

Solving Invariant Generation for Unsolvable Loops

Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács,
Marcel Moosbrugger, and Miroslav Stankovič

TU Wien, Vienna, Austria

amrollahi.daneshvar@gmail.com, ezio.bartocci@tuwien.ac.at,
george.kenison@tuwien.ac.at, laura.kovacs@tuwien.ac.at,
marcel.moosbrugger@tuwien.ac.at, miroslav.ms.stankovic@gmail.com.

Abstract. Automatically generating invariants, key to computer-aided analysis of probabilistic and deterministic programs and compiler optimisation, is a challenging open problem. Whilst the problem is in general undecidable, the goal is settled for restricted classes of loops. For the class of *solvable* loops, introduced by Kapur and Rodríguez-Carbonell in 2004, one can automatically compute invariants from closed-form solutions of recurrence equations that model the loop behaviour. In this paper we establish a technique for invariant synthesis for loops that are not solvable, termed *unsolvable* loops. Our approach automatically partitions the program variables and identifies the so-called *defective* variables that characterise unsolvability. We further present a novel technique that automatically synthesises polynomials, in the defective variables, that admit closed-form solutions and thus lead to polynomial loop invariants. Our implementation and experiments demonstrate both the feasibility and applicability of our approach to both deterministic and probabilistic programs.

Keywords: Invariant Synthesis · Algebraic Recurrences · Verification · Solvable Operators

1 Introduction

With substantial progress in computer-aided program analysis and automated reasoning, several techniques have emerged to automatically synthesise (inductive) loop invariants, thus advancing a central challenge in the computer-aided verification of programs with loops. In this paper, we address the problem of automatically generating loop invariants in the presence of polynomial arithmetic, which is still unsolved.

Inductive loop invariants, in the sequel simply *invariants*, are properties that hold before and after every iteration of a loop. As such, invariants provide the key inductive arguments for automating the verification of programs; for example, proving correctness of deterministic loops [28,21,26,20,15] and correctness of hybrid and probabilistic loops [12,16,1], or data flow analysis and compiler

optimisation [25]. One challenging aspect in invariant synthesis is the derivation of *polynomial invariants* for arithmetic loops. Such invariants are defined by polynomial relations $P(x_1, \dots, x_k) = 0$ among the program variables x_1, \dots, x_k . While deriving polynomial invariants is, in general, undecidable [11], efficient invariant synthesis techniques emerge when considering restricted classes of polynomial arithmetic in so-called *solvable loops* [28], such as loops with (blocks of) affine assignments [21,26,15,20].

A common approach for constructing polynomial invariants, pioneered by [6,17], is to (i) map a loop to a system of recurrence equations modelling the behaviour of program variables; (ii) derive closed-forms for program variables by solving the recurrences; and (iii) compute polynomial invariants by eliminating the loop counter n from the closed-forms. The central components in this setting follow. In step (i) a *recurrence operator* is employed to map loops to recurrences, which leads to closed-forms for the program variables as *exponential polynomials* in step (ii); that is, each program variable is written as a finite sum of the form $\sum_j P_j(n)\lambda_j^n$ parameterised by the n th loop iteration for polynomials P_j and algebraic numbers λ_j . From the theory of algebraic recurrences, this is the case if and only if the behaviour of each variable obeys a linear recurrence equation with constant coefficients [7,18]. Exploiting this result, the class of recurrence operators that can be linearised are called *solvable* [28]. Intuitively, a loop with a recurrence operator is solvable if the resulting system of polynomial recurrences exhibits only acyclic non-linear dependencies (see Section 3). However, even simple loops may fall outside the category of solvable operators, but still admit polynomial invariants and closed-forms for combinations of variables. This phenomenon is illustrated in Figure 1 whose recurrence operators are not solvable (i.e. unsolvable). In general, the main obstacle in the setting of unsolvable recurrence operators is the absence of “well-behaved” closed-forms for the resulting recurrences.

Related Work. To the best of our knowledge, the study of invariant synthesis from the viewpoint of recurrence operators is mostly limited to the setting of solvable operators (or minor generalisations thereof). In [28,27] the authors introduce solvable loops and mappings to model loops with (blocks of) affine assignments and propose solutions for steps (i)–(iii) for this class of loops: all polynomial invariants are derived by first solving linear recurrence equations and then eliminating variables based on Gröbner basis computation. These results have further been generalised in [21,15] to handle more generic recurrences; in particular, deriving arbitrary exponential polynomials as closed-forms of loop variables and allowing restricted multiplication among recursively updated loop variables. The authors of [8,20] generalise the setting: they consider more complex programs and devise abstract (wedge) domains to map the invariant generation problem to the problem of solving *C-finite recurrences*. (We give further details of this class of recurrences in the Section 2). All the aforementioned approaches are mainly restricted to C-finite recurrences for which closed-forms always exist, thus yielding loop invariants. In [1,2] the authors establish techniques to apply invariant synthesis techniques developed for deterministic loops to probabilistic programs.

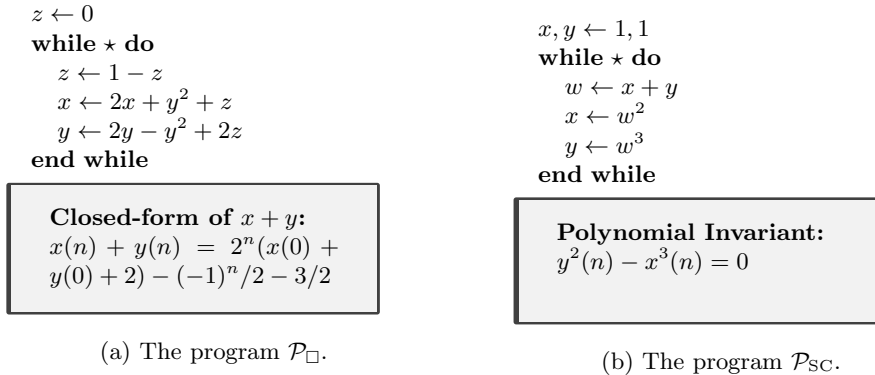


Fig. 1: Two running examples with unsolvable recurrence operators. Nevertheless, \mathcal{P}_\square admits a closed-form for combinations of variables and \mathcal{P}_{SC} admits a polynomial invariant. Herein we use \star (rather than a loop guard or `true`) as loop termination is not our focus.

Instead of devising recurrences describing the precise value of variables in step (i), their approach produces C-finite recurrences describing (higher) moments of program variables, yielding moment-based invariants after step (iii).

Pushing the boundaries in analyzing unsolvable loops is addressed in [20,9]. The approach of [20] extracts C-finite recurrences over linear combinations of loops variables from unsolvable loops. For example, the method presented in [20] can also synthesise the closed-forms identified by our work for Figure 1a. However, unlike [20], our work is not limited to linear combinations but can extract C-finite recurrences over *polynomial* relations in the loop variables. As such, the technique of [20] cannot synthesise the polynomial loop invariant in Figure 1b, whereas our work can. A further related approach to our work is given in [9], yet in the setting of loop termination. However, our work is not restricted to solvable loops that are triangular, but can handle mutual dependencies among (unsolvable) loop variables, as evidenced in Figure 1.

Our Contributions. In this paper we tackle the problem of invariant synthesis in the setting of unsolvable recurrence operators. We introduce the notions of *effective* and *defective* program variables where, figuratively speaking, the defective variables are those “responsible” for unsolvability. Our main contributions are summarized below.

1. Crucial for our synthesis technique is our novel characterisation of unsolvable recurrence operators in terms of defective variables (Theorem 1). Our approach complements existing techniques in loop analysis, by extending these methods to the setting of ‘unsolvable’ loops.
2. On the one hand, defective variables do not generally admit closed-forms. On the other hand, some polynomial combinations of such variables are well-

behaved (see e.g., Figure 1). We show how to compute the set of defective variables in polynomial time (Algorithm 1).

3. We introduce a new technique to synthesise valid polynomial relations among defective variables such that these relations admit closed-forms, from which polynomial loop invariants follow (Section 5).
4. We provide a fully automated approach in the tool `Polar`¹. Our experiments demonstrate the feasibility of invariant synthesis for ‘unsolvable’ loops and the applicability of our approach to deterministic loops, probabilistic models, and biological systems (Section 7).

Beyond Invariant Synthesis. We believe our work can provide new solutions towards compiler optimisation challenges. *Scalar evolution*² is a technique to detect general induction variables. Scalar evolution and general induction variables are used for a multitude of compiler optimisations, for example inside the LLVM toolchain [22]. On a high-level, general induction variables are loop variables that satisfy linear recurrences. As we show in our work, defective variables do not satisfy linear recurrences in general; hence, scalar evolution optimisations cannot be applied upon them. However, some polynomial combinations of defective variables *do* satisfy linear recurrences, opening up thus the venue to apply scalar evolution techniques over such defective variables. Our work automatically computes polynomial combinations of some defective loop variables, potentially thus contributing towards enlarging the class of loops that, for example, LLVM can optimise.

Structure and Summary of Results. The rest of this paper is organised as follows. We briefly recall preliminary material in Section 2.

Section 3 abstracts from concrete recurrence-based approaches to invariant synthesis via recurrence operators.

Section 4 introduces effective and defective variables, presents Algorithm 1 that computes the set of defective program variables in polynomial time, and characterises unsolvable loops in terms of defective variables (Theorem 1).

In Section 5 we present our new technique that synthesises polynomials in defective variables that admit well-behaved closed-forms. We illustrate our approach with several case-studies in Section 6, and describe a fully-automated tool support of our work in Section 7. We also report on accompanying experimental evaluation in Sections 6–7, and conclude the paper in Section 8.

2 Preliminaries

Throughout this paper, we write \mathbb{N} , \mathbb{Q} , and \mathbb{R} to respectively denote the sets of natural, rational, and real numbers. We write $\overline{\mathbb{Q}}$, the real algebraic closure of \mathbb{Q} , to denote the field of real algebraic numbers. We write $\mathbb{R}[x_1, \dots, x_k]$ and

¹ <https://github.com/probing-lab/polar>

² <https://llvm.org/docs/Passes.html>

$\overline{\mathbb{Q}}[x_1, \dots, x_k]$ for the polynomial rings of all polynomials $P(x_1, \dots, x_k)$ in k variables x_1, \dots, x_k with coefficients in \mathbb{R} and $\overline{\mathbb{Q}}$, respectively (with $k \in \mathbb{N}$ and $k \neq 0$). A *monomial* is a monic polynomial with a single term.

For a program \mathcal{P} , $\text{Vars}(\mathcal{P})$ denotes the set of program variables. We adopt the following syntax in our examples. Sequential assignments in while loops are listed on separate lines (as demonstrated in Figure 1). In programs where simultaneous assignments are performed, we employ vector notation (as demonstrated by the assignments to the variables x and y in program \mathcal{P}_{MC} in Example 2).

We refer to a directed graph with G , whose edge and vertex (node) sets are respectively denoted via $A(G)$ and $V(G)$. We endow each element of $A(G)$ with a label according to a labelling function \mathcal{L} . A *path* in G is a finite sequence of contiguous edges of G , whereas a *cycle* in G is a path whose initial and terminal vertices coincide. A graph that contains no cycles is *acyclic*. In a graph G , if there exists a path from vertex u to vertex v , then we say that v is *reachable* from vertex u and say that u is a *predecessor* of v .

C-finite recurrences. We recall relevant results on (algebraic) recurrences and refer to [7,18] for further details. A *sequence* in $\overline{\mathbb{Q}}$ is a function $u: \mathbb{N} \rightarrow \overline{\mathbb{Q}}$, shortly written also as $\langle u(n) \rangle_{n=0}^\infty$ or simply just $\langle u(n) \rangle_n$. A *recurrence* for a sequence $\langle u(n) \rangle_n$ is an equation $u(n+\ell) = \text{Rec}(u(n+\ell-1), \dots, u(n+1), u(n), n)$, for some function $\text{Rec}: \mathbb{R}^{\ell+1} \rightarrow \mathbb{R}$. The number $\ell \in \mathbb{N}$ is the *order* of the recurrence.

A special class of recurrences we consider are the *linear recurrences with constant coefficients*, in short *C-finite recurrences*. A C-finite recurrence for a sequence $\langle u(n) \rangle_n$ is an equation of the form

$$u(n+\ell) = a_{\ell-1}u(n+\ell-1) + a_{\ell-2}u(n+\ell-2) + \dots + a_0u(n) \quad (1)$$

where $a_0, \dots, a_{\ell-1} \in \overline{\mathbb{Q}}$ are constants and $a_0 \neq 0$. A sequence $\langle u(n) \rangle_n$ satisfying a C-finite recurrence (1) is a *C-finite sequence* and is uniquely determined by its initial values $u_0 = u(0), \dots, u_{\ell-1} = u(\ell-1)$. The *characteristic polynomial* associated with the C-finite recurrence relation (1) is

$$x^{n+\ell} - a_{\ell-1}x^{n+\ell-1} - a_{\ell-2}x^{n+\ell-2} - \dots - a_0x^n.$$

The terms of a C-finite sequence can be written in a closed-form as exponential polynomials, depending only on n and the initial values of the sequence. That is, if $\langle u(n) \rangle_n$ is determined by a C-finite recurrence (1), then $u(n) = \sum_{k=1}^r P_k(n) \lambda_k^n$ where $P_k(n) \in \overline{\mathbb{Q}}[n]$ and $\lambda_1, \dots, \lambda_r$ are the roots of the associated characteristic polynomial. Importantly, closed-forms of (systems of) C-finite sequences always exist and are computable [7,18].

Invariants. An inductive loop invariant is a loop property that holds before and after each loop iteration [10]. In this paper, whenever we refer to a loop invariant, we mean an inductive loop invariant; in particular, we are interested in *polynomial invariants* the class of invariants given by Boolean combinations of polynomial equations among loop variables. There is a minor caveat to our characterisation of (polynomial) loop invariants. We assume that a (polynomial)

invariant consists of a finite number of initial values together with a closed-form expression of a monomial in the loop variables. Thus the closed-form of a loop invariant must eventually hold after a (computable) finite number of loop iterations. Let us illustrate this caveat with the following loop:

```

x ← 0
while ★ do
  x ← 1
end while

```

Here the loop admits the polynomial invariant given by the initial value $x(0) = 1$ of x and the closed-form $x(n) = 1$. For each $n \geq 1$, we denote by $x(n)$ the value of loop variable x at loop iteration n . Herein, we synthesise invariants that satisfy inhomogeneous first-order recurrence relations and it is straightforward to show that each associated closed-form holds for $n \geq 1$.

Polynomial Invariants and Invariant Ideals. A polynomial *ideal* is a subset $I \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_k]$ with the following properties: I contains 0; I is closed under addition; and if $P \in \overline{\mathbb{Q}}[x_1, \dots, x_k]$ and $Q \in I$, then $PQ \in I$. For a set of polynomials $S \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_k]$, one can define the *ideal generated by* S by

$$I(S) := \{s_1q_1 + \dots + s_\ell q_\ell \mid s_i \in S, q_i \in \overline{\mathbb{Q}}[x_1, \dots, x_k], \ell \in \mathbb{N}\}.$$

Let \mathcal{P} be a program as before. For $x_j \in \text{Vars}(\mathcal{P})$, let $\langle x_j(n) \rangle_n$ denote the sequence whose n th term is given by the value of x_j in the n th loop iteration. The set of polynomial invariants of \mathcal{P} form an ideal, the *invariant ideal* of \mathcal{P} [27]. If for each program variable x_j the sequence $\langle x_j(n) \rangle_n$ is C-finite, then a basis for the invariant ideal can be computed as follows. Let $f_j(n)$ be the exponential polynomial closed-form of variable x_j . The exponential terms $\lambda_1^n, \dots, \lambda_s^n$ in each of the $f_j(n)$ are replaced by fresh symbols, yielding the polynomials $g_j(n)$. Next, with techniques from [19], the set R of all polynomial relations among $\lambda_1^n, \dots, \lambda_s^n$ (that hold for each $n \in \mathbb{N}$) is computed. Then we express the polynomial relations in terms of the fresh constants, so that we can interpret R as a set of polynomials. Thus

$$I(\{x_j - g_j(n) \mid 1 \leq i \leq k\} \cup R) \cap \overline{\mathbb{Q}}[x_1, \dots, x_k]$$

is precisely the invariant ideal of \mathcal{P} . Finally, we can compute a finite basis for the invariant ideal with techniques from Gröbner bases and elimination theory [19].

3 From Loops to Recurrences

Modelling properties of loop variables by algebraic recurrences and solving the resulting recurrences is an established approach in program analysis. Multiple works [21,8,20,14,15] associate a loop variable x with a sequence $\langle x(n) \rangle_n$ whose n th term is given by the value of x in the n th loop iteration. These works are primarily concerned with the problem of representing such sequences via recurrence equations whose closed-forms can be computed automatically, as in the case of C-finite sequences. A closely connected question to this line of research

focuses on identifying classes of loops that can be modelled by solvable recurrences, as advocated in [28]. To this end, over-approximation methods for general loops are proposed in [8,20] such that solvable recurrences can be obtained from (over-approximated) loops.

In order to formalise the above and similar efforts in associating loop variables with recurrences, herein we introduce the concept of a *recurrence operator*, and then *solvable* and *unsolvable operators*. Intuitively, a recurrence operator maps program variables to recurrence equations describing some properties of the variables; for instance, the exact values at the n th loop iteration [28,21,8] or statistical moments in probabilistic loops [1].

Definition 1 (Recurrence Operator). A recurrence operator \mathcal{R} maps $\text{Vars}(\mathcal{P})$ to the polynomial ring $\mathbb{R}[\text{Vars}_n(\mathcal{P})]$. The set of equations $\{x(n+1) = \mathcal{R}[x] \mid x \in \text{Vars}(\mathcal{P})\}$ constitutes a polynomial first-order system of recurrences. We call \mathcal{R} linear if $\mathcal{R}[x]$ is linear for all $x \in \text{Vars}(\mathcal{P})$.

One can extend the operator \mathcal{R} to $\mathbb{R}[\text{Vars}(\mathcal{P})]$. Then, with a slight abuse of notation, for $P(x_1, \dots, x_j) \in \mathbb{R}[\text{Vars}(\mathcal{P})]$ we define $\mathcal{R}(P)$ by $P(\mathcal{R}[x_1], \dots, \mathcal{R}[x_j])$.

For a program \mathcal{P} with recurrence operator \mathcal{R} and a monomial over program variables $M := \prod_{x \in \text{Vars}(\mathcal{P})} x^{\alpha_x}$, we denote by $M(n)$ the product of sequences $\prod_{x \in \text{Vars}(\mathcal{P})} x^{\alpha_x}(n)$. Given a polynomial P over program variables, $P(n)$ is defined by replacing every monomial M in P by $M(n)$. For a set T of polynomials over program variables let $T_n := \{P(n) \mid P \in T\}$.

Example 1. Consider the program \mathcal{P}_{SC} in Figure 1b. One can employ a recurrence operator \mathcal{R} in order to capture the values of the program variables in the n th iteration. For $v \in \text{Vars}(\mathcal{P}_{\text{SC}})$, $\mathcal{R}[v]$ is obtained by bottom-up substitution in the polynomial updates starting with v . As a result, we obtain the following system of recurrences:

$$\begin{aligned} w(n+1) &= \mathcal{R}[w] = x(n) + y(n) \\ x(n+1) &= \mathcal{R}[x] = x(n)^2 + 2x(n)y(n) + y(n)^2 \\ y(n+1) &= \mathcal{R}[y] = x(n)^3 + 3x(n)^2y(n) + 3x(n)y(n)^2 + y(n)^3. \end{aligned}$$

Similarly, for the program \mathcal{P}_{\square} of Figure 1a, we obtain the following system of recurrences:

$$\begin{aligned} z(n+1) &= \mathcal{R}[z] = 1 - z(n) \\ x(n+1) &= \mathcal{R}[x] = 2x(n) + y(n)^2 - z(n) + 1 \\ y(n+1) &= \mathcal{R}[y] = 2y(n) - y(n)^2 - 2z(n) + 2. \end{aligned}$$

Solvable Operators. Systems of linear recurrences with constant coefficients admit computable closed-form solutions as exponential polynomials [7,18]. This property holds for a larger class of recurrences with polynomial updates, which leads to the notion of *solvability* introduced in [28]. We adjust the notion of

solvability to our setting by using recurrence operators. In the following definition, we make a slight abuse of notation and order the program variables so that we can transform program variables by a matrix operator.

Definition 2 (Solvable Operators [28,26]). *The recurrence operator \mathcal{R} is solvable if there exists a partition of Vars_n ; that is, $\text{Vars}_n = W_1 \uplus \dots \uplus W_k$ such that for $x(n) \in W_j$,*

$$\mathcal{R}[x] = M_j \cdot W_j^\top + P_j(W_1, \dots, W_{j-1})$$

for some matrices M_j and polynomials P_j . A recurrence operator that is not solvable is said to be unsolvable.

This definition captures the notion of solvability in [28] (see the discussion in [26]).

We conclude this section by emphasising the use of (solvable) recurrence operators beyond deterministic loops, in particular relating its use to probabilistic program loops. As evidenced in [1], recurrence operators model statistical moments of program variables by essentially focusing on solvable recurrence operators extended with an expectation operator $\mathbb{E}(\cdot)$ to derive closed-forms of (higher) moments of program variables, as illustrated below.

Example 2. Consider the probabilistic program \mathcal{P}_{MC} of [30,4] modelling a non-linear Markov chain, where $\text{Bernoulli}(p)$ refers to a Bernoulli distribution with parameter p . Here the updates to the program variables x and y occur simultaneously.

```

while  $\star$  do
   $s \leftarrow \text{Bernoulli}(1/2)$ 
  if  $s = 0$  then
     $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + xy \\ \frac{1}{3}x + \frac{2}{3}y + xy \end{pmatrix}$ 
  else
     $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + y + \frac{2}{3}xy \\ 2y + \frac{2}{3}xy \end{pmatrix}$ 
  end if
end while

```

One can construct recurrence equations, in terms of the expectation operator $\mathbb{E}(\cdot)$, for this program as follows:

$$\begin{aligned} \mathbb{E}(s_{n+1}) &= \frac{1}{2} \\ \mathbb{E}(x_{n+1}) &= \mathbb{E}(x_n) + \frac{1}{2}\mathbb{E}(y_n) + \frac{5}{6}\mathbb{E}(x_n y_n) \\ \mathbb{E}(y_{n+1}) &= \frac{1}{6}\mathbb{E}(x_n) + \frac{4}{3}\mathbb{E}(y_n) + \frac{5}{6}\mathbb{E}(x_n y_n). \end{aligned}$$

4 Defective Variables

To the best of our knowledge, existing approaches in loop analysis and invariant synthesis are restricted to solvable recurrence operators. In this section, we

establish a new characterisation of unsolvable recurrence operators. Our characterisation pinpoints the program variables responsible for unsolvability, the *defective variables* (see Definition 5). Moreover, we provide a polynomial time algorithm to compute the set of defective variables (Algorithm 1), in order to exploit our new characterisation for synthesising invariants in the presence of unsolvable operators in Section 5.

For simplicity, we limit the discussion in this section to deterministic programs. We note however that the results presented herein can also be applied to probabilistic programs. The details of the necessary changes in this respect are given in Appendix B.

In what follows, we write $\mathcal{M}_n(\mathcal{P})$ to denote the set of non-trivial *monomials* in $\text{Vars}(\mathcal{P})$ evaluated at the n th loop iteration so that

$$\mathcal{M}_n(\mathcal{P}) := \left\{ \prod_{x \in \text{Vars}(\mathcal{P})} x^{\alpha_x} (n) \mid \exists x \in \text{Vars}(\mathcal{P}) \text{ with } \alpha_x \neq 0 \right\}.$$

We next introduce the notions of variable dependency and dependency graph, needed to further characterise defective variables.

Definition 3 (Variable Dependency). *Let \mathcal{P} be a loop with recurrence operator \mathcal{R} and $x, y \in \text{Vars}(\mathcal{P})$. We say x depends on y if y appears in a monomial in $\mathcal{R}[x]$ with non-zero coefficient. Moreover, x depends linearly on y if all monomials with non-zero coefficients in $\mathcal{R}[x]$ containing y are linear. Analogously, x depends non-linearly on y if there is a non-linear monomial with non-zero coefficient in $\mathcal{R}[x]$ containing y .*

Furthermore, we consider the transitive closure for variable dependency. If z depends on y and y depends on x , then z depends on x and, if in addition, one of these two dependencies is non-linear, then z depends non-linearly on x . We otherwise say the dependency is linear.

For each program with polynomial updates, we further define a *dependency graph* with respect to a recurrence operator.

Definition 4 (Dependency Graph). *Let \mathcal{P} be a program with recurrence operator \mathcal{R} . The dependency graph of \mathcal{P} with respect to \mathcal{R} is the labelled directed graph $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$ with vertex set $\text{Vars}(\mathcal{P})$, edge set $A := \{(x, y) \mid x, y \in \text{Vars}(\mathcal{P}) \wedge x \text{ depends on } y\}$, and a function $\mathcal{L}: A \rightarrow \{L, N\}$ that assigns a unique label to each edge such that*

$$\mathcal{L}(x, y) = \begin{cases} L & \text{if } x \text{ depends linearly on } y, \text{ and} \\ N & \text{if } x \text{ depends non-linearly on } y. \end{cases}$$

In our approach, we partition the variables $\text{Vars}(\mathcal{P})$ of the program \mathcal{P} into two sets: *effective-* and *defective variables*, denoted by $E(\mathcal{P})$ and $D(\mathcal{P})$ respectively. Our partition builds on the definition of the dependency graph of \mathcal{P} , as follows.

Definition 5 (Effective-Defective). *A variable $x \in \text{Vars}(\mathcal{P})$ is effective if:*

1. *x appears in no directed cycle with at least one edge with an N label, and*

2. x cannot reach a vertex of an aforementioned cycle (as in 1).

A variable is defective if it is not effective.

Example 3. From the recurrence equations of Example 1 for the program \mathcal{P}_{SC} (see Figure 1b), one obtains the dependencies between the program variables of \mathcal{P}_{SC} : the program variable w depends linearly on both x and y , whilst x and y depend non-linearly on each other and on w . By definition, the partition into effective and defective variables is $E(\mathcal{P}_{\text{SC}}) = \emptyset$ and $D(\mathcal{P}_{\text{SC}}) = \{w, x, y\}$.

Similarly, we can construct the dependency graph for the program \mathcal{P}_{\square} from Figure 1a, as illustrated in Figure 2. We derive that $E(\mathcal{P}_{\square}) = \{z\}$ and $D(\mathcal{P}_{\square}) = \{x, y\}$.

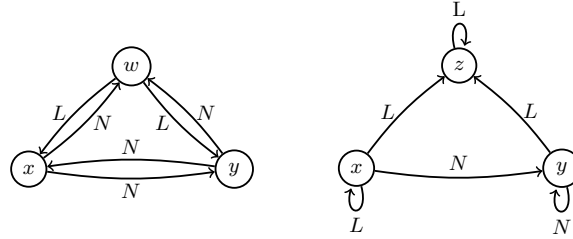


Fig. 2: The dependency graphs for \mathcal{P}_{SC} and \mathcal{P}_{\square} from Figure 1.

The concept of effective, and, especially, defective variables allows us to establish a new characterisation of programs with unsolvable recurrence operators: *a recurrence operator is unsolvable if and only if there exists a defective variable* (as stated in Theorem 1 and automated in Algorithm 1). We formalise and prove this results via the following three lemmas.

Lemma 1. *Let \mathcal{P} be a program with recurrence operator \mathcal{R} . If $D(\mathcal{P})$ is non-empty, so that there is at least one defective variable, then \mathcal{R} is unsolvable.*

Proof. Let $x \in \text{Vars}(\mathcal{P})$ be a defective variable and $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$ the dependency graph of \mathcal{P} with respect to a recurrence operator \mathcal{R} . Following Definition 5, there exists a cycle C such that x is a vertex visited by or can reach said cycle and, in addition, there is an edge in C labelled by N .

Assume, for a contradiction, that \mathcal{R} is solvable. Then there exists a partition W_1, \dots, W_k of $\text{Vars}_n(\mathcal{P})$ as described in Definition 2. Moreover, since C is a cycle, there exists $j \in \{1, \dots, k\}$ such that each variable visited by C lies in W_j . Let $(w, y) \in C$ be an edge labelled with N . Since w depends on y non-linearly, and $\mathcal{R}[w] = M_j \cdot W_j^\top + P_j(W_1, \dots, W_{j-1})$ (by Definition 2), it is clear that $y(n) \in W_\ell$ for some $\ell \neq j$. We also have that $y(n) \in W_j$ since C visits y . Thus we arrive at a contradiction as W_1, \dots, W_k is a partition of $\text{Vars}_n(\mathcal{P})$. Hence \mathcal{R} is unsolvable. \square

Given a program \mathcal{P} whose variables are all effective, it is immediate that a pair of distinct mutually dependent variables are necessarily linearly dependent and, similarly, a self-dependent variable is necessarily linearly dependent on itself. Consider the following binary relation \sim on program variables:

$$x \sim y \iff x = y \vee (x \text{ depends on } y \wedge y \text{ depends on } x).$$

Thus, any two mutually dependent variables are related by \sim . Under the assumption that all variables of a program \mathcal{P} are effective, it is easily seen that \sim defines an equivalence relation on $\text{Vars}(\mathcal{P})$. The partition of the equivalence classes Π of $\text{Vars}(\mathcal{P})$ under \sim admits the following notion of dependence between equivalence classes: for $\pi, \hat{\pi} \in \Pi$ we say that π *depends on* $\hat{\pi}$ if there exist variables $x \in \pi$ and $y \in \hat{\pi}$ such that variable x depends on variable y .

Lemma 2. *Suppose that all variables of a program \mathcal{P} are effective. Consider the graph \mathcal{G} with vertex set given by the set of equivalence classes Π and edge set $A' := \{(\pi, \hat{\pi}) \mid (\pi \neq \hat{\pi}) \wedge (\pi \text{ depends on } \hat{\pi})\}$. Then \mathcal{G} is acyclic.*

Proof. From the definition of \mathcal{G} , it is clear that the graph is directed and has no self-loops. Now assume, for a contradiction, that \mathcal{G} contains a cycle. Since the relation \sim is transitive, there exists a cycle C in \mathcal{G} of length two. Moreover, the variables in a given equivalence class are mutually dependent. Thus the elements of the two classes in C are equivalent under the relation \sim , which contradicts the partition into distinct equivalence classes. Therefore the graph \mathcal{G} is acyclic, as required. \square

Lemma 3. *Let \mathcal{P} be a program with recurrence operator \mathcal{R} . If each of the program variables of \mathcal{P} are effective then \mathcal{R} is solvable.*

Proof. By Lemma 2, the associated graph $\mathcal{G} = (\Pi, A')$ on the equivalence classes of $\text{Vars}(\mathcal{P})$ is directed and acyclic. Thus there exists a topological ordering of $\Pi = \{\pi_1, \dots, \pi_{|\Pi|}\}$ such that for every $(\pi_i, \pi_j) \in A'$ we have $i > j$. Thus if $x \in \pi_i$ then x does not depend on any variables in class π_j for $j > i$. Moreover, for each $\pi_i \in \Pi$, if $x, y \in \pi_i$ then x cannot depend on y non-linearly because every variable is effective (and all the variables in π_i are mutually dependent). Thus Π evaluated at loop iteration n partitions $\text{Vars}_n(\mathcal{P})$ and satisfies the criteria in Definition 2. We thus conclude that \mathcal{R} is solvable. \square

Together, Lemmas 1–3 yield a new characterisation of unsolvable operators.

Theorem 1 (Defective Characterisation). *Let \mathcal{P} be a program with recurrence operator \mathcal{R} , then \mathcal{R} is unsolvable if and only if $D(\mathcal{P})$ is non-empty.*

In Algorithm 1 we provide a polynomial time algorithm that constructs both $E(\mathcal{P})$ and $D(\mathcal{P})$ given a program and a recurrence operator. We use the initialism “DFS” for the *depth-first search* procedure. Algorithm 1 terminates in polynomial time as both the construction of the dependency graph and depth-first search exhibit polynomial time complexity. The procedure searches for cycles

Algorithm 1 Construct $E(\mathcal{P})$ and $D(\mathcal{P})$ from program \mathcal{P} with operator \mathcal{R} .

```

Let  $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$  be the dependency graph of  $\mathcal{P}$  with respect to  $\mathcal{R}$ .
 $D(\mathcal{P}) \leftarrow \emptyset$ 
for  $(x, y) \in A$  where  $\mathcal{L}(x, y) = N$  do
  if  $x = y$  then
    predecessor  $\leftarrow \emptyset$ 
    DFS( $x$ , predecessor)
     $D(\mathcal{P}) \leftarrow D(\mathcal{P}) \cup \text{predecessor}$ 
  end if
  if  $x \neq y$  then
    predecessor  $\leftarrow \emptyset$ 
    DFS( $y$ , predecessor)
    if  $x \in \text{predecessor}$  then
       $D(\mathcal{P}) \leftarrow D(\mathcal{P}) \cup \text{predecessor}$ 
    end if
  end if
end for
 $E(\mathcal{P}) \leftarrow \text{Vars}(\mathcal{P}) \setminus D(\mathcal{P})$ 

```

in the dependency graph with at least one non-linear edge (labelled by N). All variables that reach such cycles are, by definition, defective.

In what follows, we focus on programs with unsolvable recurrence operators, or equivalently by Theorem 1, the case where $\mathcal{D}(\mathcal{P}) \neq \emptyset$. The characterisation of unsolvable operators in terms of defective variables and our polynomial algorithm to construct the set of defective variables is the foundation for our approach synthesising invariants in the presence of unsolvable recurrence operators in Section 5.

Remark 1. The recurrence operator $\mathcal{R}[x]$ for an effective variable x will admit a closed-form solution for every initial value x_0 . For the avoidance of doubt, the same cannot be said for the recurrence operator of a defective variable. However, it is possible that a set of initial values will lead to a closed-form expression as a C-finite sequence: consider a loop with defective variable x and update $x \leftarrow x^2$ and initialisation $x_0 \leftarrow 0$ or $x_0 \leftarrow \pm 1$.

5 Synthesising Invariants

In this section we propose a new technique to *synthesise invariants for programs with unsolvable recurrence operators*. The approach is based on our new characterisation of unsolvable operators in terms of defective variables (Section 4).

For the remainder of this section we fix a program \mathcal{P} with an unsolvable recurrence operator \mathcal{R} , or equivalently with $D(\mathcal{P}) \neq \emptyset$. We start by extending the notions of *effective* and *defective* from program variables to monomials of

program variables. Let \mathcal{E} be the set of *effective monomials* given by

$$\mathcal{E}(\mathcal{P}) = \left\{ \prod_{x \in E(\mathcal{P})} x^{\alpha_x} \mid \alpha_x \in \mathbb{N} \right\}.$$

The complement, the *defective monomials*, is given by $\mathcal{D}(\mathcal{P}) := \mathcal{M}(\mathcal{P}) \setminus \mathcal{E}(\mathcal{P})$. The difficulty with defective variables is that in general they do not admit closed-forms. However, polynomials of defective variables may allow for closed-forms as illustrated in previous examples. The main idea of our technique for invariant synthesis in the presence of defective variables is to find such polynomials. We fix a *candidate polynomial* called $S(n)$ based on an arbitrary degree $d \in \mathbb{N}$:

$$S(n) = \sum_{W \in \mathcal{D}_n(\mathcal{P}) \upharpoonright_d} c_W W, \quad (2)$$

where the coefficients $c_W \in \mathbb{R}$ are unknown real constants. Our notation $\mathcal{D}_n(\mathcal{P}) \upharpoonright_d$ indicates the set of *defective monomials of degree at most d* .

Example 4. For \mathcal{P}_\square in Figure 1a we have $\mathcal{D}_n(\mathcal{P}_\square) \upharpoonright_1 = \{x, y\}$, and $\mathcal{D}_n(\mathcal{P}_\square) \upharpoonright_2 = \{x, y, x^2, y^2, xy, xz, yz\}$.

On the one hand, all variables in $S(n)$ are defective; however, $S(n)$ may admit a closed-form. This occurs if $S(n)$ obeys a “well-behaved” recurrence equation; that is to say, an inhomogeneous recurrence equation where the inhomogeneous component is given by a linear combination of effective monomials. In such instances the recurrence takes the form

$$S(n+1) = \kappa S(n) + \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M \quad (3)$$

where the coefficients c_M are unknown. Thus an intermediate step towards our goal of synthesising invariants is to determine whether there are constants $c_M, c_W, \kappa \in \mathbb{R}$ that satisfy the above equations. If such constants exist then we come to our final step: solving a first-order inhomogeneous recurrence relation. There are standard methods available to solve first-order inhomogeneous recurrences of the form $S(n+1) = \kappa S(n) + h(n)$, where $h(n)$ is the closed-form of $\sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M$, see e.g., [18]. We note $h(n)$ is computable and an exponential polynomial since it is determined by a linear sum of effective monomials. Thus $\langle S(n) \rangle_n$ is a C-finite sequence.

Remark 2. Observe that the sum on the right-hand side of equation (3) is finite, since all but finitely many of the coefficients c_M are zero. Further, the coefficient c_M of monomial M is non-zero only if M appears in $\mathcal{R}[S]$.

Going further, in equation (3) we express $S(n+1)$ in terms of a polynomial in $\text{Vars}_n(\mathcal{P})$ with unknown coefficients c_M, c_W , and κ . An alternative expression

for $S(n+1)$ in $\text{Vars}_n(\mathcal{P})$ is given by the recurrence operator $S(n+1) = \mathcal{R}[S]$. Taken in combination, we arrive at the following formula

$$\mathcal{R}[S] - \kappa S(n) - \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M = 0,$$

yielding a polynomial in $\text{Vars}_n(\mathcal{P})$. Thus all the coefficients in the above formula are necessarily zero as the polynomial is identically zero. Therefore *all* solutions to the unknowns c_M, c_W , and κ are computed by solving a (quadratic) system of equations.

Example 5. We demonstrate our procedure for invariant synthesis by applying the method to an example. Recall program \mathcal{P}_\square from Figure 1a:

```

z ← 0
while ★ do
  z ← 1 - z
  x ← 2x + y2 + z
  y ← 2y - y2 + 2z
end while

```

From Algorithm 1 we obtain $E(\mathcal{P}_\square) = \{z\}$ and $D(\mathcal{P}_\square) = \{x, y\}$. Because $D(\mathcal{P}_\square) \neq \emptyset$, we deduce using Theorem 1 that the associated operator \mathcal{R} is unsolvable. Consider the candidate $S(n) = ax(n) + by(n)$ with unknowns $a, b \in \mathbb{R}$. The recurrence for $S(n)$ given by \mathcal{R} is

$$\begin{aligned} S(n+1) &= \mathcal{R}[S] = a\mathcal{R}[x] + b\mathcal{R}[y] \\ &= a + 2b + 2ax(n) + 2by(n) - (a + 2b)z(n) + (a - b)y^2(n). \end{aligned}$$

We next express $S(n+1)$ in terms of an inhomogeneous recurrence equation (cf. equation (3)). When we substitute for $S(n)$, we obtain

$$S(n+1) = \kappa(ax(n) + by(n)) + (cz(n) + d)$$

where the coefficients in the inhomogeneous component are unknown. We then combine the preceding two equations (for brevity we suppress the loop counter n in the program variables x, y, z) and derive

$$(a + 2b - d) + (-a - c - 2b)z + (2a - \kappa a)x + (2b - \kappa b)y + (a - b)y^2 = 0.$$

Thus we have a polynomial in the program variables that is identically zero. Therefore, all the coefficients in the above equation are necessarily zero. We then solve the resulting system of quadratic equations, which leads to the non-trivial solution $a = b$, $\kappa = 2$, $d = 3a$, and $c = -3a$. We substitute this solution back into the recurrence for $\mathcal{R}[S]$ and find

$$S(n+1) = 2S(n) + 3a(1 - z(n)) = 2S(n) + 3a \frac{1 + (-1)^n}{2}.$$

Here, we have used the closed-form solution $z(n) = 1/2 - (-1)^n/2$ of the effective variable z . We can compute the solution of this inhomogeneous first-order

recurrence equation. In the case that $a = 1$, we have $S(n) = 2^n(S(0) + 2) - (-1)^n/2 - 3/2$. Therefore, the following identity holds for each $n \in \mathbb{N}$:

$$x(n) + y(n) = 2^n(x(0) + y(0) + 2) - (-1)^n/2 - 3/2$$

and so we have synthesised the closed-form of $x + y$ for program \mathcal{P}_\square of Figure 1a.

5.1 Solution Space of Invariants for Unsolvable Operators

Given a program and a recurrence operator, our invariant synthesis technique is relative-complete with respect to the degree d of the candidate $S(n)$. This means, for a fixed degree $d \in \mathbb{N}$, our approach is in theory able to compute *all* polynomials of defective variables with maximum degree d that satisfy a “well-behaved” recurrence; that is, a first-order recurrence equation of the form (3). This holds because of our reduction of the problem to a system of quadratic equations for which all solutions are computable. Our technique can also rule out the existence of well-behaved polynomials of defective variables of degree at most d if the resulting system has no (non-trivial) solutions.

Let \mathcal{P} be a program with program variables $\text{Vars}(\mathcal{P}) = \{x_1, \dots, x_k\}$. The set of polynomials P with $P(x_1(n), \dots, x_k(n)) = 0$ for all $n \in \mathbb{N}$ form an ideal, the *invariant ideal* of \mathcal{P} . The requirement of closed-forms is the main obstacle for computing a basis for the invariant ideal in the presence of defective variables. Our work introduces a method that includes defective variables in the computation of invariant ideals, via the following steps of deriving the *polynomial invariant ideal of an unsolvable loop*:

- For every effective variable x_i , let $f_i(n)$ be its closed-form and assume $h(n)$ is the closed-form for some candidate S given by a polynomial in defective variables.
- Let $\lambda_1^n, \dots, \lambda_s^n$ be the exponential terms in all $f_i(n)$ and $h(n)$. Replace the exponential terms in all $f_i(n)$ as well as $h(n)$ by fresh constants to construct the polynomials $g_i(n)$ and $l(n)$ respectively.
- Next, construct the set R of polynomial relations among all exponential terms, as explained in Section 2. Then, the ideal

$$I(\{x_i - g_i(n) \mid x_i \in E(\mathcal{P})\} \cup \{S - l(n)\} \cup R) \cap \overline{\mathbb{Q}}[x_1, \dots, x_k]$$

contains precisely all polynomial relations among program variables implied by the equations $\{x_i = f_i(n)\} \cup \{S = g(n)\}$ in the theory of polynomial arithmetic.

- A finite basis for this ideal is computed using techniques from Gröbner bases and elimination theory. This step is similar to the case of the invariant ideal for solvable loops, see e.g., [28,21].

In conclusion, we infer a *finite representation of the ideal of polynomial invariants for loops with unsolvable recurrence operators*.

6 Applications of Unsolvable Operators towards Invariant Synthesis

Our approach automatically generates invariants for programs with defective variables (Section 5), and pushes the boundaries of both theory and practice of invariant generation: we introduce and incorporate defective variable analysis into the state-of-the-art methodology of reasoning about solvable loops, complementing thus existing methods, see e.g., [28,21,20,15], in the area. As such, the class of unsolvable loops that can be handled by our work extends (aforementioned) existing approaches on polynomial invariant synthesis. The experimental results of our approach (see Section 7) demonstrate the efficiency and scalability of our work in deriving invariants for unsolvable loops. Since our approach to loops via recurrences is generic, we can deal with emerging applications of programming paradigms such as: transitions systems and statistical moments in probabilistic programs; and reasoning about biological systems. We showcase these applications in this section and also exemplify the limitations of our work. In the sequel, we write $\mathbb{E}(t)$ to refer to the expected value of an expression t , and denote by $\mathbb{E}(t_n)$ (or $\mathbb{E}(t(n))$) the expected value of t at loop iteration n .

Example 6 (Moments of Probabilistic Programs [30]). Recall the program \mathcal{P}_{MC} of Example 2. One can easily verify that $\mathbb{E}(x_n - y_n) = \frac{5^n}{6^n}(x_0 - y_0)$ and so obtain an invariant for \mathcal{P}_{MC} . Closed-form solutions for higher order expressions are also available; for example,

$$\mathbb{E}((x_n - y_n)^d) = \frac{(2^d + 3^d)^n}{2^n \cdot 3^{dn}}(x_0 - y_0)^d$$

refers to the d th moment of $x(n) - y(n)$. While the work in [30] uses martingale theory to synthesise the above invariant (of degree 1), our approach automatically generates such invariants over higher-order moments (see Table 2). We note to this end that the defective variables in \mathcal{P}_{MC} are precisely x and y as can be seen from their mutual non-linear interdependence. Namely, we have $D(\mathcal{P}_{MC}) = \{x, y\}$ and $E(\mathcal{P}_{MC}) = \{s\}$.

Example 7 (Biological Systems [3]). A widely-cited model for the decision-making process of swarming bees choosing one nest-site from a selection of two is introduced in [3] and further studied in [5,29]. An interesting question on this example arises with computing probability distributions, yielding answers to probabilistic reachability [29]. The (unsolvable) loop below yields a discrete-time model with five classes of bees (each represented by a program variable) and the proportion of bees in each class changes simultaneously at each loop execution. Here the coefficient Δ indicates the length of the time-step in the discrete-time model and the remaining coefficients parameterise the rates of these changes. All coefficients here are symbolic (representing any real number).

$$\begin{array}{l}
\begin{pmatrix} x \\ y_1 \\ y_2 \\ z_1 \\ z_2 \end{pmatrix} \leftarrow \begin{pmatrix} \text{Normal}(475, 5) \\ \text{Uniform}(350, 400) \\ \text{Uniform}(100, 150) \\ \text{Normal}(35, 1.5) \\ \text{Normal}(35, 1.5) \end{pmatrix} \\
\mathbf{while} \star \mathbf{do} \\
\begin{pmatrix} x \\ y_1 \\ y_2 \\ z_1 \\ z_2 \end{pmatrix} \leftarrow \begin{pmatrix} x - \Delta(\beta_1 x y_1 + \beta_2 x y_2) \\ y_1 + \Delta(\beta_1 x y_1 - \gamma y_1 + \delta \beta_1 y_1 z_1 + \alpha \beta_1 y_1 z_2) \\ y_2 + \Delta(\beta_2 x y_2 - \gamma y_2 + \delta \beta_2 y_2 z_2 + \alpha \beta_2 y_2 z_1) \\ z_1 \leftarrow z_1 + \Delta(\gamma y_1 - \delta \beta_1 y_1 z_1 - \alpha \beta_2 y_2 z_1) \\ z_2 \leftarrow z_2 + \Delta(\gamma y_2 - \delta \beta_2 y_2 z_2 - \alpha \beta_1 y_1 z_2) \end{pmatrix} \\
\mathbf{end while}
\end{array}$$

We note that the model in [29] uses truncated Normal distributions, as [29] is limited to finite supports for the program variables, which is not the case with our work.

In the loop above, each of the variables exhibit non-linear self-dependence, and so the variables are partitioned into $D(\mathcal{P}) = \{x, y_1, y_2, z_1, z_2\}$ and $E(\mathcal{P}) = \emptyset$. While the recurrence operator of the loop above is unsolvable, our approach infers polynomial loop invariants using defective variable reasoning (Section 5). Namely, we generate the following closed-form solutions over expected values of program variables:

$$\begin{aligned}
\mathbb{E}(x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n)) &= 1045, \\
\mathbb{E}((x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))^2) &= 3277349/3, \quad \text{and} \\
\mathbb{E}((x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))^3) &= 1142497455.
\end{aligned}$$

One can interpret such invariants in terms of the biological assumptions in the model. Take, for example, the fact that $\mathbb{E}(x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))$ is constant. This invariant is in line with the assumption in the model that total population of the swarm is constant. In fact, our invariants reflect the behaviour of the system in the original *continuous-time* model proposed in [3], because our approach is able to process all coefficients (most importantly Δ) as symbolic constants.

Example 8 (Probabilistic Transition Systems [30]). Consider the following probabilistic loop modelling a *probabilistic transition system* from [30]:

$$\begin{array}{l}
\mathbf{while} \star \mathbf{do} \\
\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} \text{Normal}(0, 1) \\ \text{Normal}(0, 1) \end{pmatrix} \\
\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + a x y \\ y + b x y \end{pmatrix} \\
\mathbf{end while}
\end{array}$$

While [30] uses martingale theory to synthesise a degree one invariant of the form $a\mathbb{E}(x_k) + b\mathbb{E}(y_k) = a\mathbb{E}(x_0) + b\mathbb{E}(y_0)$, our work automatically generates invariants over higher-order moments involving the defective variables x and y , as presented in Table 2.

We conclude this section with an unsolvable loop whose recurrence operator cannot (yet) be handled by our work.

Example 9 (Trigonometric Updates). As our approach is limited to polynomial updates of the program variables, the loop below cannot be handled by our work:

```

while * do
   $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \cos(x) \\ \sin(x) \end{pmatrix}$ 
end while

```

Note the trigonometric functions are transcendental, from which it follows that one cannot generally obtain closed-form solutions for the program variables. Nevertheless, this program does admit polynomial invariants in the program variables; for example, $x^2 + y^2 = 1$. Although our definition of a defective variables does not apply here, we could say the variable x here is *somehow defective*: while the exact value of $\sin(x)$ cannot be computed, it could be approximated using power series. Extending our work with more general notions of defective variables is an interesting line for future work.

7 Experiments

In this section we report on our implementation towards fully automating the analysis of unsolvable loops, and report on our experimental setting and results.

Implementation. Algorithm 1 together with our method for synthesising invariants involving defective variables is implemented in the tool `Polar`³. We use `python3` and the `sympy` package [23] for symbolic manipulations of algebraic expressions.

Benchmark Selection. While previous works [28,27,26,13,1,20] consider invariant synthesis, their techniques are only applicable in a restricted setting: the analysed loops are, for the most part, solvable; or, for unsolvable loops, the search for polynomial invariants is template-driven or employs heuristics. In contrast, the work herein complements and extends the techniques presented for solvable loops in [28,27,26,13,1,20]. Indeed, our automated approach turns the problem of polynomial invariant synthesis into a decidable problem for a larger class of unsolvable loops.

While solvable loops can clearly be analysed by our work, the main benefit of our work comes with handling unsolvable loops by translating them into solvable ones. For this reason, in our experimentation we are not interested in examples of solvable loops and so only focus on unsolvable loop benchmarks. There is therefore no sensible baseline that we can compare against, as state-of-the-art techniques cannot routinely synthesise invariants for unsolvable loops in the generality we present.

³ <https://github.com/probing-lab/polar>

In our work we present a set of 15 examples of unsolvable loops, as listed in Table 1⁴. Common to all 15 benchmarks from Table 1 is the exhibition of circular non-linear dependencies within the variable assignments. We display features of our benchmarks in Table 1 (for example, column 3 of Table 1 counts the number of defective variables for each benchmark).

Three examples from Table 1 are challenging benchmarks taken from the invariant generation literature [4,5,30,29]; full automation in analysing these examples was not yet possible. These examples are listed as `non-lin-markov-1`, `pts`, and `bees` in Table 1, respectively corresponding to Example 2 (and hence Example 6), Example 8, and Example 7 from Section 6.

The remaining 12 examples of Table 1 are self-constructed benchmarks to highlight the key ingredients of our work in synthesising invariants associated with unsolvable recurrence operators. The benchmarks that do not appear in the main text are listed in Appendix A.

Experimental Setup. We evaluate our approach in `Polar` on the examples from Table 1. All our experiments were performed on a machine with a 1.80GHz Intel i7 processor and 16 GB of RAM.

Evaluation Setting. The landscape of benchmarks in the invariant synthesis literature for solvable loops can appear complex with respect to a large numbers of variables, high degrees in polynomial updates, and multiple update options. However, we do not intend to compete on these metrics for solvable loops. The power of Algorithm 1 lies in its ability to handle ‘unsolvable’ loop programs: those with cyclic inter-dependencies and non-linear self-dependencies in the loop body. While the benchmarks of Table 1 may be considered simple, the fact that previous works cannot systematically handle such *simple models* crystallises that even simple loops can be unsolvable, limiting the applicability of state-of-the-art methods, as illustrated in the example below.

Example 10. Consider the question: *does the unsolvable loop program `deg-9` in Table 1 (i.e. Example 15) possess an invariant of degree 3?* The program variables for `deg-9` are x, y , and z . The variables x and y are defective. Using `Polar`, we derive that the cubic, non-trivial polynomial $p(x_n, y_n, z_n)$ given by

$$12(ay_n + by_n^2 + cy_n^3 + dx_n + ex_ny_n + fx_ny_n^2) - (3a + 24b + 117c + 2d + 17e + 26f)x_n^2 - (6a - 6b + 315c + 4d - 2e + 88f)x_n^2y_n + 3(3a - 3b + 144c + 2d - e + 35f)x_n^3$$

yields a polynomial loop invariant of degree 3, where a, b, c, d, e , and f are symbolic constants. Moreover, for $n \geq 1$, the expectation of this polynomial (`deg-9` is a probabilistic loop) in the n th iteration is given by

$$\mathbb{E}(p(x_n, y_n, z_n)) = -108a + 312b - 1962c - 68d + 52e - 68f.$$

⁴ each benchmark in Table 1 references, in parentheses, the respective example from our paper

BENCHMARK	VAR	DEF	TERM	DEG	CAND-7	EQN-7
squares (Fig. 1a)	3	2	8	2	35	113
squares+ (Ex.11)	4	2	12	2	35	204
non-lin-markov-1 (Ex. 2)	2	2	11	2	35	64
non-lin-markov-2 (Ex. 12)	2	2	11	2	35	64
prob-squares (Ex.13)	4	3	13	2	119	-
squares-and-cube (Fig.1b)	3	3	4	3	119	337
pts (Ex. 8)	4	2	6	3	35	57
squares-squared (Fig. 5)	4	4	15	4	329	-
bees (Ex. 7)	5	5	21	5	791	-
deg-5 (Ex. 15)	3	2	8	5	35	42
deg-6 (Ex. 15)	3	2	8	6	35	42
deg-7 (Ex. 15)	3	2	8	7	35	42
deg-8 (Ex. 15)	3	2	8	8	35	43
deg-9 (Ex. 15)	3	2	8	9	35	43
deg-500 (Ex. 15)	3	2	8	500	35	43

Table 1: Features of the benchmarks. VAR = Total number of loop variables; DEF = Number of defective variables; TERM = Total number of terms in assignments; DEG = Maximum degree in assignments; CAND-7 = Number of monomials in candidate with degree 7; EQN-7 = Size of the system of equations associated with a candidate of degree 7; - = Timeout (60 seconds).

Experimental Results. Our experiments using `PoLar` to synthesise invariants are summarised in Table 2, using the examples of Table 1. Patterns in Table 2 show that, if time considerations are the limiting factor, then the greatest impact cannot be attributed to the number of program variables nor the maximum degree in the program assignments (Table 1). Indeed, time elapsed is not so strongly correlated with either of these program features. As supporting evidence we note the specific attributes of benchmark `deg-500` whose assignments include polynomial updates of large degree and yet returns synthesised invariants with relatively low time elapsed in Table 2. We note the significantly longer running times associated with the benchmark `bees` (Example 7). This suggests that mutual dependencies between program variables in the loop assignment explain this phenomenon: such inter-relations lead to larger systems of equations needed to construct and then resolve the recurrence equation associated with a candidate.

Experimental Summary. Our experiments illustrate the feasibility of synthesising invariants using our approach for programs with unsolvable recurrence operators from various domains such as biological systems, probabilistic loops and classical programs (see Section 5). This further motivates the theoretical characterisation of unsolvable operators in terms of defective variables (Section 4).

BENCHMARK	Candidate Degree						
	1	2	3	4	5	6	7
squares (Fig. 1a)	*1.03	1.22	1.07	2.36	5.34	14.05	39.36
squares+ (Ex. 11)	*0.88	1.06	0.90	2.14	5.89	13.85	32.51
non-lin-markov-1 (Ex. 2)	*0.46	*0.94	*2.25	*3.84	*6.45	*12.29	*21.35
non-lin-markov-2 (Ex. 12)	*0.54	*1.06	*2.35	*4.43	*8.02	*14.07	*24.32
prob-squares (Ex. 13)	*0.80	0.93	4.29	22.50	-	-	-
squares-and-cube (Fig. 1b)	0.31	*0.72	*1.40	*3.11	*7.07	*25.74	-
pts (Ex. 8)	*0.33	*0.55	*0.93	*1.12	*1.78	*2.63	*3.75
squares-squared (Ex. 14)	*0.52	1.75	10.38	-	-	-	-
bees (Ex. 7)	*0.73	*4.80	*53.97	-	-	-	-
deg-5 (Ex. 15)	*0.43	*0.87	*1.83	*4.50	*9.88	*22.81	*45.58
deg-6 (Ex. 15)	*0.41	*0.85	*1.83	*4.39	*10.19	*23.00	*44.29
deg-7 (Ex. 15)	*0.42	*0.85	*1.79	*4.72	*10.04	*25.06	*47.27
deg-8 (Ex. 15)	*0.43	*0.93	*1.89	*4.38	*10.20	*23.91	*49.10
deg-9 (Ex. 15)	*0.43	*0.93	*1.91	*4.49	*10.83	*22.85	*51.97
deg-500 (Ex. 15)	*0.43	*0.85	*1.96	*4.55	*9.75	*23.46	*50.04

- = Timeout (60 seconds); * = Found invariant of the corresponding degree.

Table 2: The time elapsed to automatically synthesise candidates with closed-forms (results in seconds).

8 Conclusion

We establish a new technique that synthesises invariants for loops with unsolvable recurrence operators and show its applicability for deterministic and probabilistic programs. The technique is based on our new characterisation of unsolvable loops in terms of effective and defective variables: the presence of defective variables is equivalent to unsolvability. In order to synthesise invariants, we provide an algorithm to isolate the defective program variables and a new method to compute polynomial combinations of defective variables admitting exponential polynomial closed-forms. The implementation of our approach in the tool `Polar` and our experimental evaluation demonstrate the usefulness of our alternative characterisation of unsolvable loops and the applicability of our invariant synthesis technique to systems from various domains.

References

1. Bartocci, E., Kovács, L., Stankovic, M.: Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In: Proc. of ATVA. pp. 255–276 (2019)
2. Bartocci, E., Kovács, L., Stankovic, M.: Analysis of Bayesian Networks via Prob-Solvable Loops. In: Proc. of ICTAC. pp. 221–241 (2020)
3. Britton, N.F., Franks, N.R., Pratt, S.C., Seeley, T.D.: Deciding on a new home: how do honeybees agree? Proceedings of the Royal Society of London. Series B: Biological Sciences **269**(1498), 1383–1388 (2002)
4. Chakarov, A., Voronin, Y.L., Sankaranarayanan, S.: Deductive Proofs of Almost Sure Persistence and Recurrence Properties. In: Proc. of TACAS. pp. 260–279 (2016)
5. Dreossi, T., Dang, T., Piazza, C.: Parallelotope Bundles for Polynomial Reachability. In: Proc. of HSCC. p. 297–306 (2016)
6. Elspas, B., Green, M., Levitt, K., Waldinger, R.: Research in Interactive Program-Proving Techniques. Tech. rep., SRI (1972)
7. Everest, G., van der Poorten, A., Shparlinski, I., Ward, T.: Recurrence Sequences, Math. Surveys Monogr., vol. 104. Amer. Math. Soc., Providence, RI (2003)
8. Farzan, A., Kincaid, Z.: Compositional Recurrence Analysis. In: FMCAD. pp. 57–64 (2015)
9. Frohn, F., Hark, M., Giesl, J.: Termination of Polynomial Loops. In: Proc. of SAS. pp. 89–112 (2020)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
11. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: On Strongest Algebraic Program Invariants. J. of ACM (2020), to appear
12. Huang, Z., Fan, C., Mereacre, A., Mitra, S., Kwiatkowska, M.Z.: Invariant Verification of Nonlinear Hybrid Automata Networks of Cardiac Cells. In: Proc. of CAV. pp. 373–390 (2014)
13. Humenberger, A., Jaroschek, M., Kovács, L.: Aligator.jl - A Julia Package for Loop Invariant Generation. In: Proc. of CICM. pp. 111–117 (2018)
14. Humenberger, A., Jaroschek, M., Kovács, L.: Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In: Proc. of ISSAC. pp. 221–228 (2017)
15. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant Generation for Multi-Path Loops with Polynomial Assignments. In: Proc. of VMCAI. pp. 226–246 (2018)
16. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In: Proc. of ESOP. pp. 364–389 (2016)
17. Katz, S., Manna, Z.: Logical Analysis of Programs. Commun. ACM **19**(4), 188–206 (1976)
18. Kauers, M., Paule, P.: The Concrete Tetrahedron. Texts and Monographs in Symbolic Computation (2011)
19. Kauers, M., Zimmermann, B.: Computing the Algebraic Relations of C-finite Sequences and Multisequences. J. Symb. Comput. **43**, 787–803 (2008)
20. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-Linear Reasoning for Invariant Synthesis. In: Proc. of POPL. pp. 54:1–54:33 (2018)
21. Kovács, L.: Reasoning Algebraically About P-Solvable Loops. In: Proc. of TACAS. pp. 249–264 (2008)

22. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of CGO. pp. 75–88 (2004)
23. Meurer, A., Smith, C.P., Paprocki, M., Čertík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., Rathnayake, T., Vig, S., Granger, B.E., Muller, R.P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M.J., Terrel, A.R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.: SymPy: Symbolic Computing in Python. *PeerJ Computer Science* **3**, e103 (2017)
24. Moosbrugger, M., Stankovic, M., Bartocci, E., Kovács, L.: This is the moment for probabilistic loops. *CoRR* **abs/2204.07185** (2022)
25. Müller-Olm, M., Seidl, H.: Computing Polynomial Program Invariants. *Information Processing Letters* **91**(5), 233–244 (2004)
26. de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial Invariants by Linear Algebra. In: Proc. of ATVA. pp. 479–494 (2016)
27. Rodríguez-Carbonell, E., Kapur, D.: Generating All Polynomial Invariants in Simple Loops. *J. Symb. Comput.* p. 443–476 (2007)
28. Rodríguez-Carbonell, E., Kapur, D.: Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In: Proc. of ISSAC. p. 266–273 (2004)
29. Sankaranarayanan, S., Chou, Y., Goubault, E., Putot, S.: Reasoning about Uncertainties in Discrete-Time Dynamical Systems using Polynomial Forms. In: Proc. of NeurIPS. pp. 17502–17513 (2020)
30. Schreuder, A., Ong, C.L.: Polynomial Probabilistic Invariants and the Optional Stopping Theorem. *CoRR* **abs/1910.12634** (2019)

A Appendix: Index of Benchmarks

Following are the benchmarks from Tables 1–2 that are not listed in the main text.

Example 11 (squares+).

```

s, x, y, z ← 0, 2, 1, 0
while ★ do
  s ← Bernoulli(1/2)
  z ← z - 1 {1/2} z + 2
  x ← 2x + y2 + s + z
  y ← 2y - y2 + 2s
end while

```

Example 12 (non-lin-markov-2).

```

x, y ← 0, 1
while ★ do
  s ← Bernoulli(1/2)
  if s = 0 then
     $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \frac{4}{10}(x + xy) \\ \frac{4}{10}(13x + \frac{2}{3}y + xy) \end{pmatrix}$ 
  else
     $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \frac{4}{10}(x + y + \frac{2}{3}xy) \\ \frac{4}{10}(2y + \frac{2}{3}xy) \end{pmatrix}$ 
  end if
end while

```

Example 13 (prob-squares).

```

g ← 1
while ★ do
  g ← Uniform(g, 2g)
   $\begin{pmatrix} a \\ b \\ c \end{pmatrix} \leftarrow \begin{pmatrix} a^2 + 2bc - df + b \\ df - a^2 + 2bd + 2c \\ g - bc - bd + \frac{1}{2}a \end{pmatrix}$ 
end while

```

Example 14 (squares-squared).

```

while ★ do
   $\begin{pmatrix} x \\ y \\ z \\ m \end{pmatrix} \leftarrow \begin{pmatrix} xyz + x^2 \\ 2y + z - x^2 + 3ymz^2 \\ \frac{3}{2}x + \frac{3}{2}z + \frac{1}{2}y + \frac{1}{2}x^2 \\ \frac{2}{3}z + 3m - \frac{1}{3}x^2 - \frac{1}{3}xyz - ymz^2 \end{pmatrix}$ 
end while

```


Example 15 (deg-d). The benchmarks `deg-5`, `deg-6`, `deg-7`, `deg-8`, `deg-9`, and `deg-500` are parameterised by the degree d in the following program.

```

 $x, y \leftarrow 1, 1$ 
while  $\star$  do
   $z \leftarrow \text{Normal}(0, 1)$ 
   $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 2x^d + z + z^2 \\ 3x^d + z + z^2 + z^3 \end{pmatrix}$ 
end while

```

B Appendix: Adjusting Algorithm 1 for Probabilistic Programs

The works [24,1] defined a recurrence operator for probabilistic loops. Specifically, the recurrence operator is defined for loops with polynomial assignments, probabilistic choice, and drawing from common probability distributions with constant parameters. Recurrences for deterministic loops model the precise values of program variables. For probabilistic loops, this approach is not viable, due to the stochastic nature of the program variables. Thus the recurrence operator for a probabilistic loop models (*higher*) *moments* of program variables. As illustrated in Example 2, the recurrences of a probabilistic loop are taken over expected values of program variable monomials.

In [24,1], the authors explicitly excluded the case of circular non-linear dependencies to guarantee computability. However, in contrast to our notions in Sections 3, they defined variable dependence not on the level of recurrences but on the level of assignments in the loop body. To use the notions of effective and defective variables for probabilistic loops, we follow the same approach and base the dependency graph on assignments rather than recurrences. We illustrate the necessity of this adaptation in the following example.

Example 16. Consider the following probabilistic loop and associated set of first-order recurrence relations in terms of the expectation operator $\mathbb{E}(\cdot)$:

<pre> while \star do $y \leftarrow 4y(1 - y)$ $x \leftarrow x - y \{1/2\} x + y$ end while </pre>	$\mathbb{E}(y_{n+1}) = 4\mathbb{E}(y_n) - 4\mathbb{E}(y_n^2)$ $\mathbb{E}(x_{n+1}) = \mathbb{E}(x_n)$ $\mathbb{E}(x_{n+1}^2) = \mathbb{E}(x_n^2) + \mathbb{E}(y_{n+1}^2)$
---	---

It is straightforward to see that variable y is defective from the deterministic update $y \leftarrow 4y(1 - y)$ with its characteristic non-linear self-dependence. Moreover, y appears in the probabilistic assignment of x . However, due to the particular form of the assignment, the recurrence of $\mathbb{E}(x_n)$ does not contain y . Nevertheless, y appears in the recurrence of $\mathbb{E}(x_n^2)$. This phenomenon is specific to the probabilistic setting. For deterministic loops, it is always the case that if the values of a program variable w do not depend on defective variables, then neither do the values of any power of w .

In light of the phenomenon exhibited in Example 16, for probabilistic loops, we adapt our notion of *variable dependency*. Without loss of generality, we assume that every program variable has exactly one assignment in the loop body. Let \mathcal{P} be a probabilistic loop and $x, y \in \text{Vars}(\mathcal{P})$. We say x *depends on* y , if y appears in the assignment of x . Additionally, the dependency is *linear* if all occurrences of y in the assignment of x are linear, else the dependency is *non-linear*. Further, we consider the transitive closure of variable dependency analogous to deterministic loops and Definition 3.

With variable dependency thus defined, the dependency graph and the notions of effective and defective variables follow immediately. Analogous to our characterisation of unsolvable recurrence operators in terms of defective variables for deterministic loops, *all (higher) moments* of effective variables of probabilistic loops can be described by a system of linear recurrences [24,1]. For defective variables this property will generally fail. For instance, in Example 16, the variable x is now classified as defective and $\mathbb{E}(x_n^2)$ cannot be modelled by linear recurrences for some initial values.

The only necessary change to the invariant synthesis algorithm from Section 5 is that instead of program variable monomials, we consider expected values of program variable monomials. Now, our synthesis technique from Section 5 can also be applied to probabilistic loops to synthesise combinations of expected values of defective variable monomials that do satisfy a linear recurrence.