Nonlinear Invariant Generation for Polynomial Loops

Abstract. Invariant generation is crucial and challenging in program analysis and verification. In this paper, we propose a novel invariant generation method for polynomial loops through constraint solving. We reformulate the problem of invariant generation as a single-objective optimization problem and name it BSOS programming, which is short for bilinear sum of squares programming. The presence of bilinear terms makes the problem nonconvex and difficult to solve. To address this challenge, we develop an alternative iterative numerical optimization method based on sum-of-squares (SOS) programming. Nevertheless, iteration-based numerical approaches are customarily sensitive to the selection of initial values. We propose an effective initial value construction method that uses the Craig interpolant to address this issue. Our method can be proved to be correct, terminated, and semicomplete under the Archimedean condition. Experimental results on various benchmarks demonstrate the effectiveness and efficiency of our approach.

Keywords: Invariant · While loop · Program verification · Semidefinite programming · Bilinear SOS.

1 Introduction

Ensuring the correctness of heterogeneous programs is a significant and challenging task. A prominent technique closely associated with this is Floyd-Hoare-Naur's assertion method based on Hoare logic [24], in which the core concept is the Hoare triple with the following form:

$$\{Pre\}Prog\{Post\}, \tag{1}$$

where $Prog$ is a segment of a program, Pre and $Post$ are two predicates constituted by the state variables. Suppose $Prog$ receives an input state that satisfies the specified input assumption Pre. Upon execution, it produces an output state that meets the output assertion $Post$. We then conclude that the Hoare triple holds or is valid. The validity of the Hoare triple is of utmost importance for the correctness of programs. The question of how to ensure the validity of a Hoare triple is a classical problem in program verification.

Invariant generation is an essential method to ensure the validity of a Hoare triple. If an invariant of the Hoare triple can be found, then the Hoare triple is valid. Among diverse programs, invariant generation for programs with loops, i.e., loop invariant generation, is one of the most difficult tasks in program verification. Loop invariant generation involves identifying the properties that remain true throughout the execution of the loops. Complex loop dependencies

and nonlinear conditions throughout repeated iterations further complicate the task of identifying loop invariants. Therefore, loop invariant generation has been an important but challenging problem in program analysis and verification, especially when dealing with loop programs with nonlinear conditions.

As we know, when dealing with nonlinear constraints, quantifier elimination is a standard solution, but this approach is inefficient in practice, as it suffers from a double exponential time complexity [14,28]. In search of more efficient solutions, many researchers have turned to heuristic approaches when solving nonlinear constraints [35,53]. Several studies investigate how to efficiently generate polynomial invariants with polynomial loop programs [2,6,11,21,34,63]. They are all faced with solving a nonconvex problem produced by invariant constraints. In order to obtain an efficient invariant synthesis algorithm, they either approximate invariant constraints by constraint relaxation or transform the constraints into easier-to-solve ones.

In this paper, we focus on generating polynomial invariants for polynomial loop programs. We propose using the SOS relaxation to transform the invariant conditions into SOS polynomial constraints. Note that the SOS constraint system is not a standard SOS programming problem but a non-convex problem, since a product of two unknown polynomials appears in the constraints. This SOS constraint solving problem is called the BSOS problem or BSOS programming. In order to solve the BSOS problem for a nonlinear invariant effectively and efficiently, we propose a novel approach by leveraging constraint solving and Craig interpolant techniques. In this approach, we suppose a polynomial inequality $h(\mathbf{x}) \geq 0$ (a template polynomial with unknown coefficients) to be an invariant for a given polynomial loop, then a polynomial constraint system w.r.t. $h(\mathbf{x})$ can be obtained from the invariant conditions. Then, the BSOS constraint system can be constructed using SOS relaxation. Nonetheless, due to the nonconvex nature of the BSOS constraint system, classic algorithms based on the interior point method, such as SOS programming and semidefinite programming (SDP), cannot be directly utilized. To this end, we transform the BSOS constraint system into a single-objective optimization problem by introducing a penalty term in the constraints. Subsequently, we propose an alternative iterative optimization method to solve the single-objective optimization problem. Considering the influence of initial value selection on numerical optimization methods, we propose a method based on the Craig interpolation technique to select a suitable initial value for our alternative iterative optimization method.

Motivation Example. Consider the code of a discrete-time dynamical system:

Listing 1 A discrete-time dynamical system.

```
// Precondition: {1 − x1^2 − x2^2 ≥ 0}
while (3 − x1^2 − x2^2 ≥ 0) {
    x1 = x1^2 + x2 − 1;
    x2 = x2 + x1x2 + 1;
}
// Postcondition: {x2 − 0.2x1^2 + 1 ≥ 0}
```

We try to use some state-of-the-art methods, including Z3 [15], PolySynth [21], invSDP [63], Prajna's classic iterative method [42] and BCDC [61], to generate an invariant for Code 1. We set a polynomial template with a total degree of 2 for the invariant. Neither Z3 nor PolySynth can obtain any results in 600 seconds. Prajna's method generates a meaningless result "0" as a candidate invariant, and invSDP reports "no solution". BCDC generates a candidate invariant as

$$3.91 + 3.65x_1 + 27.24x_1^2 + 0.91x_2 + 4.63x_1x_2 + 1.90x_2^2 \geq 0. \tag{2}$$

This candidate invariant is excluded after post verification. For LaM4Inv, all candidate invariants it generates are verified to be invalid.

In some existing studies [13,18,33], the Craig interpolant technique and the guess-verify strategy are applied in invariant generation. In their methods, the interpolant serves directly as an invariant candidate, and is subsequently verified to determine whether it is a true invariant. Synthesizing invariants for this code is beyond the ability of the existing approaches [13,18,33]. Using their approaches, one may successfully obtain an interpolant as following

$$3.45 + 0.52x_2 - 1.60x_1^2 - 1.71x_2^2 \geq 0. \tag{3}$$

However, it can be checked that this interpolant is not a true invariant because the inductive condition, i.e. condition (2) in Definition 1, does not hold.

Using our approach, we take the interpolant in Eq (3) as an initial value of the optimal problem (12); then, the following invariant can be found by our approach after only one iteration

$$1.31 + 0.04x_1 + 1.19x_2 - 0.36x_1^2 - 0.07x_2^2 + 0.03x_1x_2 \geq 0.$$

This candidate invariant can be proved to be a true invariant.

Our Contribution. In summary, our key contributions include the following:

- We present a novel method to translate the invariant generation problem into a constraint-solving problem with bilinear SOS constraints using SOS relaxation. Furthermore, it transforms the problem into a single-objective optimization problem by introducing a penalty in the constraints.
- We introduce an innovative alternative iterative optimization algorithm to solve the single-objective optimization problem. In each iteration, the algorithm solves two classic SOS programming problems. Theoretically, the optimal value of the single-objective function will never increase, ensuring a monotonically decreasing approach to the optimal solution.
- We propose a systematic and heuristic method to obtain a suitable initial value for the single-objective optimization based on the Craig interpolant.
- We prove the correctness, termination and semicompleteness of our method under the Archimedean condition.

The remaining sections of the paper are organized as follows. Preliminaries are introduced in Section 2. Section 3 transforms the invariant generation problem into an optimization problem and proposes an iterative optimization-based method to solve it. Section 4 finds a suitable initial value for the optimization problem. Section 5 presents the corresponding algorithm. Experimental results are shown in Section 6. Finally, we discuss related work in Section 7 and conclude this paper in Section 8.

2 Preliminaries

In this section, we introduce some fundamental concepts. Let $\mathbb{N}$ and $\mathbb{R}$ be the sets of positive integers and real numbers, respectively. $\mathbb{R}[\mathbf{x}]$ denotes the polynomial ring over $\mathbb{R}$ w.r.t. $\mathbf{x} = (x_1, ..., x_d)$, $d \in \mathbb{N}$. Vectors or vector functions are denoted by boldface letters. Suppose P is a predicate formula containing variable $\mathbf{x}$, then $P(\mathbf{x})$ means $\mathbf{x}$ satisfies P; if P contains other variables such as $\mathbf{y}$, then $P(\mathbf{x})$ means that there exists some $\mathbf{y}$ such that $(\mathbf{x}, \mathbf{y})$ satisfies P.

In this paper, we focus on invariant generation for the Hoare triple presented in the following formula (4), in which all of the input assumption, output assertion, loop condition, and loop body are polynomial formulae.

$$\begin{array}{ll} \{Pre\} & /*\, Assumption\, */ \\ while(Cond)\{Body\} & /*\, LoopBody\, */ \\ \{Post\} & /*\, Assertion\, */ \end{array} \tag{4}$$

To establish the validity of the polynomial loop (4), common approach is to find an invariant for the loop. In general, invariant generation methods try to find an inductive invariant [57]. In this paper, we will use the term "invariant" to refer to the concept of inductive invariant. An inductive invariant is defined as follows:

Definition 1 (Invariant). *A formula $Inv(\mathbf{x})$ is an invariant of the program (4) if it satisfies the following three conditions:*

1). $Pre(\mathbf{x}) \models Inv(\mathbf{x})$;
2). $Inv(\mathbf{x}) \wedge Cond(\mathbf{x}) \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) \models Inv(\mathbf{y})$;
3). $Inv(\mathbf{x}) \wedge \neg Cond(\mathbf{x}) \models Post(\mathbf{x})$.

These conditions are invariant conditions and are also called the precondition, inductive condition, and postcondition.

The rest of the paper defines $Pre(\mathbf{x})$, $\neg Post(\mathbf{x})$, $Cond(\mathbf{x})$ and $Body$ as:

$$\begin{array}{l} Pre(\mathbf{x}) : f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0; \\ Cond(\mathbf{x}) : c(\mathbf{x}) \geq 0; \quad Body : \mathbf{x} = \mathbf{p}(\mathbf{x}); \\ \neg Post(\mathbf{x}) : g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0, \end{array} \tag{5}$$

Here, we use $\neg Post(\mathbf{x})$ rather than $Post(\mathbf{x})$ in formula (5), which will bring many benefits to the proof, and the original problem could be easily restored. $f_1(\mathbf{x})$,

..., $f_m(\mathbf{x})$, $c(\mathbf{x})$, $\mathbf{p}(\mathbf{x}) = (p_1(\mathbf{x}), ..., p_d(\mathbf{x}))$, $g_1(\mathbf{x})$, ..., $g_n(\mathbf{x})$ are all polynomials. As mentioned above, the polynomial loop (4) is valid when, given an input state that satisfies the input assumption $Pre(\mathbf{x})$, once the loop terminates, the resulting output shall satisfy the output assertion $Post(\mathbf{x})$.

Next, we introduce some notions from real algebraic geometry that will be used in the following; for more details, see [5].

Definition 2 (SOS). *A polynomial $f(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ is said to be sum of squares or SOS if there exist some polynomials $p_1(\mathbf{x}), ..., p_s(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ such that $f(\mathbf{x}) = \sum_{i=1}^{s} p_i^2(\mathbf{x})$. We use $\sum \mathbb{R}[\mathbf{x}]^2$ to denote the set of SOS polynomials in $\mathbb{R}[\mathbf{x}]$.*

Definition 3 (Quadratic Module [38]). *A set $\mathcal{M} \subset \mathbb{R}[\mathbf{x}]$ is called a quadratic module if it contains 1 and is closed under addition and multiplication with squares, i.e. $1 \in \mathcal{M}$, $\mathcal{M} + \mathcal{M} \subset \mathcal{M}$, and for any $p \in \mathbb{R}[\mathbf{x}]$, $p^2\mathcal{M} \subset \mathcal{M}$.*

Let $F = \{f_1, ..., f_m\} \subset \mathbb{R}[\mathbf{x}]$ be a finite set, the quadratic module $\mathcal{M}_{\mathbf{x}}(F)$ generated by F is

$$\mathcal{M}_{\mathbf{x}}(F) = \{\sum_{i=0}^{m} \delta_i f_i \mid \delta_i \in \sum \mathbb{R}[\mathbf{x}]^2, f_0 = 1\}. \tag{6}$$

Definition 4 (Archimedean). *Let $\mathcal{M} \subset \mathbb{R}[\mathbf{x}]$ be a quadratic module. $\mathcal{M}$ is said to be Archimedean if there exists $a > 0$ such that $a - x_1^2 - \cdots - x_d^2 \in \mathcal{M}$.*

Let $\phi(\mathbf{x})$ be a formula defined by some polynomial inequalities as following

$$\phi(\mathbf{x}) : f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0. \tag{7}$$

The formula $\phi(\mathbf{x})$ is said to be Archimedean if $\mathcal{M}_{\mathbf{x}}(f_1, ..., f_m)$ is Archimedean. We also use $\mathcal{M}_{\mathbf{x}}(\phi)$ to denote $\mathcal{M}_{\mathbf{x}}(f_1, ..., f_m)$.

Theorem 1 (Putinar's Positivstellensatz [44]). *Let $f_1(\mathbf{x}), ..., f_m(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$. If $\mathcal{M}_{\mathbf{x}}(f_1, ..., f_m)$ is Archimedean and $f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models p(\mathbf{x}) > 0$. Then $p \in \mathcal{M}_{\mathbf{x}}(f_1, ..., f_m)$.*

From Theorem 1 and formula (6) we have that if $\mathcal{M}_{\mathbf{x}}(f_1, ..., f_m)$ is an Archimedean quadratic module and $f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models p(\mathbf{x}) > 0$, then there exist $\delta_0, ..., \delta_m \in \sum \mathbb{R}[\mathbf{x}]^2$ such that $p(\mathbf{x}) = \sum_{i=1}^{m} \delta_i f_i + \delta_0$.

SOS relaxations. In Definition 1, the invariant conditions consist of three implications, which are difficult to solve directly. A common way is to reformulate these implications into polynomial constraints by SOS relaxations. Let $f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models h(\mathbf{x}) \geq 0$ be a implication relation and $f_1(\mathbf{x}), ..., f_m(\mathbf{x}), h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, the SOS relaxation of this implication relation is:

$$\exists\ \delta_0, ..., \delta_m \in \sum \mathbb{R}[\mathbf{x}]^2.\ h(\mathbf{x}) = \sum_{i=1}^{m} \delta_i f_i + \delta_0. \tag{8}$$

Definition 5 (Craig interpolant). *Let ϕ and ψ be two formulas and $\phi \wedge \psi \models \bot$. A formula I is said to be a Craig interpolant of ϕ and ψ if the following three conditions hold: $\phi \models I$; $I \wedge \psi \models \bot$; I only contains the common variables of ϕ and ψ.*

Let ϕ and ψ be two formulas defined as follows

$$\begin{aligned} \phi(\mathbf{x},\mathbf{y}):\ & f_1(\mathbf{x},\mathbf{y}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x},\mathbf{y}) \geq 0, \\ \phi(\mathbf{x},\mathbf{z}):\ & g_1(\mathbf{x},\mathbf{z}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x},\mathbf{z}) \geq 0, \end{aligned} \tag{9}$$

where $f_1(\mathbf{x},\mathbf{y}),...,f_m(\mathbf{x},\mathbf{y}) \in \mathbb{R}[\mathbf{x},\mathbf{y}]$ and $g_1(\mathbf{x},\mathbf{z}),...,g_n(\mathbf{x},\mathbf{z}) \in \mathbb{R}[\mathbf{x},\mathbf{z}]$. Suppose $\phi(\mathbf{x},\mathbf{y}) \wedge \psi(\mathbf{x},\mathbf{z}) \models \bot$. If $\mathcal{M}_{\mathbf{x},\mathbf{y}}(f_1,...,f_m)$ and $\mathcal{M}_{\mathbf{x},\mathbf{z}}(g_1,...,g_n)$ are Archimedean quadratic modules, utilizing the method in [18], a Craig interpolant $h(\mathbf{x}) > 0$ can be obtained by solving the following SOS programming problem.

$$\text{find: } h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$$
$$\text{s.t.} \begin{cases} h(\mathbf{x}) - 1 - \sum_{i=1}^{m} u_i(\mathbf{x},\mathbf{y}) f_i(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \\ -h(\mathbf{x}) - 1 - \sum_{i=1}^{n} v_i(\mathbf{x},\mathbf{z}) g_i(\mathbf{x},\mathbf{z}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{z}]^2, \\ u_1(\mathbf{x},\mathbf{y}),...,u_m(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \\ v_1(\mathbf{x},\mathbf{z}),...,v_n(\mathbf{x},\mathbf{z}) \sum \mathbb{R}[\mathbf{x},\mathbf{z}]^2. \end{cases} \tag{10}$$

3 Invariant Generation with Iterative Optimization

In this section, we first translate the invariant generation problem to a bilinear SOS constraint system by SOS relaxation. Then, a single-objective optimization problem is solved by introducing a penalty term to the constraints. Ultimately, we propose a method to solve the optimization problem iteratively. Due to the limited space, all the detailed proofs have been attached in the Appendix Section.

3.1 The Objective Optimization Problem

Theorem 2. *$h(\mathbf{x}) \geq 0$ is an invariant of the polynomial program (4) if there exist $m+n+3$ SOS polynomials $u_1(\mathbf{x}), \ldots, u_m(\mathbf{x}), v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_n(\mathbf{x})$, $w_h(\mathbf{x},\mathbf{y})$, $w_c(\mathbf{x},\mathbf{y})$ and d polynomials $w_1(\mathbf{x},\mathbf{y}), \ldots, w_d(\mathbf{x},\mathbf{y})$ such that the following three conditions hold,*

i. $h(\mathbf{x}) - \sum_{i=1}^{m} u_i(\mathbf{x}) f_i(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2$,
ii. $h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x},\mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x},\mathbf{y}) h(\mathbf{x}) - w_c(\mathbf{x},\mathbf{y}) c(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2$,
iii. $-h(\mathbf{x}) - \sum_{i=1}^{n} v_i(\mathbf{x}) g_i(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) - \epsilon \in \sum \mathbb{R}[\mathbf{x}]^2$.

where $\mathbf{x} \in \mathbb{R}^d$ *is the program variable,* $\mathbf{y} \in \mathbb{R}^d$ *is an auxiliary variable corresponding to the loop body* $y_i = p_i(x)(i = 1, 2, ..., d)$*, having the same dimension as* $\mathbf{x}$*, and* $\epsilon > 0$*.*

From the above Theorem 2, an invariant $h(\mathbf{x}) \geq 0$ can be obtained by solving the following SOS constraint system:

$$\begin{cases} h(\mathbf{x}) - \sum_{i=1}^{m} u_i(\mathbf{x}) f_i(\mathbf{x}), u_1(\mathbf{x}), \ldots, u_m(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x}, \mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x}, \mathbf{y}) h(\mathbf{x}) - w_c(\mathbf{x}, \mathbf{y}) c(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \\ -h(\mathbf{x}) - \sum_{j=1}^{n} v_j(\mathbf{x}) g_j(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) - \epsilon, v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_n(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ w_h(\mathbf{x}, \mathbf{y}), w_c(\mathbf{x}, \mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \\ w_1(\mathbf{x}, \mathbf{y}), \ldots, w_d(\mathbf{x}, \mathbf{y}) \in \mathbb{R}[\mathbf{x}, \mathbf{y}], h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]. \end{cases} \tag{11}$$

This is an SOS constraint system (11) that cannot be solved directly by any SOS programming solver or SDP solver. A standard method for solving an SOS programming problem is presetting the degrees of the unknown polynomials, and translating the SOS constraints concerning the unknown polynomials to SDP constraints. An SDP constraint is a matrix inequality that can be solved efficiently using SDP programming tools [36]. The coefficients of the unknown polynomials can be determined by solving the SDP constraints, providing the complete solution to the SOS constraint system [18,32,61]. Unfortunately, due to the presence of the bilinear term $w_h(\mathbf{x}, \mathbf{y}) h(\mathbf{x})$, where both $w_h(\mathbf{x}, \mathbf{y})$ and $h(\mathbf{x})$ are unknown polynomials, the SOS constraint system (11) cannot be solved directly using any SDP solver.

In order to solve the constraint system (11) and obtain $h(\mathbf{x})$, we first introduce a single-objective optimization problem as follows.

$$\begin{aligned} \min \quad & \lambda \\ s.t. \quad & \begin{cases} h(\mathbf{x}) - \sum_{i=1}^{m} u_i(\mathbf{x}) f_i(\mathbf{x}), u_1(\mathbf{x}), ..., u_m(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x}, \mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x}, \mathbf{y}) h(\mathbf{x}) - \\ \qquad w_c(\mathbf{x}, \mathbf{y}) c(\mathbf{x}) + \lambda t(\mathbf{x}, \mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \\ -h(\mathbf{x}) - \sum_{j=1}^{n} v_j(\mathbf{x}) g_j(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) - \epsilon, v_0(\mathbf{x}), ..., v_n(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ w_h(\mathbf{x}, \mathbf{y}), w_c(\mathbf{x}, \mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \lambda \geq -1, \\ w_1(\mathbf{x}, \mathbf{y}), \ldots, w_d(\mathbf{x}, \mathbf{y}) \in \mathbb{R}[\mathbf{x}, \mathbf{y}], h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]. \end{cases} \end{aligned} \tag{12}$$

where $t(\mathbf{x})$ is a SOS polynomial that can be determined with degrees of $h(\mathbf{x})$, $w_1(\mathbf{x},\mathbf{y})$, ..., $w_d(\mathbf{x},\mathbf{y})$, $p_1(\mathbf{x})$, ..., $p_d(\mathbf{x})$, $w_h(\mathbf{x},\mathbf{y})$, $w_c(\mathbf{x},\mathbf{y})$ and $c(\mathbf{x})$. Let

$$k = \max\{\deg(w_i(y_i - p_i))_{i=1,\ldots,d}, \deg(w_h h), \deg(w_c c)\} \tag{13}$$

where $\deg(\cdot)$ is the degree function that returns the degree of the polynomial. Then $t(\mathbf{x},\mathbf{y})$ could be selected as the sum of the squares of all the monomials of degree $\lceil \frac{k}{2} \rceil$, i.e.

$$t(\mathbf{x},\mathbf{y}) = \sum_{|\alpha|+|\beta|=\lceil \frac{k}{2} \rceil} (\mathbf{x}^\alpha \mathbf{y}^\beta)^2 + 1 \tag{14}$$

where $\alpha = (\alpha_1, ..., \alpha_d) \in \mathbb{N}^d$, $\beta = (\beta_1, ..., \beta_d) \in \mathbb{N}^d$, $|\alpha| = \sum_{i=1}^d \alpha_i$, $|\beta| = \sum_{i=1}^d \beta_i$, $\mathbf{x}^\alpha \mathbf{y}^\beta = x_1^{\alpha_1} \cdots x_d^{\alpha_d} y_1^{\beta_1} \cdots y_d^{\beta_d}$. The goal of introducing the polynomial $t(\mathbf{x})$ is to guard the satisfiability of the second constraint in the optimization problem (12), i.e., once the unknown polynomials $h(\mathbf{x})$, $w_1(\mathbf{x},\mathbf{y})$, ..., $w_d(\mathbf{x},\mathbf{y})$, $w_h(\mathbf{x},\mathbf{y})$, $w_c(\mathbf{x},\mathbf{y})$ are obtained, there always exists λ such that the second constraint in the optimization problem (12) holds.

The optimization problem also cannot be solved directly using any SOS or SDP solver. Next, we will utilize an iterative optimization method to approximate the optimal solution of the optimization problem (12). Actually, if the optimal value of the optimization problem (12) is not greater than 0, or a feasible solution makes the objective function not greater than 0, the current $h(\mathbf{x})$ is a solution to the problem (11), i.e., $h(\mathbf{x}) \geq 0$ is an invariant of the polynomial program (4). The following theorem shows this result.

Theorem 3. *Let λ, $h(\mathbf{x})$, $u_1(\mathbf{x})$, ..., $u_m(\mathbf{x})$, $w_1(\mathbf{x},\mathbf{y})$, ..., $w_d(\mathbf{x},\mathbf{y})$, $w_h(\mathbf{x},\mathbf{y})$, $w_c(\mathbf{x},\mathbf{y})$, $v_0(\mathbf{x})$, ..., $v_n(\mathbf{x})$ be a feasible solution of the optimization problem (12), and $\lambda \leq 0$, then $h(\mathbf{x}) \geq 0$ is an invariant of the polynomial program (4).*

3.2 Iterative Solving Method

From Theorem 3, to obtain an invariant of the problem (4), it is sufficient to find a feasible solution to the optimization problem (12) such that the objective function $\lambda \leq 0$ holds. Since the bilinear term $w_h(\mathbf{x},\mathbf{y})h(\mathbf{x})$ makes the optimization problem (12) unsolvable by directly existing solvers, our method fixes one of the two unknown polynomials $h(\mathbf{x})$ and $w_h(\mathbf{x},\mathbf{y})$ in turn to solve the optimization problem (12) iteratively. In one iteration, the approach fixes $h(\mathbf{x})$ first to obtain $w_h(\mathbf{x},\mathbf{y})$. Then, it fixes $w_h(\mathbf{x},\mathbf{y})$ to get $h(\mathbf{x})$ next. This leads to the construction of two constrained optimization problems based on the optimization problem (12) to update $h(\mathbf{x})$ and $w_h(\mathbf{x},\mathbf{y})$. Before then, the initial value $h^{(0)}(\mathbf{x})$ of $h(\mathbf{x})$ could be selected by solving the following constrained system (15).

$$\begin{cases} h(\mathbf{x}) - \sum_{i=1}^{m} u_i(\mathbf{x}) f_i(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}], \\ -h(\mathbf{x}) - \sum_{j=1}^{n} v_j(\mathbf{x}) g_j(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) - \epsilon \in \sum \mathbb{R}[\mathbf{x}]^2, \\ u_1(\mathbf{x}), ..., u_m(\mathbf{x}), v_0(\mathbf{x}), v_1(\mathbf{x}), ..., v_n(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2. \end{cases} \tag{15}$$

This is a classic SOS programming problem that can be efficiently solved with SOS programming tools or SDP tools. For a more suitable initial solution $h^{(0)}(\mathbf{x})$, we will discuss it in section 4.

Let $h^{(T)}(\mathbf{x})$ denote the solution of $h(\mathbf{x})$ after the T-th iteration. Consider the following constrained optimization problems (16).

$$\begin{aligned} \min \quad & \lambda \\ s.t. \quad & \begin{cases} h^{(T)}(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x},\mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x},\mathbf{y}) h^{(T)}(\mathbf{x}) - \\ \qquad w_c(\mathbf{x},\mathbf{y}) c(\mathbf{x}) + \lambda t(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \\ w_h(\mathbf{x},\mathbf{y}), w_c(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \lambda \geq -1, \\ w_1(\mathbf{x},\mathbf{y}), \ldots, w_d(\mathbf{x},\mathbf{y}) \in \mathbb{R}[\mathbf{x},\mathbf{y}]. \end{cases} \end{aligned} \tag{16}$$

The above-constrained optimization problems (16) is obtained by modifying the optimization problem (12). Since $h^{(T)}(\mathbf{x})$ is a known polynomial, constrained optimization problem (16) is a classic SOS programming problem that can be solved efficiently in practice. Let $w_h^{(T)}(\mathbf{x},\mathbf{y})$ be the optimal solution of $w_h(\mathbf{x},\mathbf{y})$ by solving the constrained optimization problems (16).

Setting $w_h(\mathbf{x},\mathbf{y})$ to $w_h^{(T)}(\mathbf{x},\mathbf{y})$ in the optimization problem (12), we can obtain the following constrained optimization problems.

$$\begin{aligned} \min \quad & \lambda \\ s.t. \quad & \begin{cases} h(\mathbf{x}) - \sum_{i=1}^{n} u_i(\mathbf{x}) f_i(\mathbf{x}), u_1(\mathbf{x}), ..., u_m(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x},\mathbf{y})(y_i - p_i(\mathbf{x})) - w_h^{(T)}(\mathbf{x},\mathbf{y}) h(\mathbf{x}) - \\ \qquad w_c(\mathbf{x},\mathbf{y}) c(\mathbf{x}) + \lambda t(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \\ -h(\mathbf{x}) - \sum_{j=1}^{n} v_j(\mathbf{x}) g_j(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) - \epsilon, v_0(\mathbf{x}), ..., v_n(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ w_c(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \lambda \geq -1, \\ w_1(\mathbf{x},\mathbf{y}), \ldots, w_d(\mathbf{x},\mathbf{y}) \in \mathbb{R}[\mathbf{x},\mathbf{y}], h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]. \end{cases} \end{aligned} \tag{17}$$

The above-constrained optimization problem (17) is also a classic SOS programming problem. Let $h^{(T+1)}(\mathbf{x})$ be the optimal solution of $h(\mathbf{x})$ by solving the constrained optimization problems (17).

With a suitable value $h^{(0)}(\mathbf{x})$, let $T = 0, 1, 2, ...$, alternating solving the constrained optimization problems (16) and (17), we could obtain a sequence: $h^{(0)}(\mathbf{x})$, $w_h^{(0)}(\mathbf{x}, \mathbf{y})$, $h^{(1)}(\mathbf{x})$, $w_h^{(1)}(\mathbf{x}, \mathbf{y})$, etc. In fact, following this strategy, theoretically, the value of the objective function, that is, the value of λ, will never increase.

Theorem 4. *Fix the degrees of all unknown polynomials in the constrained optimization problems (16) and (17) and set $t(\mathbf{x}, \mathbf{y})$ as formula (14). Suppose $h^{(0)}(\mathbf{x})$ is a suitable initial solution of $h(\mathbf{x})$ that satisfies the constrained system (15). Let $\lambda_1^{(T)}$ and $\lambda_2^{(T)}$ be the optimal value of the objective function in the constrained optimization problems (16) and (17), respectively. Then,*

$$\lambda_1^{(0)} \geq \lambda_2^{(0)} \geq \lambda_1^{(1)} \geq \lambda_2^{(1)} \geq \cdots \geq \lambda_1^{(T)} \geq \lambda_2^{(T)} \geq \lambda_1^{(T+1)} \geq \lambda_2^{(T+1)} \geq \cdots \quad (18)$$

From Theorem 4, we have that by setting a suitable initial solution $h^{(0)}(\mathbf{x})$ of $h(\mathbf{x})$, then alternating solving the constrained optimization problems (16) and (17) for $T = 0, 1, 2, ...$, the optimal values will never increase. Actually, an invariant can be obtained once an optimal value $\lambda_1^{(T)} \leq 0$ or $\lambda_2^{(T)} \leq 0$ holds.

Theorem 5. *Under the conditions of Theorem 4, if $\lambda_1^{(T)} \leq 0$ or $\lambda_2^{(T)} \leq 0$, then the current value of $h(\mathbf{x})$ makes $h(\mathbf{x}) \geq 0$ an invariant of the polynomial program(4).*

In this part, we propose an iterative optimization method to approximate the optimal solution of the optimization problem (12). The optimization problem (12) is not a convex problem; multiple locally optimal solutions can exist. Given this characteristic, the selection of an appropriate initial solution becomes crucial in achieving the desired final result. Next, we discuss obtaining a suitable initial solution in Section 4.

4 A Suitable Initial Value

Initial value selection plays a key role in solving problem (12). Given that the optimization problem (12) is not convex, it is difficult to guarantee global optimal convergence. The solution of nonconvex problems based on numerical methods is closely correlated with the initial value selection. Thus, finding a suitable initial value is vital for leading a better convergence path to the optimal value of problem (12). In the rest of this section, we propose a strategy to identify a suitable initial value of $h^{(0)}(\mathbf{x})$ based on Craig interpolant.

Recap: in this paper, our goal is to find an invariant for the polynomial loop (4), i.e., find a feasible solution to the optimization problem (12) such that the objective function $\lambda \leq 0$. With this, it is sufficient to find a polynomial $h^{\star}(\mathbf{x})$

such that $h^\star(\mathbf{x}) \geq 0$ is an invariant for the polynomial program (4). Our strategy is as follows. 1. Select a suitable initial value $h^{(0)}(\mathbf{x})$; 2. solve the optimization problems (16) and (17) iteratively, as stated above in Section 3.

In order to select a suitable initial value $h^{(0)}(\mathbf{x})$, we introduce the concepts of the forward reachable set A_i and the unsafe set B_0 for the polynomial program (4) as follows.

Definition 6. *For the polynomial program (4) define A_i and B_0 as follows*

$$\begin{aligned} A_0 &= \{\mathbf{x} \mid f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0\}, \\ B_0 &= \{\mathbf{x} \mid g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0 \wedge -c(\mathbf{x}) > 0\}, \\ A_i &= \{\mathbf{x} \mid \mathbf{x} = f(\mathbf{y}) \wedge \mathbf{y} \in A_{i-1} \wedge c(\mathbf{y}) \geq 0\}, i = 1, 2, ... \end{aligned} \tag{19}$$

A_i is called the i-th step forward reachable set from A_0,

From the above definition, A_0 is the valid input set in the polynomial loop (4); A_i is the set of states that the while loop executes exactly i times from the initial set A_0. Naturally, B_0 could be treated as an unsafe set. Since once the set B_0 is reachable, then the polynomial loop (4) does not hold, i.e., there exists an input state that satisfies the precondition Pre and makes the while loop terminate within finite steps, and the output does not satisfy the postcondition $Post$; Therefore, if the two sets $\bigcup_{i=0}^{\infty} A_i$ and B_0 are disjoint, or equivalently, there exists a separate function that separates the two sets $\bigcup_{i=0}^{\infty} A_i$ and B_0, then the polynomial loop (4) holds.

Suppose $h(\mathbf{x}) \geq 0$ is an invariant of the polynomial loop (4). In that case, $h(\mathbf{x})$ is a separate function that can separate the two sets $\bigcup_{i=0}^{\infty} A_i$ and B_0. Let $h_a(\mathbf{x})$ be a separate function that can separate the two sets $\bigcup_{i=0}^{a} A_i$ and B_0, where $a \in \mathbb{N}$. Intuitively, the larger the number a, the closer the $h_a(\mathbf{x})$ and an exact invariant is. Based on this, we propose a heuristic strategy to obtain a suitable initial value, $h^{(0)}(\mathbf{x})$, by finding a separate function of $\bigcup_{i=0}^{a} A_i$ and B_0 with the number a increasing.

Remark 1. Suppose that we define $B_i = \{\mathbf{x} \mid \mathbf{y} = f(\mathbf{x}) \wedge \mathbf{y} \in B_{i-1} \wedge Cond(\mathbf{x})\}$ as the i-th step backward reachable set from B_0. Intuitively, selecting the separate function of $\bigcup_{i=0}^{a} A_i$ and $\bigcup_{j=0}^{b} B_j$ is better than the separate function of $\bigcup_{i=0}^{a} A_i$ and B_0. However, the existence of a polynomial separate function cannot be guaranteed when $b \geq 1$, even if both A_0 and B_0 are bounded sets. For the case of $b = 0$, when A_0 and B_0 are bounded sets, we can propose a method to obtain a polynomial separate function of $\bigcup_{i=0}^{a} A_i$ and B_0 based on Craig interpolant.

In the rest of this section, we transform the problem of finding a polynomial separate function of $\bigcup_{i=0}^{a} A_i$ and B_0 to a Craig interpolant generation problem, which can be solved efficiently using SOS programming or semidefinite programming when A_0 and B_0 are both bounded sets.

Let ϕ_{A_i} be a formula defined below that describes the set A_i, i.e., $A_i = \{\mathbf{x} \mid \phi_{A_i}(\mathbf{x})\}$.

$$\begin{aligned} \phi_{A_0} &: f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0, \\ \phi_{A_i} &: \mathbf{x} = \mathbf{p}(\mathbf{y}) \wedge \phi_{A_{i-1}}(\mathbf{y}) \wedge Cond(\mathbf{y}), \end{aligned} \tag{20}$$

where $i = 1, 2, ...$ Next, we prove that for any $i \geq 1$ the formula ϕ_{A_i} is Archimedean if ϕ_{A_0} is Archimedean. The following three lemmas are in preparation for the general conclusion in Theorem 6. Then our conclusion in Theorem 7 can be derived from Theorem 6.

Lemma 1. *Let $p(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ be an any given polynomial, there exist $a > 0$ and $k \in \mathbb{N}$ such that $a(k + x_1^{2k} + ... + x_d^{2k}) - p(\mathbf{x})$ is a SOS polynomial.*

Lemma 2. *Let $\mathcal{M} \subset \mathbb{R}[\mathbf{x}]$ be a quadratic module, $f \in \mathbb{R}[\mathbf{x}]$ satisfies that $f \in \mathcal{M}$ and $-f \in \mathcal{M}$, then for any $p \in \mathbb{R}[\mathbf{x}]$, $pf \in \mathcal{M}$ holds.*

Lemma 3. *Let $\mathcal{M} \subset \mathbb{R}[\mathbf{x}]$ be an Archimedean quadratic module, $\mathbf{x} = (x_1, ..., x_d)$. For any given $k \in \mathbb{N}$, there exists $a > 0$ such that $a - (x_1^{2k} + ... + x_d^{2k}) \in \mathcal{M}$.*

Theorem 6. *Suppose $\mathcal{M}_{\mathbf{x}}(f_1, ..., f_m) \subset \mathbb{R}[\mathbf{x}]$ is an Archimedean quadratic module, $\mathbf{x} = (x_1, ..., x_d)$, $p(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ is a given polynomial. Let $f_{m+1} = x_{r+1} - p(\mathbf{x})$, then the quadratic module $\mathcal{M}_{\mathbf{x},x_{d+1}}(f_1, ..., f_m, f_{m+1}, -f_{m+1}) \subset \mathbb{R}[\mathbf{x}, x_{d+1}]$ is also an Archimedean quadratic module.*

Theorem 7. *Consider the polynomial loop as (4). Let ϕ_{A_i} $(i = 0, 1, 2, ...)$ be defined as (20). Suppose ϕ_{A_0} is Archimedean, then for any $i \in \mathbb{N}^+$, ϕ_{A_i} is also Archimedean.*

From Theorem 7, if ϕ_{A_0} is Archimedean, then for any $i \geq 1$ the formula ϕ_{A_i} is Archimedean. Let $\psi_{B_0} : g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0 \wedge -c(\mathbf{x}) > 0$. Therefore, finding a separate function $h(\mathbf{x})$ of $\bigcup_{i=0}^{a} A_i$ and B_0 is equivalent to finding a polynomial $h(\mathbf{x})$ such that $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x}) \models h(\mathbf{x}) > 0$ and $\psi_{B_0}(\mathbf{x}) \models -h(\mathbf{x}) > 0$. Actually, when $a \geq 1$, there exist auxiliary variables in $\bigvee_{i=0}^{a} \phi_{A_0}(\mathbf{x})$ except $\mathbf{x}$, and $\mathbf{x}$ is the common variable of formulae $\bigvee_{i=0}^{a} \phi_{A_0}(\mathbf{x})$ and $\psi_{B_0}(\mathbf{x})$. Thus, $h(\mathbf{x})$ is a Craig interpolant of $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x})$ and $\psi_{B_0}(\mathbf{x})$. Since $\phi_{A_0}(\mathbf{x})$, ..., $\phi_{A_a}(\mathbf{x})$, and $\psi_{B_0}(\mathbf{x})$ are all Archimedean, then $h(\mathbf{x})$ can be obtained by using the Craig interpolant method in [18]. For example, suppose $a = 1$,

$$\phi_{A_1}(\mathbf{x}) : x_1 - p_1(\mathbf{y}) = 0, ..., x_d - p_d(\mathbf{y}) = 0, c(\mathbf{y}) \geq 0, f_1(\mathbf{y}) \geq 0, ..., f_m(\mathbf{y}) \geq 0.$$

$h(\mathbf{x})$ should be a Craig interpolant of $\phi_{A_0}(\mathbf{x}) \vee \phi_{A_1}(\mathbf{x})$ and $\psi_{B_0}(\mathbf{x})$, using the interpolant generation method in [18], $h(\mathbf{x})$ can be obtained by solving the following SOS programming.

$$\begin{cases} h(\mathbf{x}) - 1 - \sum_{i=1}^{m} u_i(\mathbf{x}) f_i(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ h(\mathbf{x}) - 1 - \sum_{i=1}^{d} q_i(\mathbf{x}, \mathbf{y})(x_i - p_i(\mathbf{y})) - u_0'(\mathbf{x}, \mathbf{y}) c(\mathbf{y}) - \sum_{i=1}^{m} u_i'(\mathbf{x}, \mathbf{y}) f_i(\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \\ -h(\mathbf{x}) - 1 - \sum_{j=1}^{n} v_j(\mathbf{x}) g_j(\mathbf{x}) + v_0(\mathbf{x}) c(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2, \\ u_1(\mathbf{x}), ..., u_m(\mathbf{x}), u_0'(\mathbf{x}, \mathbf{y}), ..., u_m'(\mathbf{x}, \mathbf{y}), v_0(\mathbf{x}), ..., v_n(\mathbf{x}), w_1(\mathbf{x}), w_2(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, \\ h(\mathbf{x}), q_1(\mathbf{x}, \mathbf{y}), ..., q_r(\mathbf{x}, \mathbf{y}) \in \mathbb{R}[\mathbf{x}, \mathbf{y}]. \end{cases}$$

This is a classic interpolant generation problem, as shown in [18], which can be efficiently solved using SOS solvers [43] or SDP solvers [36].

5 Algorithm

In this section, we propose an algorithm based on the above sections 3 and 4 in Algorithm 1, then prove the correctness, termination and semicompleteness.

Theorem 8 (Correctness). *$h(\mathbf{x}) \geq 0$ is an invariant of program loop (4) if Algorithm 1 returns a formula $h(\mathbf{x}) \geq 0$; the program loop (4) does not hold if Algorithm 1 returns "Error".*

Theorem 9 (Termination). *Algorithm 1 terminates in finite steps if the formulae $Pre(\mathbf{x})$ and $\neg Post(\mathbf{x}) \wedge \neg Cond(\mathbf{x})$ in program loop (4) are Archimedean.*

Algorithm 1 returns "$\mathcal{I}$" when the formula $\mathcal{I}$ is an invariant proving the Hoare triple C holds, "ERROR" when the Hoare triple (4) is proven not to hold, or "UNKNOWN" when the Hoare triple (4) cannot be proven to hold or not by the current algorithm. The "UNKNOWN" result occurs due to insufficient polynomial degree or suboptimal initial values, which can be addressed by increasing k and a.

The completeness of our method is difficult to achieve. Specifically, Theorem 2 provides a sufficient condition for $h \geq 0$ to be an invariant. In order to obtain a semicomplete result, we need to add some assumptions about the invariants and the program loop in formula (4). We present the following theorem as a semicomplete result.

Theorem 10 (Semicompleteness). *Let $h(\mathbf{x}) > 0$ be an invariant of the program loop (4). Suppose the following two assumptions hold: 1. $h(\mathbf{x})$ satisfies stronger invariant conditions than those in Definition 1, as follows:*

1'). $Pre(\mathbf{x}) \models h(\mathbf{x}) > 0$,
2'). $h(\mathbf{x}) \geq 0 \wedge Cond(\mathbf{x}) \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) \models h(\mathbf{y}) > 0$,
3'). $h(\mathbf{x}) \geq 0 \wedge \neg Cond(\mathbf{x}) \models Post(\mathbf{x})$;

2. during the execution of the program loop (4), the values of the state variables are bounded by a known ball, that is, there exists a known $N > 0$, s.t., $N - \sum_{i=1}^{d} x_i^2 \geq 0$. If the formulae $Pre(\mathbf{x})$ and $\neg Post(\mathbf{x}) \wedge \neg Cond(\mathbf{x})$ in program loop (4) are Archimedean, then the invariant $h(\mathbf{x}) > 0$ corresponds to a solution of the SOS constraint system (11) with the second constraint modified as follows:

$$\begin{aligned} &h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x},\mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x},\mathbf{y})h(\mathbf{x}) - w_c(\mathbf{x},\mathbf{y})c(\mathbf{x}) \\ &-w_b(\mathbf{x},\mathbf{y})(N^2 - \sum_{i=1}^{d} x_i^2) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, w_b(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2 \end{aligned} \tag{21}$$

There are two assumptions in Theorem 10 above. The first pertains to the invariant $h(\mathbf{x})$, and the second concerns the program loop (4). In this paper, we use a numerical method to derive invariants. Since an error in the calculations is inevitable in the numerical methods, it is unnecessary and unrealistic to

Algorithm 1 Invariant Generation for Polynomial Loop

Require: Polynomial Loops as (4): $\{Pre\}$ while $Cond$ do $Body$ $\{Post\}$, and degree parameter k

Ensure: ($\mathcal{I}$/Error/Unknown), where $\mathcal{I}$ is an invariant with degree parameter k; Error means the polynomial loop doesn't hold; Unknown means we can't find an invariant with degree parameter k.

1: $\phi_{A_0}(\mathbf{x}) \leftarrow Pre$; $\psi_{B_0}(\mathbf{x}) \leftarrow (\neg Cond \wedge \neg Post)$;
2: **if** $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x}) \wedge \psi_{B_0}(\mathbf{x}) \models \bot$ **then**
3: Calculate a Craig interpolant $\hat{h} > 0$ of $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x})$ and $\psi_{B_0}(\mathbf{x})$;
/*From [18], the Craig interpolant can be obtained by using semidefinite programming if $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x})$ and $\psi_{B_0}(\mathbf{x})$ are Archimedean*/.
4: **else**
5: **return** Error;
6: **end if**
7: Setting the degrees of unknown polynomials in optimization problems (16) and (17) w.r.t. the degree parameter k;
8: **while** true **do**
9: Treating $\hat{h}$ as $h^{(T)}$, and solving optimization problems (16), then obtain the solution of λ and $w_h(\mathbf{x}, \mathbf{y})$ as $\hat{\lambda}_1$ and $\hat{w}_h$;
10: **if** $\hat{\lambda}_1 \leq 0$ **then**
11: **return** $\hat{h}(\mathbf{x}) \geq 0$;
12: **end if**
13: Treating $\hat{w}_h$ as $w_h^{(T)}$, and solving optimization problems (17), then obtain the solution of λ and $h(\mathbf{x})$ as $\hat{\lambda}_2$ and $\hat{h}$;
14: **if** $\hat{\lambda}_2 \leq 0$ **then**
15: **return** $\hat{h}(\mathbf{x}) \geq 0$;
16: **end if**
17: **if** $|\hat{\lambda}_1 - \hat{\lambda}_2| \leq \sigma$ **then**
18: **return** Unknown;
19: **end if**
20: **end while**

distinguish between $\geq$ and $>$ for for polynomials with floating-point coefficients. Thus, if an invariant $\hat{h}(\mathbf{x})$ is derived using a numerical method, then it should be a polynomial with floating point coefficients so that it is unnecessary and unrealistic to distinguish between $\geq$ and $>$ for $h(\mathbf{x})$. Therefore, we make the first assumption in Theorem 10. Since in many real-world programs, the values of the parameter are bounded with a known number, then we make the second assumption in Theorem 10 which is used for Archimedean condition. From some perspectives, these two assumptions in Theorem 10 are reasonable. Similar assumptions have been made in many related invariant synthesis works based on numerical methods [2,6,18,63].

Remark 2. Essentially, the semicompleteness of the methods of [6,21,63] and our method is based on the two assumptions in Theorem 10 and Putinar's Positivstellensatz, i.e. Theorem 1. In order to obtain semicompleteness, we change the second constraint of the SOS constraint system (11) in Theorem 10.

But in many real-world programs, the trivial upper bound N for the parameters can be extremely large during execution. For example, the upper bound for 64-bit floating-point numbers is approximately 1.798×10^{308}. If such a bound is directly introduced into our calculations, the results would likely be impractical or unusable. Therefore, we do not use the boundedness of the program parameters in our actual method. However, if the upper bound N is known with an acceptable size, our method can easily be changed accordingly, like changing the second constraint of the SOS constraint system (11) in Theorem 10.

Complexity Analysis. The goal of Algorithm 1 is actually to solve the single-objective optimization problem (12). The number of parameters in problem (12) is about $O(d^k)$ when setting the degree parameter k. Hence, Algorithm 1 is essentially solving for these $O(d^k)$ parameters. In Algorithm 1, the most computationally expensive process consists of the following two parts, namely (1) calculating a Craig interpolant in line 3, and (2) iterative solving procedure in lines 8-20. Calculating a Craig interpolant requires solving an SDP problem, the time complexity is a polynomial function w.r.t. its parameters, i.e. $O(g(d^k))$, where g is a polynomial function. The iterative solving procedure in lines 8-20 is a while loop, two SDP problems need to be solved in each iteration, and the time complexity is about $O(\tau g(d^k))$, where τ refers to the number of iterations of the while loop. Thus, the time complexity is a polynomial function w.r.t. d and an exponential function w.r.t. k.

6 Experimental Results

Setup. We use `MATLAB(R2023b)` as our experimental platform and develop the algorithm in this paper, utilizing `YALMIP` [36] with `Mosek` [40] as SOS solvers. During the validation phase, we use symbolic solvers `Z3` [15] and `Mathematica` [26] to perform the posterior validation. We conducted all the experiments on a desktop computer with an i7 13700KF processor, 32GB of RAM. All codes and benchmarks can be obtained via: `https://github.com/NonlinearCraig/CraigInv/`.

Benchmarks. We propose benchmarks common in program verification with nonlinear constraints and real variables that originate from different domains, including dynamic systems, circuit design, and physics. All benchmarks are cited from other papers, except sgd. sgd is modified from a stochastic gradient descent problem. This kind of problem is prevalent in modern neural networks. In some peers' work, they use dataset [49] to test their approaches. To compare with their work, we also conduct experiments on this dataset.

Comparision. For comparison, we performed experiments with the state-of-the-art methods PolySynth [21], LaM4Inv [62], invSDP [63] and BCDC [61]. The primary tool we want to compare to is PolySynth. PolySynth [21] is an elaborately designed efficient tool for invariant generation and program synthesis. Another essential tool we use is LaM4Inv [62], which is a new neuro-symbolic loop invariant inference algorithm based on LLM. For invSDP [63], similar to our

Table 1. Experimental results with benchmarks 1.

Benchmarks	Iter	Ours	Z3	PolySynth	LaM4Inv	BCDC	invSDP	Prajna
car[13]	0	**0.1s**	1.7s	>600s	76.6s	Failed	3.0s	Failed
discrete[63]	0	**0.2s**	>600s	>600s	135.1s	3.8s	Failed	Failed
logistic[31]	0	**0.1s**	2.6s	>600s	393.7s	Failed	Failed	Failed
sgd	5	1.5s	**0.1s**	>600s	Failed	Failed	Failed	Failed
sgd 0.001	8	1.7s	**0.1s**	>600s	>600s	Failed	Failed	Failed
cav13-1[13]	1	**0.4s**	>600s	>600s	Failed	9.7s	Failed	Failed
cav13-2 [13]	2	**0.7s**	>600s	>600s	>600s	7.5s	Failed	Failed
unicycle [58]	1	**0.5s**	>600s	0.97s	Failed	Failed	Failed	Failed
circuit [37]	0	**0.2s**	>600s	>600s	Failed	9.8s	Failed	Failed
deter[41]	1	**0.5s**	>600s	>600s	Failed	Failed	Failed	20.6s
bound [2]	1	**0.5s**	>600s	>600s	>600s	14.7s	Failed	Failed
bound2 [2]	1	**0.5s**	>600s	>600s	116.5s	30.6s	Failed	Failed
contrived [61]	1	**0.5s**	>600s	>600s	291.8s	11.1s	Failed	Failed
transcend [46]	3	1.4s	>600s	>600s	>600s	Failed	Failed	**1.1s**
basin [55]	6	**2.5s**	>600s	>600s	Failed	Failed	Failed	Failed
lyapunov [45]	3	**2.9s**	>600s	>600s	>600s	400.1s	Failed	Failed
motivate [52]	1	**1.3s**	>600s	>600s	>600s	Failed	Failed	7.5s

"Iter": The number of iterations to get an invariant. "Ours": Algorithm 1.

method, it also uses semidefinite programming to generate polynomial invariants. Besides, we have conducted experiments with BCDC [61] on benchmark 1. BCDC is an excellent synthetic algorithm for BMI problems based on difference-of-convex programming. A detailed introduction and discussion of these methods will be made in Section 7. We also tested the effects of different polynomial templates and the benefits of taking Craig interpolant as initial in the Appendix.

Numerical Errors. To mitigate numerical errors, we have taken several tricks. One is to perform a posterior verification using symbolic solvers such as Z3 [15]. Another is introducing a suitable error bound ϵ, which appears in constraints (15) and (17). Generally speaking, a large ϵ can strengthen the constraints, guaranteeing the correctness of the solutions. However, it may also affect the completeness of solutions when ϵ is too large [63]. In our approach, ϵ is set within the range of 10^{-8} to 10^{-6}. In addition, we also use a rounding-off strategy. In general cases, numerical perturbations of the coefficients will have a relatively large impact on candidate invariants when their coefficients are small, making the solutions very unstable. In our experiments, this phenomenon occurs frequently. Thus, we discard candidate invariants when their coefficients are all very small, even if $\lambda < 0$. Specifically, we reject the solution and continue to iterate when all absolute values of coefficients are less than 0.1. These tricks will make our algorithm easier to get a real solution.

Table 2. Experimental results with benchmarks 2.

Benchmarks	Iter	Ours	Z3	PolySynth	LaM4Inv	invSDP	Prajna
z3sqrt	0	**0.5s**	Failed	>600s	Failed	Failed	Failed
cohendiv	1	2.9s	>600s	**1.1s**	23.1s	Failed	Failed
berkeley	1	0.8s	**0.4s**	10.3s	Failed	Failed	0.7s
euclidex2	0	Failed	>600s	>600s	279.4s	Failed	Failed
cohencu	0	**0.3s**	>600s	>600s	131.9s	Failed	4.2s
fermat2	0	0.4s	**0.3s**	0.5s	71.7s	Failed	Failed
firefly	1	1.0s	**0.5s**	>600s	Failed	Failed	0.9s
mannadiv	0	**0.3s**	**0.3s**	0.5s	Failed	Failed	Failed
freire1	0	**0.3s**	>600s	>600s	Failed	Failed	0.5s
freire2	3	**2.4s**	>600s	25.4s	Failed	Failed	8.3s
illinois	1	1.2s	>600s	14.7s	Failed	Failed	**1.0s**
lcm	1	1.8s	**0.4s**	0.9s	77.2s	Failed	Failed
wensely	0	Failed	>600s	>600s	Failed	Failed	Failed
mesi	9	12.0s	**0.9s**	8.3s	15.9s	Failed	Failed
moesi	5	**10.3s**	**0.6s**	10.5s	15.7s	Failed	Failed
petter	0	**0.3s**	>600s	0.6s	Failed	Failed	3.0s
readerwriter	1	**1.7s**	>600s	8.8s	28.1s	Failed	Failed
ex_sqrt	0	**0.4s**	>600s	>600s	Failed	Failed	Failed

Measure. Tables 1 and 2 measure the running time in seconds, with a time limitation of 600 seconds. The bold font denotes the best result w.r.t. the running time. "Failed" means the candidate invariant is not a true invariant, or solvers can't get any solution.

Results on benchmark 1. Experimental results demonstrate the effectiveness and efficiency of our approach in practice. In Table 1, our approach successfully solves all the benchmarks, significantly exceeding others in time consumption and solving capacity. There are a few cases with one variable where directly using Z3 consumes less time than ours. This is mainly because the numerical solvers we rely on are susceptible to numerical errors and sometimes get stuck near local optimal solutions. Excluding our algorithm, BCDC performs the best. This may be due to some of the benchmarks involving dynamic systems, and BCDC is a state-of-the-art level tool for solving such problems. The experimental results also show that the heuristics of Z3 and PolySynth do not perform well under complex nonlinear conditions.

Results on benchmark 2. In benchmark 2, our method successfully solves most of the problems and gets the best results in half of the data set. PolySynth and Z3 perform much better in benchmark 2 than in benchmark 1, showing the effects of their heuristics under linear conditions. However, our algorithm still outperforms their methods in terms of effectiveness. As for invSDP, it fails in

nearly all benchmarks in Tables 1 and Table 2. This may be due to invSDP lack of necessary effective means of eliminating numerical errors. It should be noted that the results in [21] are somewhat different from ours. This is because we dig "holes" at invariant positions while [21] digs "holes" at loop bodies for some benchmarks. Besides, we eliminate the results of BCDC because their method fails in multibranch loops.

7 Related Work

As a key technique in program verification, invariant generation has attracted a lot of attention in the past several decades, and a large amount of work has been proposed. In this section, we compare our method with some very related works and present a concise introduction to some other different works.

The significance and inherent difficulty have made loop invariant generation a long-standing problem. Finding the strongest invariants for polynomial loops is hard and remains wide open [41]. Therefore, most works about loop invariants are devoted to finding a valid invariant that can prove the correctness of the Hoare triple. In recent years, various techniques have been introduced in invariant generation, including constraint solving [2,6,9,21,27,28,34,63], logical inference [4,16,19,54,12], counterexample guided [7,8,20], Craig interpolation [18,33,39], abstract interpretation [10,22,48,50], recurrence analysis [1,17,25,29,30,59,60], and machine learning [3,23,51,56,62]. Among the methods, with the continuous advancement of constraint-solving tools, constraint-solving-based approaches have shown increasing relevance and promise. In addition, methods based on counterexample-guided, Craig interpolation, and machine learning can also be classified within this scenario. Nonetheless, due to the undecidability of program analysis in general [47], no universal automated method today guarantees the generation of loop invariants for all possible programs.

For linear invariant synthesis, a comprehensive and classical approach based on Farkas' lemma has been proposed, providing a complete solution [9]. Nevertheless, when dealing with nonlinear constraints, quantifier elimination is a standard solution, but this approach is inefficient in practice since it suffers from a double exponential time complexity [14,28]. In order to find more efficient methods, many researchers have turned to heuristic approaches when solving nonlinear constraints [35,53]. A few studies investigate how to generate polynomial invariants with polynomial loop programs efficiently [2,6,11,21,34,63]. They are all faced with solving a nonconvex problem produced by invariant constraints. In order to obtain an efficient invariant synthesis algorithm, they either approximate invariant constraints by constraint relaxation or transform the constraints into easier-to-solve ones. Our work falls into this category. We then compare our work with some very related works as follows.

Compared with [6,21]: [6,21] employ SOS relaxation to transform the invariant conditions into a SOS constraint system, a quadratic constraint system can then be obtained after setting the templates of all unknown polynomials. Next, they utilize quadratic programming solvers to handle the constraint

system. In contrast, we transform the SOS constraint system into a single-objective optimization problem and propose an alternate iterative method to handle this optimization problem. They claim that their method PolySynth is semicomplete, and the complexity is subexponential. The semicompleteness of their method comes from the fact that any invariant corresponds to a solution of the quadratic constraint system under the compactness condition, also the Archimedean condition. Our method also has a similar semicompleteness. The complexity of their method is actually a subexponential function with respect to the number of parameters, while the number of parameters is about $O(d^k)$ when setting the templates with d variables and the total degree k. In other words, their complexity is a subexponential function with respect to $O(d^k)$ arguments. For comparison, the complexity of our method is a polynomial function with respect to $O(d^k)$ arguments, with a product factor the number of iterations τ. The experiments in Section 6 show that the value of τ is generally small.

Compared with [63]: [63] also employs SOS relaxation to transform invariant conditions into a SOS constraint system. Next, they set the templates of all unknown polynomials, an under-approximation of the valid set of the unknown polynomials can be obtained by constructing high-level SOS relaxations. Finally, they used SMT solvers to extract a valid solution from the under-approximation. The main contribution of their work is theoretical. With mild restrictions on the assumptions of templates, the method in [63], namely invSDP, is sound, convergent, and weak-complete. However, the process of constructing high-level SOS relaxations brings high complexity, and the under-approximation of the valid set derives usually cannot get a correct solution due to numerical errors, i.e. the numerical stability needs to be improved. In contrast, our goal is to obtain a valid invariant. And we solve the SOS constraint system directly based on an alternate iterative strategy. We fully consider that the SOS constraint system is a nonconvex constraint system, and the numerical method is very dependent on the selection of initial values, so we propose an initial value selection method based on Craig interpolant.

Compared with barrier certificate works [61,64]: Barrier certificate is a concept in hybrid system verification, which is of great significance in reachability analysis, safety verification, and control synthesis of hybrid system, and is often called invariant of continuous system. To obtain a barrier certificate, the state-of-the-art methods [61,64] first employ SOS relaxation to translate the barrier certificate conditions into a bilinear SOS constraint system. The methods in [61,64] and ours are all needed to solve a bilinear SOS constraint system. [61,64] further transform them into bilinear matrix inequalities (BMI). [64] proposes an iteration method to solve the BMI for a numerical invariant, and then obtain an exact polynomial invariant via Newton refinement and rational recovery techniques. [61] proposes a method based on difference-of-convex programming, which approaches a local optimum by solving a series of convex optimization problems. In our method, we further transform the bilinear SOS constraint system into an optimization problem and then solve it based on the iterative method. At the same time, we propose a suitable initial value based on Craig

interpolation, which makes it easier for our method to obtain an effective solution and ensure the feasibility of the solution during the solving process.

8 Conclusion and Further Work

This paper proposes a novel invariant generation method for polynomial loops. We first introduce a penalty term in the constraints, obtain a single-objective optimization problem, and subsequently solve the problem iteratively in its feasible solution region. The iterative method depends on the initial value, so we propose a heuristic approach to obtain a suitable initial value based on the nonlinear Craig interpolation technique. In the end, we prove the correctness and termination of the method under Archimedean's condition. Our approach for finding a suitable initial value is a heuristic approach with a qualitative conclusion rather than a quantitative one. What initial value can ensure the convergence to a desired solution is worth considering further in the future.

Actually, the correctness of our approach is independent of the Archimedean condition. So, can we calculate a Craig interpolant and deal with termination without the Archimedean condition? An intuitive idea is to homogenize the polynomials, which allows us to constrain the variables on the unit sphere, and the Archimedean condition is naturally satisfied. We aim to develop this idea further and provide complete proof in future research.

References

1. Amrollahi, D., Bartocci, E., Kenison, G., Kovács, L., Moosbrugger, M., Stankovic, M.: Solving invariant generation for unsolvable loops. In: SAS (2022). https://doi.org/10.1007/978-3-031-22308-2_3
2. AssaléAdjé, Garoche, P.L., Magron, V.: Property-based polynomial invariant generation using sums-of-squares optimization. In: SAS (2015). https://doi.org/10.1007/978-3-662-48288-9_14
3. Bao, J., Pathak, D., Hsu, J., Roy, S.: Data-driven invariant learning for probabilistic programs. In: CAV (2022). https://doi.org/10.1007/978-3-319-08867-9_6
4. Batz, K., Chen, M., Junges, S., Kaminski, B.L., Katoen, J.P., Matheja, C.: Probabilistic program verification via inductive synthesis of inductive invariants. In: TACAS (2023). https://doi.org/10.1007/978-3-031-30820-8_25
5. Bochnak, J., Coste, M., Roy, M.: Real Algebraic Geometry. SpringerScience & Business Media (1998). https://doi.org/10.1007/978-3-662-03718-8
6. Chatterjee, K., Fu, H., Goharshady, A.K., Goharshady, E.K.: Polynomial invariant generation for non-deterministic recursive programs. In: PLDI (2020). https://doi.org/10.1145/3385412.3385969
7. Chen, Y.F., Hong, C.D., Wang, B.Y., Zhang, L.: Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In: CAV (2015). https://doi.org/10.1007/978-3-319-21690-4_44
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643

9. Colón, M., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: CAV (2003). https://doi.org/10.1007/978-3-540-45069-6_39
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977). https://doi.org/10.1145/512950.512973
11. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: VMCAI (2005). https://doi.org/10.1007/978-3-540-30579-8_1
12. Cyphert, J., Kincaid, Z.: Solvable polynomial ideals: The ideal reflection for program analysis. In: POPL (2024). https://doi.org/10.1145/3632867
13. Dai, L., Xia, B., Zhan, N.: Generating non-linear interpolants by semidefinite programming. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_25
14. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. Journal of Symbolic Computation **5**(1-2), 29–35 (1988). https://doi.org/10.1016/S0747-7171(88)80004-X
15. De Moura, L., Bjorner: Z3: An efficient SMT solver tools, and algorithms for the construction and analysis of systems. In: CAV (2008). https://doi.org/https://doi.org/10.1007/978-3-540-78800-3_24
16. Dillig, I., Dillig, T., Li, B., Mcmillan, K.: Inductive invariant generation via abductive inference. In: OOPSLA (2013). https://doi.org/10.1145/2509136.2509511
17. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542253
18. Gan, T., Xia, B., Xue, B., Zhan, N., Dai, L.: Nonlinear Craig interpolant generation. In: CAV (2020). https://doi.org/10.1007/978-3-031-71162-6_5
19. Garg, P., Lding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV (2014). https://doi.org/10.1007/978-3-319-08867-9_5
20. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL (2016). https://doi.org/10.1145/2837614.2837664
21. Goharshady, A.K., Hitarth, S., Mohammadi, F., Motwani, H.J.: Algebro-geometric algorithms for template-based synthesis of polynomial programs. In: PACM PL (2023). https://doi.org/10.1145/3586052
22. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: VMCAI (2009). https://doi.org/10.1007/978-3-319-08867-9_5
23. He, J., Singh, G., Püschel, M., Vechev, M.: Learning fast and precise numerical analysis. In: PLDI (2020). https://doi.org/10.1145/3385412.3386016
24. Hoare, C., A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/3632863
25. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant generation for multi-path loops with polynomial assignments. In: VMCAI (2018). https://doi.org/10.1007/978-3-319-73721-8_11
26. Inc., W.R.: Mathematica, Version 13.2, `https://www.wolfram.com/mathematica`, champaign, IL, 2022
27. Ji, Y., Fu, H., Fang, B., Chen, H.: Affine loop invariant generation via matrix algebra. In: CAV (2022). https://doi.org/10.1145/3385412.3385986
28. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Deduction and Applications (2005). `http://drops.dagstuhl.de/opus/volltexte/2006/511`

29. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. In: PLDI (2017). https://doi.org/10.1145/3062341.3062373
30. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. In: POPL (2018). https://doi.org/10.1145/3158142
31. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: FORMATS (2011). https://doi.org/10.1007/978-3-642-24310-3_17
32. Laurent, M.: Sums of squares, moment matrices and optimization over polynomials. Emerging Applications of Algebraic Geometry pp. 157–270 (2009). https://doi.org/10.1007/978-0-387-09686-5_7
33. Lin, S.W., Sun, J., Xiao, H., Liu, Y., Sanán, D., Hansen, H.: Fib: Squeezing loop invariants by interpolation between forward/backward predicate transformers. In: ASE (2017). https://doi.org/10.1109/ASE.2017.8115690
34. Lin, W., Wu, M., Yang, Z., Zeng, Z.: Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. Frontiers of Computer Science **8**, 192–202 (2014). https://doi.org/10.1007/S11704-014-3150-6
35. Liu, H., Fu, H., Yu, Z., Song, J., Li, G.: Scalable linear invariant generation with farkas' lemma. In: PACM PL (2022). https://doi.org/10.1145/3563295
36. Löfberg, J.: Pre- and post-processing sum-of-squares programs in practice. J. of IEEE Transactions on Automatic Control **54**(5), 1007–1011 (2009). https://doi.org/10.1109/TAC.2009.2017144
37. Mahathi Anand, Vishnu Murali, A.T., Zamani, M.: Safety verification of dynamical systems via k-inductive barrier certificates. In: CDC (2021). https://doi.org/10.1109/CDC45484.2021.9682889
38. Marshall, M.: Positive Polynomials and Sums of Squares. American Mathematical Society (2008)
39. Mcmillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_31
40. Mosek, A.: Mosek optimization toolbox for matlab. User's Guide and Reference Manual, Version **4**, 1 (2019), `https://docs.mosek.com/11.0/toolbox.pdf`
41. Müllner, J., Moosbrugger, M., Kovács, L.: Strong invariants are hard: On the hardness of strongest polynomial invariants for (probabilistic) programs. In: POPL (2024). https://doi.org/10.1145/3632872
42. Prajna: Optimization-Based Methods for Nonlinear and Hybrid Systems Verification. Ph.D. thesis (2005). https://doi.org/10.7907/S3BJ-4M47
43. Prajna, S., Papachristodoulou, A., Seiler, P., Parrilo, P.A.: Sostools and its control applications. Lecture Notes in Control & Information Sciences **312**, 580–580 (2005). https://doi.org/10.1007/10997703_14
44. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana University Mathematics Journal **42**(3), 969–984 (1993), `http://www.jstor.org/stable/24897130`
45. Ratschan, S., She, Z.: Providing a basin of attraction to a target region of polynomial systems by computation of lyapunov-like functions. SIAM J. Control. Optim. **48**(7), 4377–4394 (2010). https://doi.org/10.1137/090749955
46. Rebiha, R., Matringe, N., Moura, A.V.: Transcendental inductive invariants generation for non-linear differential and hybrid systems. In: HSCC (2012). https://doi.org/10.1145/2185632.2185640
47. Rice., H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American **74**(2), 358–366 (1953). https://doi.org/10.2307/1990888

48. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Science of Computer Programming **64**(1), 54–75 (2007). https://doi.org/10.1016/J.SCICO.2006.03.003
49. Rodríguez-Carbonell, E.: Some programs that need polynomial invariants in order to be verified (2016), https://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html
50. Rodríguez-Carbonell, E., Kapur, D.: An abstract interpretation approach for automatic generation of polynomial invariants. In: SAS (2004). https://doi.org/10.1007/978-3-540-27864-1_21
51. Ryan, G., Yao, J., Wong, J., Gu, R., Jana, S.: Learning nonlinear loop invariants with gated continuous logic networks. In: PLDI (2020)
52. Sankaranarayanan, S.: Automatic abstraction of non-linear systems using change of bases transformations. In: HSCC (2011). https://doi.org/10.1145/1967701.1967723
53. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS (2004). https://doi.org/10.1007/978-3-540-27864-1_7
54. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. Formal Methods in System Design **8559**(3), 235–256 (2016). https://doi.org/10.1007/s10703-016-0248-5
55. She, Z., Xue, B.: Computing a basin of attraction to a target region by solving bilinear semi-definite problems. In: CASC (2011). https://doi.org/10.1007/978-3-642-23568-9_26
56. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: CAV (2020). https://doi.org/10.1007/978-3-030-53291-8_9
57. Sriram Sankaranarayanan, H.S., Manna., Z.: Constructing invariants for hybrid systems. In: HSCC (2004). https://doi.org/10.1007/978-3-540-24743-2_36
58. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI (2009). https://doi.org/10.1145/1542476.1542501
59. Wang, C., Lin, F.: On polynomial expressions with c-finite recurrences in loops with nested nondeterministic branches. In: CAV (2024). https://doi.org/10.1007/978-3-031-65627-9_20
60. Wang, C., Lin, F.: Solving conditional linear recurrences for program verification: The periodic case. In: OOPSLA (2023). https://doi.org/10.1145/3586028
61. Wang, Q., Chen, M., Xue, B., Zhan, N., Katoen, J.P.: Synthesizing invariant barrier certificates via difference-of-convex programming. In: CAV (2021). https://doi.org/10.1007/978-3-030-81685-8_21
62. Wu, G., Cao, W., Yao, Y., Wei, H., Chen, T., Ma, X.: LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In: ASE (2024). https://doi.org/10.1145/3691620.3695014
63. Wu, H., Wang, Q., Xue, B., Zhan, N., Zhi, L., Yang, Z.H.: Synthesizing invariants for polynomial programs by semidefinite programming. ACM Trans. Program. Lang. Syst. (2024). https://doi.org/10.1145/3708559, just Accepted
64. Yang, Z., Lin, W., Wu, M.: Exact safety verification of hybrid systems based on bilinear SOS representation. ACM Transactions on Embedded Computing Systems **14**(1), 1–19 (2015). https://doi.org/10.1145/2629424

A Extensions Experiments

A.1 Templates

Table 3. Experimental on benchmarks 1 with template 2.

Benchmarks	Template 2	Our	Z3	PolySynth	invSDP	BCDC
car	$1 + a_1x_1 + a_2x_1^2$	0.3s	**0.2s**	>600s	1.0s	Failed
discrete	$1 + a_1x_1^2 + a_2x_2^2$	**0.2s**	>600s	>600s	Failed	Failed
logistic	$1 + a_1x_1 + a_2x_1^2$	**0.1s**	**0.1s**	>600s	0.9s	Failed
sgd 0.01	$a_0 + a_1x_1 - x_1^2$	0.9s	**0.4s**	0.5s	Failed	Failed
sgd 0.001	$a_0 + a_1x_1 - x_1^2$	2.8s	**0.1s**	0.5s	Failed	Failed
cav13-1	$a_0 + a_1x_1^2 + a_2x_2^2$	**0.4s**	>600s	>600s	Failed	Failed
cav13-2	$a_0 + a_1x_2$ $+a_2x_1^2 + a_3x_2^2$	**0.7s**	>600s	>600s	Failed	Failed
unicycle	$a_0 + a_1x_1x_2$ $+a_2x_2^2 + a_3x_1^2$	**1.5s**	>600s	>600s	Failed	267.0s
circuit	$a_0 + a_1x_1^2$ $+a_2x_2^2 + a_3x_1x_2$	**0.5s**	10.2s	>600s	Failed	21.3s
deter	$a_0 + a_1x_2^2$ $+a_2x_1^2 + a_3x_1$	**0.5s**	>600s	>600s	Failed	Failed
bound	$a_0 + a_1x_2^2 + a_2x_1^2$	**0.5s**	>600s	>600s	Failed	Failed
bound2	$a_0 + a_1x_2^2 + a_2x_1^2$	**0.2s**	>600s	>600s	Failed	Failed
contrived	$a_1x_1^2 + a_2x_2^2$ $+a_3x_1 + a_4x_2 - 1$	**0.5s**	>600s	>600s	Failed	Failed
transcend	$a_0 - x_1^2 - x_2^2 + a_3x_2$	1.0s	>600s	**0.8s**	Failed	404.0s
basin	$a_0 + a_1x_1^2$ $+a_2x_2^2 + a_3x_1 + a_4x_2$	**16.7s**	>600s	>600s	Failed	Failed
lyapunov	$a_0 + a_1x_1^2 + a_2x_2^2$ $+a_3x_3^2 + a_4x_1x_2 +$ $a_5x_2x_3 + a_6x_1x_3$	**3.9s**	>600s	>600s	Failed	Failed
motivate	$a_0 + a_1x_2$ $+a_2x_2^2 + a_3x_3$	**1.0s**	>600s	>600s	Failed	Failed

Templates. Generally, different invariant templates imply different search spaces for a solver, leading to different solutions. we specialize template 1 to template 2 by fixing some coefficients to 0, 1, or -1 in template 1 so that template 2 with fewer parameters. For example, we set "a0" as 1 in benchmark car, so the template becomes $1 + a_1x_1 + a_2x_1^2$. The results show that our algorithm is relatively insensitive to the template specified, which allows

our algorithm broader application scenarios. The change of templates brings significant improvements for Z3, PolySynth and invSDP. This may be due to fewer parameters means less search space, leading to less search efforts for these methods. Meanwhile, for BCDC, its iterative process requires the involvement of all coefficients of the constraint system, and merely reducing the parameters of invariant is unlikely to improve its time efficiency.

A.2 Craig Interpolant as Initial.

In the constraint system (11), the Craig interpolant corresponds to the first and third constraints, indicating that the Craig interpolant is a necessary condition for a loop invariant. We evaluate two scenarios: "OnlyCraig," which only takes the Craig interpolant as the candidate invariant, and "NoCraig," which means taking "0" rather than the Craig interpolant as the initial value in our method. Table 4 shows that "OnlyCraig" can only solve about one-third of the problems. When taking 0 as the initial value, which is the same as Prajna's method, our algorithm still outperforms Prajna's approach. This also shows that our iterative solving process does have some advantages. In addition, the results of "NoCraig" and "Ours" have shown that using the Craig interpolant as the initial value significantly improves the effectiveness and efficiency of the algorithm, while only requiring a small amount of additional time.

Table 4. Some statistics of Craig interpolant.

Algorithm	Solved	Unsolved	Avg.Time
OnlyCraig	11	24	**0.3s**
NoCraig	16	19	1.3s
Prajna	10	25	4.8s
Ours	**33**	**2**	1.6s

B Proofs

B.1 Proof of Theorem 2

Proof. From Definition 1 and formula (5), in order to prove $h(\mathbf{x}) \geq 0$ is an invariant of the polynomial program (4), we just need to prove the following three implications hold

$$f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models h(\mathbf{x}) \geq 0, \tag{22}$$

$$h(\mathbf{x}) \geq 0 \wedge c(\mathbf{x}) \geq 0 \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) \models h(\mathbf{y}) \geq 0, \tag{23}$$

$$g_1(\mathbf{x}) \geq 0 \wedge \cdots g_n(\mathbf{x}) \geq 0 \wedge c(\mathbf{x}) < 0 \models h(\mathbf{x}) < 0. \tag{24}$$

Since $u_1(\mathbf{x}), \ldots, u_m(\mathbf{x})$ are all SOS polynomials, we have that the implication (22) holds from condition i in Theorem 2. Since $w_h(\mathbf{x}, \mathbf{y})$ and $w_c(\mathbf{x}, \mathbf{y})$ are both SOS polynomials, we have that the implication (23) holds from condition ii in Theorem 2. Since $v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_n(\mathbf{x})$ are all SOS polynomials and $\epsilon > 0$, we have that the implication (24) holds from condition iii in Theorem 2. Thus, the Theorem 2 holds.

B.2 Proof of Theorem 3

Proof. Under the given conditions, the constraints in problem (12) hold. Consider the following constraint in problem (12):

$$h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x}, \mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x}, \mathbf{y})h(\mathbf{x}) - w_c(\mathbf{x}, \mathbf{y})c(\mathbf{x}) + \lambda t(\mathbf{x}, \mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2$$

With the definition of $t(\mathbf{x}, \mathbf{y})$, we have $t(\mathbf{x}, \mathbf{y}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2$. If $\lambda \leq 0$, then $-\lambda t(\mathbf{x}, \mathbf{y})$ is a SOS polynomial in problem (12). Thus, the following constraint holds:

$$h(\mathbf{y}) - \sum_{i=1}^{d} w_i(\mathbf{x}, \mathbf{y})(y_i - p_i(\mathbf{x})) - w_h(\mathbf{x}, \mathbf{y})h(\mathbf{x}) - w_c(\mathbf{x}, \mathbf{y})c(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2$$

Combined with constraints in problem (12), we can conclude that the constraints (11) hold. From Theorem 2, we have $h(\mathbf{x})$ as a solution to problem (11), i.e., $h(\mathbf{x}) \geq 0$ as an invariant of the polynomial program (4).

B.3 Proof of Theorem 4

Proof. In the T-th iteration, consider the optimization problem(17). The optimal solution of (16), combined with $h^{(T)}(x)$ is a feasible solution for problem(17). $\lambda_2^{(T)}$ is just the optimal value for λ in (17). Thus, $\lambda_1^{(T)} \geq \lambda_2^{(T)}$ holds, as the optimal solution value for λ is no greater than the feasible solution value. In the $(T+1)$-th iteration, the solution obtained from T-th iteration, combined with $w_h^{(T)}(\mathbf{x}, \mathbf{y})$ is just a feasible solution for problem (16). Thus, the optimal solution for (16), i.e., $\lambda_1^{(T+1)}$, is no greater than $\lambda_2^{(T)}$, which means that $\lambda_2^{(T)} \geq \lambda_1^{(T+1)}$. Repeat the process, we can naturally obtain the sequence in the theorem.

B.4 Proof of Theorem 5

Proof. If $\lambda_1^{(T)} \leq 0$, with Theorem 4, we have $\lambda_2^{(T)} \leq \lambda_1^{(T)} \leq 0$. If $\lambda_2^{(T)} \leq 0$, The solution for problem (17), i.e., $\lambda_2^{(T)}$, $h^{(T+1)}(\mathbf{x})$, $w_1(\mathbf{x}, \mathbf{y})$, $\ldots$, $w_d(\mathbf{x}, \mathbf{y})$, $u_1(\mathbf{x}), ..., u_m(\mathbf{x})$, $v_1(\mathbf{x}), ..., v_n(\mathbf{x})$, $w_c(\mathbf{x}, \mathbf{y})$ and $c(\mathbf{x})$ in problem (17), combined with $w_h^{(T)}(\mathbf{x}, \mathbf{y})$, is just a feasible solution for problem 12. With Theorem 3, we can conclude that $h^{(T+1)}(\mathbf{x})$ is just an invariant for polynomial program (4).

B.5 Proof of Lemma 1

Proof. Firstly, we prove that for any two integers $j_1 > j_2 \geq 0$

$$x_i^{2j_1} - x_i^{2j_2} + 1 \in \sum \mathbb{R}[x_i]^2. \tag{25}$$

Since $x_i^{2j_1} - x_i^{2j_2} + 1 \geq 0$ for any $x_i \in \mathbb{R}$ holds, and $x_i^{2j_1} - x_i^{2j_2} + 1$ is an univariate polynomial, from the Hilbert's 17th Problem [38], we have that $x_i^{2j_1} - x_i^{2j_2} + 1$ is a sum of squares, i.e., formula (25) holds.

Secondly, we prove that for any monomial $\mathbf{x}^\alpha = x_1^{\alpha_1} \cdots x_d^{\alpha_d}$, there exists $k_\alpha \in \mathbb{N}$ such that

$$1 + x_1^{2k_\alpha} + ... + x_d^{2k_\alpha} - \mathbf{x}^\alpha \in \sum \mathbb{R}[\mathbf{x}]^2. \tag{26}$$

Consider the following identity

$$\sum_{i=1}^{d-1} \frac{1}{2^i} x_i^{2^i\alpha_i} + \frac{1}{2^{d-1}} x_d^{2^{d-1}\alpha_d} - \mathbf{x}^\alpha = \sum_{i=1}^{d-1} \frac{1}{2^i} (x_i^{2^{i-1}\alpha_i} - x_{i+1}^{2^{i-1}\alpha_{i+1}} \cdots x_d^{2^{i-1}\alpha_d})^2$$

i.e., $\sum_{i=1}^{d-1} \frac{1}{2^i} x_i^{2^i\alpha_i} + \frac{1}{2^{d-1}} x_d^{2^{d-1}\alpha_d} - \mathbf{x}^\alpha$ is a SOS polynomial. Let

$$k_\alpha = \max\{\alpha_1, 2\alpha_2, \ldots, 2^{d-2}\alpha_{d-1}, 2^{d-2}\alpha_d\},$$

from formula (25), it's not hard to obtain

$$\begin{aligned} &1 + x_1^{2k_\alpha} + ... + x_d^{2k_\alpha} - \sum_{i=1}^{d-1} \frac{1}{2^i} x_i^{2^i\alpha_i} - \frac{1}{2^{d-1}} x_d^{2^{d-1}\alpha_d} \\ =&\sum_{i=1}^{d-1} \Big(\frac{2^i - 1}{2^i} x_i^{2k_\alpha} + \frac{1}{2^i} (x_i^{2k_\alpha} - x_i^{2^i\alpha_i} + 1) \Big) \\ &+ \frac{2^{d-1} - 1}{2^{d-1}} x_d^{2k_\alpha} + \frac{1}{2^{d-1}} (x_d^{2k_\alpha} - x_d^{2^{d-1}\alpha_d} + 1) \in \sum \mathbb{R}[\mathbf{x}]. \end{aligned}$$

Therefore, the formula (26) holds.

Suppose $p(\mathbf{x}) = \sum_{\alpha \in \Lambda} c_\alpha x^\alpha$, let $k = \max\{k_\alpha \mid \alpha \in \Lambda\}$, $a = \sum_{\alpha \in \Lambda} |c_\alpha|$, then it can be checked that $a(1 + x_1^{2k} + ... + x_d^{2k}) - p(\mathbf{x})$ is a SOS polynomial.

B.6 Proof of Lemma 2

Proof. Since $f, -f \in \mathcal{M}$, $(p + \frac{1}{2})^2$ and $p^2 + \frac{1}{4}$ are SOS polynomials, then $(p + \frac{1}{2})^2 f + (p^2 + \frac{1}{4})(-f) \in \mathcal{M}$, i.e., $pf \in \mathcal{M}$.

B.7 Proof of Lemma 3

Proof. Since $\mathcal{M} \subset \mathbb{R}[\mathbf{x}]$ is an Archimedean quadratic module, then there exists $a_1 > 0$ such that $a_1 - (x_1^2 + ... + x_d^2) \in \mathcal{M}$. Therefore, $a_1 - x_i^2 \in \mathcal{M}$, $i = 1, ..., d$. If $k = 0$ or $k = 1$, it's easy to see this lemma holds. When $k \geq 2$, since

$$a_1^k - x_i^{2k} = a_1^k - (x_i^2)^k = (a_1 - x_i^2)(a_1^{k-1} + a_1^{k-2} x_i^2 + \ldots + a_1 x_i^{2k-4} + x_i^{2k-2}),$$

and $(a_1^{k-1} + a_1^{k-2} x_i^2 + \ldots + a_1 x_i^{2k-4} + x_i^{2k-2})$ is a SOS polynomial, we have $a_1^k - x_i^{2k} \in \mathcal{M}$. Thus, let $a = d a_1^k$, then $a - (x_1^{2k} + ... + x_d^{2k}) \in \mathcal{M}$.

B.8 Proof of Theorem 6

Proof. From Lemma 1 and Lemma 3, it can be proved that there exists $a > 0$ such that $a - p^2(\mathbf{x}) \in \mathcal{M}_{\mathbf{x},x_{d+1}}(f_1, ..., f_m, f_{m+1}, -f_{m+1})$. Combined with Lemma 2, then $\mathcal{M}_{\mathbf{x},x_{d+1}}$ is an Archimedean quadratic module.

B.9 Proof of Theorem 7

Proof. It can be proved by mathematical induction. Obviously, ϕ_{A_0} is Archimedean. Suppose ϕ_{A_i} is Archimedean, then we just need to prove $\phi_{A_{i+1}}$ is also Archimedean.

Since ϕ_{A_i} is Archimedean, then $\phi_{A_{i-1}}(\mathbf{y}) \wedge Cond(\mathbf{y})$ is also Archimedean. For j from 1 to d, adding $x_i - p_i(\mathbf{y})$ and $-x_i + p_i(\mathbf{y})$ to the formula $\phi_{A_i}(\mathbf{y}) \wedge Cond(\mathbf{y})$, using Theorem 6 with d times, we have that $\mathbf{x} = \mathbf{p}(\mathbf{y}) \wedge \phi_{A_i}(\mathbf{y}) \wedge Cond(\mathbf{y})$ is Archimedean, that is, $\phi_{A_{i+1}}$ is Archimedean.

B.10 Proof of Theorem 8

Proof. The correctness of the case that returns "$\mathcal{I}$" can be obtained from Theorem 5. For the case of return "Error", $\bigvee_{i=0}^{a} \phi_{A_i}(\mathbf{x}) \wedge \psi_{B_0}(\mathbf{x}) \models \bot$ does not hold, a counterexample can be constructed for the program loop. Thus, the Hoare triple does not hold.

B.11 Proof of Theorem 9

Proof. We just need to prove the termination of calculating Craig interpolant in line 3 and the while loop in lines 8-20. The termination of calculating the Craig interpolant can be obtained from [18]. The termination of the while loop in lines 8-20 can be proved using Theorem 4. Thus, Algorithm 1 terminates in finite steps.

B.12 Proof of Theorem 10

Proof. From the first assumption we have that

$$\begin{aligned} Pre(\mathbf{x}) &\models h(\mathbf{x}) > 0, \\ h(\mathbf{x}) \geq 0 \wedge Cond(\mathbf{x}) \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) &\models h(\mathbf{y}) > 0 \\ h(\mathbf{x}) \geq 0 \wedge \neg Cond(\mathbf{x}) &\models Post(\mathbf{x}) \end{aligned} \tag{27}$$

Combing with Eq. (5) and adding $N - \sum_{i=1}^{d} x_i^2 \geq 0$ into the second expression of implication, we have

$$f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models h(\mathbf{x}) > 0, \tag{28}$$

$$h(\mathbf{x}) \geq 0 \wedge c(\mathbf{x}) \geq 0 \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) \wedge N - \sum_{i=1}^{d} x_i^2 \geq 0 \models h(\mathbf{y}) > 0, \tag{29}$$

$$-c(\mathbf{x}) \geq 0 \wedge g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0 \models -h(\mathbf{x}) > 0. \tag{30}$$

Since $\neg Cond(\mathbf{x}) \wedge \neg POST(\mathbf{x})$, i.e., $\mathcal{M}(-c(\mathbf{x}), g_1(\mathbf{x}), ..., g_n(\mathbf{x}))$ is Archimedean, then the semi-algebraic set

$$K = \{\mathbf{x} \mid -c(\mathbf{x}) \geq 0 \wedge g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0\} \tag{31}$$

is compact and $-h > 0$ on K. Moreover, there exists $\epsilon > 0$ such that $-h - \epsilon > 0$ on K. Therefore,

$$f_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}) \geq 0 \models h(\mathbf{x}) > 0, \tag{32}$$

$$h(\mathbf{x}) \geq 0 \wedge c(\mathbf{x}) \geq 0 \wedge \mathbf{y} = \mathbf{p}(\mathbf{x}) \wedge N - \sum_{i=1}^{d} x_i^2 \geq 0 \models h(\mathbf{y}) > 0, \tag{33}$$

$$-c(\mathbf{x}) \geq 0 \wedge g_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}) \geq 0 \models -h(\mathbf{x}) - \epsilon > 0. \tag{34}$$

Using Theorem 1 we can get the Theorem 10 holds.