigg(\bigwedge_{r\in R} f_{i,r}(\boldsymbol{x}) = I_r(\boldsymbol{a}, \boldsymbol{f}_{i,J}(\boldsymbol{x}_J)) \wedge \bigwedge_{t\in T} I_t(\boldsymbol{f}(\boldsymbol{x})) \le 0\Bigg), \quad i = 1, \ldots, k,\end{aligned} \tag{28}$$

$$\begin{aligned}\forall \boldsymbol{x} \in C_{\boldsymbol{x}}.\ &\bigwedge_{t\in T} I_t(\boldsymbol{x}) \le 0 \wedge \bigwedge_{j=1}^{i-1} g_i(\boldsymbol{x}) \le 0 \wedge g_i(\boldsymbol{x}) \ge 0 \\ &\implies \mathrm{Sub}_R\Big(\boldsymbol{q}_{post}(\boldsymbol{x}) \le 0\Big), \quad i = 1 \ldots, m_{\boldsymbol{g}},\end{aligned} \tag{29}$$

where $\mathrm{Sub}_R(A)$ replaces all occurrences of x_r by $I_r(\boldsymbol{a}, \boldsymbol{x}_J)$ in formula A. The following theorem is straightforward.

Theorem 6 (Sufficiency). *A valid assignment $\boldsymbol{a}_0$ for Constraints (25),(28) and (29) is also valid for Constraints (25)-(27).*

Since $\boldsymbol{f}_{i,J}(\boldsymbol{x}_J)$ is independent of non-core variables and $I_r(\boldsymbol{a}, \boldsymbol{x})$ is linear in $\boldsymbol{a}$, the substitution will not introduce nonlinear terms of $\boldsymbol{a}$. Therefore, Constraints (25),(28) and (29) conform to the form of Constraint (4) and can be solved by standard techniques in Section 2.3.

Example 2 (Continued). *In this example, x and r are core variables and y is the only non-core variable. The given masked template contains a parametric equation $y = I(\boldsymbol{a}, x, r)$ and a known inequality $x \ge 0$. Therefore, the strengthened invariant conditions of Code 4 should write*

$$\begin{aligned}&\forall (x, y, r) \in C_{\boldsymbol{x}}.\ y \ge 0 \wedge x = y/2 \wedge r = 0 \implies y = I(\boldsymbol{a}, x, r) \wedge x \ge 0, \\ &\forall (x, y, r) \in C_{\boldsymbol{x}}.\ x \ge 0 \wedge x \ge r \implies I(\boldsymbol{a}, x, r) = I(\boldsymbol{a}, x - r, r + 1) \wedge x \ge 0, \\ &\forall (x, y, r) \in C_{\boldsymbol{x}}.\ x \ge 0 \wedge x \le r \implies r^2 + r \ge I(\boldsymbol{a}, x, r) \wedge r^2 - r \le I(\boldsymbol{a}, x, r).\end{aligned}$$

In particular, when restricting sum-of-squares polynomials in the relaxation to be of degree 2*, we obtain an assignment* $\boldsymbol{a}_0$ *that gives*

$$I(\boldsymbol{a}_0, x, r) = 1.999986x + 0.999991r^2 - 0.999990r.$$

After rounding off, we know that $y = 2x + r^2 - r \wedge x \geq 0$ *is an invariant of the program.*

6 EXPERIMENTS

Implementation. We have developed prototypical implementations of our two SDP-based synthesis algorithms in Matlab (R2020a), interfaced with Yalmip [49] and Mosek [4] for solving the underlying sum-of-square relaxations. The implementation and benchmarks can be found at https://github.com/EcstasyH/invSDP. All the experiments are performed on a 2.50GHz Intel Core i9-12900H laptop running 64-bit Windows 11 with 16GB of RAM and Nvidia GeForce RTX 3060 GPU.

Comparison. We wish to compare our tool with the most relevant template-based invariant synthesis tool [13], but,unfortunately, their implementation is not public available. Instead, we primarily compare with QP [29]. QP is a recent template-based synthesis tool that supports both generation of programs and invariants. When focusing on synthesizing invariants, QP adopts the same strategy as in [13] to encode the invariant conditions into quadratic constraints. The difference is that QP uses SMT solvers to solve the quadratic constraints, while [13] uses the solver Loqo [71]. QP also employs additional techniques and heuristics for further speedup. Apart from QP, we also compared with the state-of-the-art machine learning approach LIPuS [75] and the direct solving approach using Z3[22].

Benchmarks. Our two algorithms are applicable to different situations depending on whether the valid set R_I has an interior point. A simple criterion is that, if the (basic semialgebraic) template includes equations, then R_I must have no interior point. Regarding this, we design two sets of benchmarks:

- **Polynomial template benchmarks** include two groups of problem instances, modified programs and dynamical systems. The modified programs are obtained from the corresponding programs in the masked template benchmarks, by relaxing the specifications and adjusting the templates. Such modification ensures that the corresponding valid sets have an interior point. For programs abstracted from dynamical systems in literature, we do not know whether the assumption holds and try to search for an ellipsoid-shaped invariant. In all these examples, we restrict the number of parameters to not exceed 3, and set $h^{(d)}(a)$ to include all monomials in $\boldsymbol{a}$ up to degree d.
- **Masked template benchmarks** include programs from [58], which are mostly polynomial programs that need polynomial invariants. The programs that contain nested loops or need quantified invariants (such as $\gcd(x, y)$) are not supported and removed from the benchmarks. It is worth noting that the invariants of the remaining programs all conform to the definition of masked templates. For these programs, as in Example 2, we set the invariant templates to contain all monomials (of core variables) up to the degree of the real invariants[†].

For the polynomial template benchmarks, we primarily compare our tool with complete approaches QP and Z3. It is worth noting that LIPuS does not support floating-point data. As for the masked template benchmarks, the comparisons encompass QP, LIPuS and Z3. The experimental results are reported in Table 1 and Table 2 respectively.

[†] The masked template benchmarks are more difficult than those used in [29] due to more general templates.

Table 1. Experimental results over polynomial template benchmarks.

	Ours (Sec. 3)			QP[29]		Z3[22]	
Benchmark	$\deg(h(\boldsymbol{a}))$	result	time	result	time	result	time
freire1-1	1	✓	1.1s	✓	3.6s	✓	**0.1s**
freire1-2	2	✓	**4.0s**	timeout	>600s	timeout	>600s
freire1-3	1	✓	**1.1s**	✓	3.1s	✓	61.0s
cohencu-1	2	✓	7.8s	timeout	>600s	✓	**0.2s**
cohencu-2	3	✓	**9.4s**	timeout	>600s	timeout	>600s
cohencu-3	3	✓	10.7s	timeout	>600s	✓	**0.2s**
Example 1	4	✓	**7.5s**	✓	8.7s	timeout	>600s
circuit[3]	3	✓	**4.6s**	timeout	>600s	timeout	>600s
unicycle[65]	≥ 7	timeout	>600s	timeout	>600s	timeout	>600s
overview[20]	≥ 8	timeout	>600s	unsat	**43.5s**	timeout	>600s

$\deg(h(\boldsymbol{a}))$ is the smallest degree d such that $\{\boldsymbol{a} \in [-1,1]^{n'} \mid h^{(d)}(\boldsymbol{a}) \leq 0\} \neq \emptyset$. For our algorithm, "time" includes both the time of solving SDPs and checking whether $h^{(d)}(\boldsymbol{a}) \leq 0$ has a solution. Boldface marks the winner;

Experimental Results over Polynomial Template Benchmarks. For all benchmarks in the first group, our algorithm successfully synthesized a satisfying $h^{(d)}(\boldsymbol{a})$ with $d \leq 3$, but there were a few cases where directly using Z3 outperformed our approach. Note that both QP and Z3 failed in synthesizing a valid invariant for freire1-2 and cohencu-2, this was possibly attributed to the fact that the templates are quadratic in parameters $\boldsymbol{a}$. Even though the problems in the first group are relatively easy, they already pose a challenge for QP and Z3.

As for the second group, both our algorithm and QP demonstrated superior performance compared to Z3 and provided unique advantages. Our algorithm was capable of synthesizing more invariants, but it faced challenges in the unsatisfiable case overview, where we do not have a clear stopping criterion for the searching process. On the other hand, QP utilizes refinement-based heuristics, enabling it to find a certificate of unsatisfiability and report "unsat" in certain cases.

Experimental Results over Masked Template Benchmarks. In this set of benchmarks, our algorithm demonstrated its effectiveness by successfully synthesizing valid invariants for almost all problem instances, with the exception of a single instance. The runtimes of our algorithm for this set of benchmarks were all under 10 seconds, showcasing its practical applicability and efficiency. In the sqrt benchmark, our algorithm reported "unsat" because the strengthened constraint turned out to be infeasible. Overall, these experimental results highlight the robustness and utility of our approach in handling masked template benchmarks and solving program invariant synthesis problems in practice. In comparison, QP managed to solve only five instances, while LIPuS and Z3 (omitted from the table) failed to produce results for all instances in 10 minutes. In order to gain a comprehensive understanding of LIPuS's capacity, we also collaborated with its creators to evaluate our benchmarks within their environment. The results of these tests are showcased in the final column of Table 2, distinguished by parentheses.

Numerical Errors in SDP solving. Since our approaches heavily rely on SDP solvers, it is crucial to address potential numerical errors that can arise during the numerical computation process. These errors could potentially lead to unsound results. For example, we find that the binary search framework presented in Section 4 may fail due to numerical errors when the region of parameters

Table 2. Experimental results over masked template benchmarks.

	Ours (Sec. 5)		QP [29]		LIPuS [75]	
Benchmark	result	time	result	time	result	time†
berkeley	✓	2.4s	timeout	>600s	timeout	>600s
cohencu	✓	1.1s	timeout	>600s	timeout	>600s
cohendiv	✓	1.0s	✓	6.8s	(✓)	(142.0s)
euclidex2	✓	4.8s	timeout	>600s	timeout	>600s
fermat2	✓	1.5s	✓	10.3s	timeout	>600s
firefly	✓	4.7s	timeout	>600s	timeout	>600s
freire1	✓	0.6s	✓	70.4s	(✓)	(460.0s)
freire2	✓	1.0s	timeout	>600s	unsupported	-
illinois	✓	7.3s	timeout	>600s	timeout	>600s
lcm	✓	2.4s	✓	17.1s	timeout	>600s
mannadiv	✓	1.4s	timeout	>600s	timeout	>600s
mesi	✓	2.7s	timeout	>600s	(✓)	(592.5s)
moesi	✓	3.8s	timeout	>600s	(✓)	(117.2s)
petter	✓	0.5s	✓	4.4s	timeout	>600s
readerswriters	✓	3.3s	timeout	>600s	timeout	>600s
sqrt	unsat	1.1s	timeout	>600s	(✓)	(421.0s)
wensley	✓	4.2s	timeout	>600s	unsupported	-
z3sqrt	✓	1.4s	timeout	>600s	unsupported	-

"unsupported": benchmarks containing floating-point variables are not supported by LIPuS.
†: the results in the last column were provided by the authors of LIPuS using their computational environment in [75].

$\boldsymbol{a}$ is too small. In the following, we present some strategies that can be employed to mitigate the effects of numerical errors:

- **Posterior Verification**: One way to address the potential unsoundness of numerical solutions is by using an exact symbolic method to check the soundness of the results obtained from the numerical solvers [20]. Compared with directly solving the constraints, checking the soundness of a certain solution is much easier for symbolic solvers, such as Redlog [25] or Z3 [22]. This approach is relatively easy to employ, and can be used *after* the numerical solutions are given. However, for some larger problems, even checking the soundness of a solution symbolically can be difficult.
- **Precise SDP Solving**: Another approach involves increasing the precision during the SDP solving process. This can be achieved by using multiple-precision or arbitrary-precision solvers [35, 52]. While increasing the precision can help reduce numerical errors, it cannot completely eliminate the possibility of unsoundness caused by these errors. Furthermore, exact SDP solving [31], which relies on symbolic methods, is limited in its ability to handle larger SDP instances and might not be suitable for complex problems.
- **Validated SDP Solving**: A more robust approach is to employ validated SDP solving, as proposed by [62] and extended in [28]. This method involves computing an error bound ϵ for the numerical errors in the results given by solvers. The original constraints are then replaced by their ϵ-strengthened versions (e.g., $A \succeq 0$ to $A + \epsilon I \succeq 0$). By solving the strengthened constraints, one can obtain sound solutions with a guarantee of correctness. This

approach minimizes performance loss while ensuring soundness. However, the strengthening of constraints may lead to a loss of completeness due to the shrinking of the feasible set.

- **Alternative Polynomial Representations**: In our formulation of constraints, a polynomial is represented as a linear combination of monomials. However, this is not the only way to specify polynomials. Besides the monomial basis, there are other non-trivial polynomial bases such as Bernstein basis and Chebyshev basis. While using the monomial basis in our constraints does not alter the number of decision variables, it is worth considering alternative bases for their potential numerical stability during practical computation [8, Section 3.1.5].

In this paper, as the examples used in experiments are relatively small, we apply the symbolic posterior verification methods to check the soundness of the results given by numerical solvers.

7 RELATED WORK

In this section, we present different methods for invariant synthesis and compare our approaches with the most related works.

Constraint Solving. As the constraint solving techniques have made significant advancements in recent years, constraint-solving-based approaches have become increasingly relevant and promising. Specifically, for synthesizing linear invariants, [16] proposes the first complete approach based on Farkas' lemma, which can be seen as a linear version of Putinar's Positivstellensatz, and solves the resulting nonlinear constraints through quantifier elimination. However, due to the double-exponential time complexity of quantifier elimination procedures [21], this method is impractical even for programs in moderate size. Therefore, many works consider using heuristics to solve the nonlinear constraints for better scalability [48, 64].

In the context of synthesizing polynomial invariants for polynomial programs, which is the focus of this paper, [36] introduces the first complete approach. As the derived constraints are rather complex, the final part of solving constraints is usually done by some powerful but time-consuming symbolic methods such as quantifier elimination [37] or SMT solving [69]. Subsequent works can be broadly categorized into two classes: one group focuses on efficiently solving the general constraints of invariant conditions[13, 29, 73], while the other group strengthens the invariant conditions to make the constraints easier to solve[1, 18, 46]. Our two approaches correspond to these two cases, respectively.

Comparison between [13, 29] and our first approach: [13, 29] also employ Putinar's Positivstellensatz to transform the invariant conditions into constraints involving sum-of-squares polynomials, essentially resulting in bilinear matrix inequalities (see Remark 3). To handle these constraints, they further encode them into quadratic constraints and rely on general-purpose solvers. In contrast, our first approach utilizes the technique in [43] to construct high-level sum-of-squares relaxations, which provides an under-approximation of the valid set. Though we still use SMT solvers to extract a valid solution from the under-approximation, the new problem is much easier and can be efficiently handled. Notably, to the best of our knowledge, our approach is the first SDP-based method that offers a completeness guarantee.

Comparison between [1, 46] and our second approach: All these three approaches strengthen the invariant conditions into the form of Constraint (5), allowing for standard sum-of-squares relaxation. However, the major difference between our approach and [1, 46] lies in the restriction on templates. Both [1, 46] limit themselves to polynomial templates, which means the invariant must be a sub-level set of a single polynomial. As a result, these approaches can not synthesize invariants for programs in our masked template benchmarks. On the other hand, in our settings, we deal with a subclass of basic semialgebraic templates that are sufficient for verifying usual programs.

Craig Interpolation. The interpolation-based technique is a power tool for local and modular reasoning. In first order logic, if a formula P implies a formula Q, then there exists a formula I, called an interpolation, such that $P \implies I, I \implies Q$, and every non-logical symbol in I occurs in both P and Q. In the context of program verification, the interpolation I serves as an invariant, although it may not necessarily be inductive. [20, 27, 28] apply this idea in invariant synthesis by first generating an interpolation and then strengthening it into an inductive invariant.

Abstract Interpretation. Abstract interpretation is a widely used and classic method for invariant generation [2, 5, 50, 59, 61]. The process involves fixing an abstract domain and iteratively performing forward propagation until a fixed point is reached, which serves as an invariant. The effectiveness and efficiency of abstract interpretation approaches heavily rely on the choice of abstract domains. Different abstract domains may lead to varying levels of precision and scalability in the obtained invariants. In most cases, there is no theoretical guarantee on the accuracy of generated invariants. In other words, it is uncertain whether the obtained invariant is strong enough to accurately represent the desired properties of the system under analysis. The absence of such guarantees necessitates careful consideration of the abstract domains and fine-tuning of the analysis to strike a balance between precision and tractability.

Recurrence Analysis. Approaches based on recurrence analysis consider loops as recurrence relations and aim to compute closed-form solutions for program variables [11, 33, 34, 38, 40]. However, not all recurrence relations have closed-form solutions, which is why these approaches typically focus on synthesizing equality invariants for a class of "solvable" loops. The invariant synthesis problem is also closely related to the Skolem problem in recurrence analysis [47], which involves deciding whether the values of a linear recursion sequence contain at least one zero.

Other Methods. Recently, methods based on machine learning[30, 67, 74, 75] and logical inference [23, 39, 54, 66] have shown significant promise. Beyond classic programs, the problem of invariant generation is also being actively explored in the context of hybrid systems [19, 68, 72] and stochastic systems [6, 7, 15], combining techniques from differential equations and probability theory.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present two novel SDP-based approaches to synthesize invariants from polynomial templates and masked templates. The first algorithm employs an SDP technique from robust optimization [43] to under-approximate the valid set and provides a weak completeness guarantee. The second algorithm relies on identifying special equality structures in program invariants. In summary, our approaches significantly expand the boundaries of constraint-solving-based invariant synthesis methods, offering improvements in both efficiency and effectiveness.

Currently, our first approach becomes impractical when the template includes an excessive number of parameters. This limitation arises because the size of sum-of-squares polynomials in the relaxations depends on the total number of program variables and parameters. To address this problem, we consider exploring the internal structure of the constraints to improve the algorithm. Moreover, we plan to extend the techniques presented in this paper to invariant synthesis for hybrid systems and probabilistic programs.

ACKNOWLEDGMENTS

The authors would like to thank Shiwen Yu for his assistance in producing the benchmark results of LIPuS.

REFERENCES

[1] Assalé Adjé, Pierre-Loïc Garoche, and Victor Magron. 2015. Property-based Polynomial Invariant Generation Using Sums-of-Squares Optimization. In *Static Analysis - 22nd International Symposium (Lecture Notes in Computer Science, Vol. 9291)*. Springer, 235–251. https://doi.org/10.1007/978-3-662-48288-9_14

[2] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. 2012. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:1)2012

[3] Mahathi Anand, Vishnu Murali, Ashutosh Trivedi, and Majid Zamani. 2021. Safety Verification of Dynamical Systems via k-Inductive Barrier Certificates. In *2021 60th IEEE Conference on Decision and Control*. IEEE, 1314–1320. https://doi.org/10.1109/CDC45484.2021.9682889

[4] Erling D. Andersen, Cornelis Roos, and Tamás Terlaky. 2003. On implementing a primal-dual interior-point method for conic quadratic optimization. *Mathematical Programming* 95, 2 (2003), 249–277.

[5] Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. 2005. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In *Static Analysis, 12th International Symposium (Lecture Notes in Computer Science, Vol. 3672)*, Chris Hankin and Igor Siveroni (Eds.). Springer, 19–34. https://doi.org/10.1007/11547662_4

[6] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Computer Aided Verification - 34th International Conference (Lecture Notes in Computer Science, Vol. 13371)*. Springer, Haifa, Israel, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

[7] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference (Lecture Notes in Computer Science, Vol. 13994)*. Springer, Paris, France, 410–429. https://doi.org/10.1007/978-3-031-30820-8_25

[8] Grigoriy Blekherman, Pablo A Parrilo, and Rekha R Thomas. 2012. *Semidefinite optimization and convex algebraic geometry*. SIAM.

[9] Jacek Bochnak, Michel Coste, and Marie-Françoise Roy. 1998. *Real algebraic geometry*. Vol. 36. Springer Science & Business Media.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Computer Aided Verification, 17th International Conference (Lecture Notes in Computer Science, Vol. 3576)*. Springer, 491–504. https://doi.org/10.1007/11513988_48

[11] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas W. Reps. 2020. Templates and recurrences: better together. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 688–702. https://doi.org/10.1145/3385412.3386035

[12] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference (Lecture Notes in Computer Science, Vol. 9636)*. Springer, 387–393. https://doi.org/10.1007/978-3-662-49674-9_22

[13] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 672–687. https://doi.org/10.1145/3385412.3385969

[14] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. 2022. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In *Computer Aided Verification - 34th International Conference (Lecture Notes in Computer Science, Vol. 13371)*. Springer, 55–78. https://doi.org/10.1007/978-3-031-13185-1_4

[15] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Computer Aided Verification - 27th International Conference (Lecture Notes in Computer Science, Vol. 9206)*. Springer, San Francisco, CA, USA, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

[16] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference (Lecture Notes in Computer Science, Vol. 2725)*. Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[17] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *Computer Aided Verification, 20th International Conference (Lecture Notes in Computer Science, Vol. 5123)*. Springer, 328–340. https://doi.org/10.1007/978-3-540-70545-1_32

[18] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (Lecture Notes in Computer Science, Vol. 3385)*. Springer, 1–24. https://doi.org/10.1007/978-3-540-30579-8_1

[19] Liyun Dai, Ting Gan, Bican Xia, and Naijun Zhan. 2017. Barrier certificates revisited. *Journal of Symbolic Computation* 80 (2017), 62–86. https://doi.org/10.1016/j.jsc.2016.07.010

[20] Liyun Dai, Bican Xia, and Naijun Zhan. 2013. Generating Non-linear Interpolants by Semidefinite Programming. In *Computer Aided Verification - 25th International Conference (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 364–380. https://doi.org/10.1007/978-3-642-39799-8_25

[21] James H Davenport and Joos Heintz. 1988. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation* 5, 1-2 (1988), 29–35.

[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[23] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 443–456. https://doi.org/10.1145/2509136.2509511

[24] Andreas Dolzmann and Thomas Sturm. 1996. *Redlog user manual.*

[25] Andreas Dolzmann and Thomas Sturm. 1997. Redlog: Computer algebra meets computer logic. *Acm Sigsam Bulletin* 31, 2 (1997), 2–9.

[26] Robert W Floyd. 1967. Assigning meanings to programs. *Mathematical Aspects of Computer Science* 19, 19-32 (1967), 1.

[27] Ting Gan, Liyun Dai, Bican Xia, Naijun Zhan, Deepak Kapur, and Mingshuai Chen. 2016. Interpolant Synthesis for Quadratic Polynomial Inequalities and Combination with EUF. In *Automated Reasoning - 8th International Joint Conference (Lecture Notes in Computer Science, Vol. 9706)*. Springer, 195–212. https://doi.org/10.1007/978-3-319-40229-1_14

[28] Ting Gan, Bican Xia, Bai Xue, Naijun Zhan, and Liyun Dai. 2020. Nonlinear Craig Interpolant Generation. In *Computer Aided Verification - 32nd International Conference (Lecture Notes in Computer Science, Vol. 12224)*. Springer, 415–438. https://doi.org/10.1007/978-3-030-53288-8_20

[29] Amir Kafshdar Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit J. Motwani. 2023. Algebro-geometric Algorithms for Template-Based Synthesis of Polynomial Programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 727–756. https://doi.org/10.1145/3586052

[30] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 1112–1127. https://doi.org/10.1145/3385412.3386016

[31] Didier Henrion, Simone Naldi, and Mohab Safey El Din. 2021. Exact algorithms for semidefinite programs with degenerate feasible set. *Journal of Symbolic Computation* 104 (2021), 942–959. https://doi.org/10.1016/j.jsc.2020.11.001

[32] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[33] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial Invariants for Affine Programs. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 530–539. https://doi.org/10.1145/3209108.3209142

[34] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2018. Invariant Generation for Multi-Path Loops with Polynomial Assignments. In *Verification, Model Checking, and Abstract Interpretation - 19th International (Lecture Notes in Computer Science, Vol. 10747)*. Springer, 226–246. https://doi.org/10.1007/978-3-319-73721-8_11

[35] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. 2017. Implementation and Performance Evaluation of an Extended Precision Floating-Point Arithmetic Library for High-Accuracy Semidefinite Programming. In *24th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 27–34. https://doi.org/10.1109/ARITH.2017.18

[36] Deepak Kapur. 2005. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications, 23.-28. October 2005 (Dagstuhl Seminar Proceedings, Vol. 05431)*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/511

[37] Deepak Kapur. 2006. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity* 19, 3 (2006), 307–330.

[38] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 54:1–54:33. https://doi.org/10.1145/3158142

[39] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. 2022. Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 338–356. https://doi.org/10.1007/978-3-030-99524-9_18

[40] Laura Kovács. 2008. Reasoning Algebraically About P-Solvable Loops. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008 (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 249–264. https://doi.org/10.1007/978-3-540-78800-3_18

[41] Jean Bernard Lasserre. 2000. Global Optimization with Polynomials and the Problem of Moments. *SIAM Journal on Optimization* 11 (2000), 796–817. https://api.semanticscholar.org/CorpusID:16740871

[42] Jean Bernard Lasserre. 2009. *Moments, positive polynomials and their applications*. Vol. 1. World Scientific.

[43] Jean B Lasserre. 2015. Tractable approximations of sets defined with quantifiers. *Mathematical Programming* 151, 2 (2015), 507–527.

[44] Jean B Lasserre and Mihai Putinar. 2012. Positivity and optimization: beyond polynomials. In *Handbook on Semidefinite, Conic and Polynomial Optimization*. Springer, 407–434.

[45] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. 2017. FiB: squeezing loop invariants by interpolation between Forward/Backward predicate transformers. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 793–803. https://doi.org/10.1109/ASE.2017.8115690

[46] Wang Lin, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2014. Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. *Frontiers of Computer Science* 8, 2 (2014), 192–202. https://doi.org/10.1007/s11704-014-3150-6

[47] Richard Lipton, Florian Luca, Joris Nieuwveld, Joël Ouaknine, David Purser, and James Worrell. 2022. On the Skolem Problem and the Skolem Conjecture. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science*, Christel Baier and Dana Fisman (Eds.). ACM, 5:1–5:9. https://doi.org/10.1145/3531130.3533328

[48] Hongming Liu, Hongfei Fu, Zhiyong Yu, Jiaxin Song, and Guoqiang Li. 2022. Scalable linear invariant generation with Farkas' lemma. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 204–232. https://doi.org/10.1145/3563295

[49] J. Löfberg. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. In *CACSD'04*. 284–289.

[50] Markus Müller-Olm and Helmut Seidl. 2004. Computing polynomial program invariants. *Inform. Process. Lett.* 91, 5 (2004), 233–244. https://doi.org/10.1016/j.ipl.2004.05.004

[51] Markus Müller-Olm and Helmut Seidl. 2004. A Note on Karr's Algorithm. In *Automata, Languages and Programming: 31st International Colloquium (Lecture Notes in Computer Science, Vol. 3142)*. Springer, 1016–1028. https://doi.org/10.1007/978-3-540-27836-8_85

[52] Maho Nakata. 2010. A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and-DD.. In *2010 IEEE International Symposium on Computer-Aided Control System Design*. IEEE, 29–34.

[53] Peter Naur. 1966. Proof of algorithms by general snapshots. *BIT Numerical Mathematics* 6, 4 (1966), 310–316.

[54] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. 2022. Induction duality: primal-dual search for invariants. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498712

[55] Pablo A Parrilo. 2000. *Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization*. California Institute of Technology.

[56] Pablo A Parrilo and Rekha R Thomas. 2020. *Sum of Squares: Theory and Applications*. Vol. 77. American Mathematical Soc.

[57] Mihai Putinar. 1993. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984.

[58] Enric Rodríguez-Carbonell. 2016. Some programs that need polynomial invariants in order to be verified. https://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html

[59] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *Static Analysis, 11th International Symposium (Lecture Notes in Computer Science, Vol. 3148)*. Springer, 280–295. https://doi.org/10.1007/978-3-540-27864-1_21

[60] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming* 64, 1 (2007), 54–75.

[61] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476.

[62] Pierre Roux, Yuen-Lam Voronin, and Sriram Sankaranarayanan. 2018. Validating numerical semidefinite programming solvers for polynomial invariants. *Formal Methods in System Design* 53, 2 (2018), 286–312.

[63] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. 2004. Constructing Invariants for Hybrid Systems. In *Hybrid Systems: Computation and Control, 7th International Workshop (Lecture Notes in Computer Science, Vol. 2993)*. Springer, 539–554. https://doi.org/10.1007/978-3-540-24743-2_36

[64] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *Static Analysis, 11th International Symposium (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[65] Mohamed Amin Ben Sassi and Antoine Girard. 2012. Controller synthesis for robust invariance of polynomial dynamical systems using linear programming. *Systems & control letters* 61, 4 (2012), 506–512.

[66] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Computer Aided Verification - 26th International Conference (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 88–105. https://doi.org/10.1007/978-3-319-08867-9_6

[67] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *Computer Aided Verification - 32nd International Conference (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 151–164. https://doi.org/10.1007/978-3-030-53291-8_9

[68] William Simmons and André Platzer. 2023. Differential Elimination and Algebraic Invariants of Polynomial Dynamical Systems. *CoRR* abs/2301.10935 (2023). https://doi.org/10.48550/arXiv.2301.10935 arXiv:2301.10935

[69] Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 223–234. https://doi.org/10.1145/1542476.1542501

[70] Alfred Tarski. 1951. *A decision method for elementary algebra and geometry*. University of California Press, Berkeley.

[71] Robert J. Vanderbei and Version. 2006. LOQO User's Manual – Version 4.05.

[72] Qiuye Wang, Mingshuai Chen, Bai Xue, Naijun Zhan, and Joost-Pieter Katoen. 2022. Encoding inductive invariants as barrier certificates: Synthesis via difference-of-convex programming. *Information and Computation* 289, Part (2022), 104965. https://doi.org/10.1016/j.ic.2022.104965

[73] Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. 2010. Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China* 4, 1 (2010), 1–16. https://doi.org/10.1007/s11704-009-0074-7

[74] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.

[75] Shiwen Yu, Ting Wang, and Ji Wang. 2023. Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, René Just and Gordon Fraser (Eds.). ACM, 175–187. https://doi.org/10.1145/3597926.3598047

[76] Shaowei Zhu and Zachary Kincaid. 2021. Termination analysis without the tears. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 1296–1311. https://doi.org/10.1145/3453483.3454110