Amrollahi, D, Bartocci, E, Kenison, G, Kovacs, L, Moosbrugger, M and Stankovic, M

(Un)Solvable loop analysis

https://researchonline.ljmu.ac.uk/id/eprint/24655/

Article

Formal Methods in System Design (2025) 65:163–194
https://doi.org/10.1007/s10703-024-00455-0

RESEARCH

(Un)Solvable loop analysis

Daneshvar Amrollahi[1] · Ezio Bartocci[1] · George Kenison[1] · Laura Kovács[1] · Marcel Moosbrugger[1] · Miroslav Stankovič[1]

Received: 25 May 2023 / Accepted: 13 May 2024 / Published online: 11 June 2024

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 Rodríguez-Carbonell and Kapur (in: Proceedings of the ISSAC, pp 266–273, 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. Herein we consider the following two applications. First, we present a novel technique that automatically synthesises polynomials from defective monomials, that admit closed-form solutions and thus lead to polynomial loop invariants. Second, given an unsolvable loop, we synthesise solvable loops with the following property: the invariant polynomials of the solvable loops are all invariants of the given unsolvable loop. Our implementation and experiments demonstrate both the feasibility and applicability of our approach to both deterministic and probabilistic programs.

Keywords Invariant generation · Solvable loop synthesis · Algebraic recurrences · Verification · Solvable operators

✉ George Kenison
george.kenison@tuwien.ac.at

Daneshvar Amrollahi
daneshvar.amrollahi@tuwien.ac.at

Ezio Bartocci
ezio.bartocci@tuwien.ac.at

Laura Kovács
laura.kovacs@tuwien.ac.at

Marcel Moosbrugger
marcel.moosbrugger@tuwien.ac.at

Miroslav Stankovič
miroslav.stankovic@tuwien.ac.at

[1] TU Wien, Vienna, Austria

Extension of previous work

This paper is an extended version of the conference paper 'Solving invariant generation for unsolvable loops' published at SAS 2022 [1]. We extended the text and results from the conference paper.

In addition, this paper brings the following two new contributions when compared to our SAS 2022 paper [1]. First, we introduce an algorithmic approach (Algorithm 2) that synthesises solvable loops from unsolvable loops, such that the polynomial invariants of the solvable loop are also invariants of the unsolvable one. Second, we expand the list of unsolvable loops (with new Examples 36–43), drawn from the Mathematics and Physics literature; our benchmarks yield a new test suite for evaluating loop analysis techniques and demonstrate the feasibility of our approach.

1 Introduction

1.1 Background and motivation

With substantial progress in computer-aided program analysis and automated reasoning, several techniques have emerged to automatically synthesise 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. This problem remains unsolved even when we restrict consideration to loops that are non-nested, without conditionals, and/or without exit conditions. Our work improves the state of the art under such and similar considerations.

Loop invariants, in the sequel simply *invariants*, are properties that hold before and after every iteration of a loop. Invariants therefore provide the key inductive arguments for automating the verification of programs; for example, proving correctness of deterministic loops [2–6] and correctness of hybrid and probabilistic loops [7–9], or data flow analysis and compiler optimisation [10]. 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* [2], such as loops with (blocks of) affine assignments [3–6].

A common approach for constructing polynomial invariants, first pioneered in [12, 13], 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 polynomial invariants resulting from step (iii) over-approximate the fixed point of the loop. 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 nth 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 [14, 15]. Exploiting this result, the class of recurrence operators that can be linearised are called *solvable* [2]. Intuitively, a loop with a recurrence operator is solvable only if the non-

```
  $z \leftarrow 0$
while $\star$ do
  $z \leftarrow 1 - z$
  $x \leftarrow 2x + y^2 + z$
  $y \leftarrow 2y - y^2 + 2z$
end while
```

Closed-form of $x + y$:
$x(n) + y(n) = 2^n(x(0) + y(0) + 2) - (-1)^n/2 - 3/2$

(a) The program $\mathcal{P}_\square$.

```
$x, y \leftarrow 1, 1$
while $\star$ do
  $w \leftarrow x + y$
  $x \leftarrow w^2$
  $y \leftarrow w^3$
end while
```

Polynomial Invariant:
$y^2(n) - x^3(n) = 0$

(b) The program $\mathcal{P}_{\text{SC}}$.

Fig. 1 Two running examples with unsolvable recurrence operators. Nevertheless, $\mathcal{P}_\square$ admits a closed-form for combinations of variables and $\mathcal{P}_{\text{SC}}$ admits a polynomial invariant. Herein we use $\star$ (rather than a loop guard or `true`) as loop termination is not our focus. For the avoidance of doubt, in this paper we consider standard mathematical arithmetic (e.g. mathematical integers) rather than machine floating-point and finite precision arithmetic

linear dependencies in the resulting system of polynomial recurrences are acyclic (see Sect. 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 Fig. 1 whose recurrence operators are not solvable (i.e. unsolvable). In other works, the generation of polynomial invariants is usually limited to those variables that admit closed-forms. In our work, specifically step (iii), we can generate polynomial invariants from *combinations of program variables* that admit closed-forms (where individual variables may fail to do so). This analysis can lead to a tighter over-approximation of a loop's fixed point. In general, the main obstacle in the setting of unsolvable recurrence operators is the absence of "well-behaved" closed-forms for the resulting recurrences.

1.2 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 [2, 16] 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 [3, 6] 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 [5, 17] 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 Sect. 2). All the aforementioned approaches are mainly restricted to C-finite recurrences for which closed-forms always exist, thus yielding loop invariants. In [9, 18] the authors establish techniques to apply invariant synthesis techniques developed for deterministic loops to probabilistic programs. 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 [5, 19]. The approach of [5] extracts C-finite recurrences over linear combinations of loops variables from unsolvable loops. For example, the method presented in [5] can also synthesise the closed-forms identified by our work for Fig. 1a. However, unlike [5], our work is not limited to linear combinations (we can extract C-finite recurrences over *polynomial* relations in the loop variables). As such, the technique of [5] cannot synthesise the polynomial loop invariant in Fig. 1b, whereas our work can. A further related approach to our work is given in [19], 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 Fig. 1.

Related work in the literature introduces techniques from the theory of martingales in order to synthesise invariants in the setting of probabilistic programs [20]. Therein, the programming model is represented by a class of loop programs where all updates are linear and the synthesised invariants are given by linear templates. By contrast, our method allows us to handle polynomial arithmetic; in particular, we automatically generate invariants given by monomials in the program variables. On the other hand, the approach of [20] can also synthesise supermartingales whereas our work is restricted to invariants defined by equalities.

1.3 Our contributions

In this paper we consider the sister problems of invariant generation and solvable loop 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 summarised below.

1. Crucial for our synthesis technique is our novel characterisation of unsolvable recurrence operators in terms of defective variables (Theorem 14). 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., Fig. 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 linear relations in defective monomials such that these relations admit closed-forms, from which polynomial loop invariants follow (Sect. 5).
4. Given an unsolvable loop, we introduce an algorithmic approach (Algorithm 2) that synthesises a solvable loop with the following property: every polynomial invariant of the solvable loop is also an invariant of the given unsolvable loop (Sect. 6).
5. We generalise our work to the analysis of probabilistic program loops (Sect. 7) and showcase further applications of unsolvable operators in such programs (Sect. 8).
6. We provide a fully automated approach in the tool `Polar`.[1] For evaluating our work, we compiled an extensive lists of challenging loops from the literature, including applications of mathematical and physical modelling. 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 (Sect. 9).

[1] https://github.com/probing-lab/polar.

1.4 Beyond invariant generation

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 [21]. 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 linear combinations of defective monomials *do* satisfy linear recurrences, which opens avenues where we can apply scalar evolution techniques over such monomials. In particular, our work automatically computes polynomial combinations of some defective loop variables, which potentially enlarges the class of loops that, for example, LLVM can optimise.

1.5 Structure and summary of results

The rest of this paper is organised as follows. We briefly recall preliminary material in Sect. 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 14). In Sect. 5 we present our new technique that synthesises linear relations in defective monomials that admit well-behaved closed-forms. In Sect. 6 we introduce Algorithm 2 to synthesise solvable loops. That is, given an unsolvable loop, Algorithm 2 outputs a solvable loop, if it exists such that each polynomial invariant of the solvable loop is also an invariant of the unsolvable loop. In Sect. 7 we detail the necessary changes to the algorithms in Sects. 5 and 6 for probabilistic programs. We illustrate our approach with several case-studies in Sect. 8, and describe a fully-automated tool support of our work in Sect. 9. We also report on accompanying experimental evaluation in Sects. 8–9, and conclude the paper in Sect. 10.

2 Preliminaries

2.1 Notation

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 algebraic closure of $\mathbb{Q}$, to denote the field of algebraic numbers. We write $\mathbb{R}[x_1, \ldots, x_k]$ and $\overline{\mathbb{Q}}[x_1, \ldots, x_k]$ for the polynomial rings of all polynomials $P(x_1, \ldots, x_k)$ in k variables $x_1, \ldots, x_k$ 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 Fig. 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 $\mathcal{P}_{\text{MC}}$ in Example 5).

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

[2] https://llvm.org/docs/Passes.html.

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.

2.2 C-finite recurrences

We recall relevant results on (algebraic) recurrences and refer to [14, 15] for further details. A *sequence* in $\overline{\mathbb{Q}}$ is a function $u\colon \mathbb{N} \to \overline{\mathbb{Q}}$, 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 $u(n+\ell) = \mathrm{Rec}(u(n+\ell-1), \dots, u(n+1), u(n), n)$, for some function $\mathrm{Rec}\colon \mathbb{R}^{\ell+1} \to \mathbb{R}$. 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

$$u(n+\ell) = a_{\ell-1}u(n+\ell-1) + a_{\ell-2}u(n+\ell-2) + \cdots + a_0 u(n) \tag{1}$$

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

$$x^{n+\ell} - a_{\ell-1}x^{n+\ell-1} - a_{\ell-2}x^{n+\ell-2} - \cdots - a_0 x^n.$$

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 [14, 15].

2.3 Invariants

A loop invariant is a loop property that holds before and after each loop iteration [22]. In this paper, 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 example.

Example 1

```
$x, y, z \leftarrow 0, 1, 0$
 while $\star$ do
   $x \leftarrow 1$
   $y \leftarrow y + x$
   $z \leftarrow z + 1$
 end while
```

The loop admits the polynomial invariant $y - z - 1 = 0$ given by the initial values $x(0) = 0$, $y(0) = 1$, $z(0) = 0$ and the closed-forms $x(n) = 1$, $y(n) = n + 1$, and $z(n) = n$. For each $n \geq 1$, we denote by $v(n)$ the value of a loop variable v 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 $n \geq 1$.

2.4 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

$$I(S) := \{s_1 q_1 + \cdots + s_\ell q_\ell \mid s_i \in S, q_i \in \overline{\mathbb{Q}}[x_1, \dots, x_k], \ell \in \mathbb{N}\}.$$

Let $\mathcal{P}$ be a program as before. For $x_j \in \mathrm{Vars}(\mathcal{P})$, let $\langle x_j(n) \rangle_n$ denote the sequence whose nth term is given by the value of x_j in the nth loop iteration. The set of polynomial invariants of $\mathcal{P}$ form an ideal, the *invariant ideal* of $\mathcal{P}$ [16]. 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 [23], 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 [23].

3 From loops to recurrences

3.1 Recurrence operators

Modelling properties of loop variables by algebraic recurrences and solving the resulting recurrences is an established approach in program analysis. Multiple works [3, 5, 6, 17, 24] associate a loop variable x with a sequence $\langle x(n) \rangle_n$ whose nth term is given by the value of x in the nth 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 [2]. To this end, over-approximation methods for general loops are proposed in [5, 17] 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 the characterisation of both *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 nth loop iteration [2, 3, 17] or statistical moments in probabilistic loops [9].

Definition 2 (Recurrence operator) A *recurrence operator* $\mathcal{R}$ maps the program variables $\text{Vars}(\mathcal{P})$ 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 3 Consider the program $\mathcal{P}_{\text{SC}}$ in Fig. 1b. One can employ a recurrence operator $\mathcal{R}$ in order to capture the values of the program variables in the nth 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 Fig. 1a, we obtain the following system of recurrences:

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

3.2 Solvable operators

Systems of linear recurrences with constant coefficients admit computable closed-form solutions as exponential polynomials [14, 15]. This property holds for a larger class of recurrences with polynomial updates, which leads to the notion of *solvability* introduced in [2]. 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 4 (Solvable operators [2, 4]) The recurrence operator $\mathcal{R}$ is *solvable* if there exists a partition of Vars_n; that is, $\text{Vars}_n = W_1 \uplus \cdots \uplus W_k$ 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 [2] (see the discussion in [4]).

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 [9], 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 5 Consider the probabilistic program $\mathcal{P}_{\text{MC}}$ of [25, 26] 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
```

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

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

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 8). 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 Sect. 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 Sect. 7.

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 nth 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 6 (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.

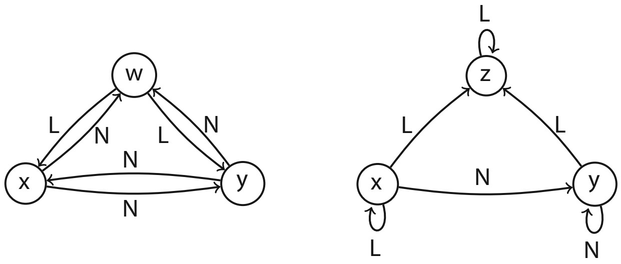

Fig. 2 The dependency graphs for $\mathcal{P}_{\mathrm{SC}}$ and $\mathcal{P}_{\square}$ from Fig. 1

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 7 (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 $G = (\mathrm{Vars}(\mathcal{P}), A, \mathcal{L})$ with vertex set $\mathrm{Vars}(\mathcal{P})$, edge set $A := \{(x, y) \mid x, y \in \mathrm{Vars}(\mathcal{P}) \wedge x \text{ depends on } y\}$, and a function $\mathcal{L}\colon A \to \{L, N\}$ that assigns a unique *label* to each edge such that

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

Given a program and a recurrence operator, its dependency graph can be constructed automatically with standard techniques. In our approach, we partition the variables $\mathrm{Vars}(\mathcal{P})$ 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 8 (Effective–defective) A variable $x \in \mathrm{Vars}(\mathcal{P})$ 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 9 From the recurrence equations of Example 3 for the program $\mathcal{P}_{\mathrm{SC}}$ (see Fig. 1b), one obtains the dependencies between the program variables of $\mathcal{P}_{\mathrm{SC}}$: 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 $E(\mathcal{P}_{\mathrm{SC}}) = \emptyset$ and $D(\mathcal{P}_{\mathrm{SC}}) = \{w, x, y\}$.

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

We give the following straightforward corollary of Definition 8.

Corollary 10 *Given any effective variable $x \in E(\mathcal{P})$, the recurrence relation $\mathcal{R}[x]$ is a polynomial in effective variables.*

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 14 and automated in Algorithm 1). We formalise and prove this results via the following three lemmas.

Lemma 11 *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 8, 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 $W_1, \ldots, W_k$ of $\text{Vars}_n(\mathcal{P})$ as described in Definition 4. Moreover, since C is a cycle, there exists $j \in \{1, \ldots, k\}$ 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 $\mathcal{R}[w] = M_j \cdot W_j^\top + P_j(W_1, \ldots, W_{j-1})$ (by Definition 4), 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 $W_1, \ldots, W_k$ is a partition of $\text{Vars}_n(\mathcal{P})$. Hence $\mathcal{R}$ is unsolvable. □

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 12 *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. □

Lemma 13 *Let $\mathcal{P}$ be a program with recurrence operator $\mathcal{R}$. If each of the program variables of $\mathcal{P}$ is effective then $\mathcal{R}$ is solvable.*

Proof By Lemma 12, 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 $\Pi = \{\pi_1, \ldots, \pi_{|\Pi|}\}$ such

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

```
Construct the dependency graph $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$ 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)
    $D(\mathcal{P}) \leftarrow D(\mathcal{P}) \cup$ predecessor
  end if
  if $x \neq y$ then
    predecessor $\leftarrow \emptyset$
    DFS($y$, predecessor)
    if $x \in$ predecessor then
      $D(\mathcal{P}) \leftarrow D(\mathcal{P}) \cup$ predecessor
    end if
  end if
end for
$E(\mathcal{P}) \leftarrow \text{Vars}(\mathcal{P}) \setminus D(\mathcal{P})$
```

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 4. We thus conclude that $\mathcal{R}$ is solvable. □

Together, Lemmas 11–13 yield a new characterisation of unsolvable operators.

Theorem 14 (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 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 14, 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 Sect. 5.

Remark 15 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 monomials (Sect. 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, linear combinations of defective monomials 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 $d \in \mathbb{N}$:

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

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

Example 16 For $\mathcal{P}_\square$ in Fig. 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

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

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., [15]. 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 17 Observe that the sum on the right-hand side of Eq. (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 $\mathrm{Vars}_n(\mathcal{P})$ with unknown coefficients c_M, c_W, and κ. An alternative expression for $S(n{+}1)$ in $\mathrm{Vars}_n(\mathcal{P})$

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 $\mathrm{Vars}_n(\mathcal{P})$. 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. The main complexity of our invariant synthesis technique lies in solving the quadratic system. In the candidate polynomial, every monomial in defective variables (of degree at most d) is associated with a unique unknown coefficient. Hence, the size of the quadratic system can be polynomial in d and exponential in the number of defective variables.

Example 18 Consider the following illustration of our invariant synthesis procedure. Recall program $\mathcal{P}_\Box$ from Fig. 1a:

```
$z \leftarrow 0$
while $\star$ do
    $z \leftarrow 1 - z$
    $x \leftarrow 2x + y^2 + z$
    $y \leftarrow 2y - y^2 + 2z$
end while
```

From Algorithm 1 we obtain $E(\mathcal{P}_\Box) = \{z\}$ and $D(\mathcal{P}_\Box) = \{x, y\}$. Because $D(\mathcal{P}_\Box) \neq \emptyset$, we deduce using Theorem 14 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 Fig. 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. It is not guaranteed that a solution does exist. In that case, our technique can 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.

Example 19 The following loop models the *logistic map* [27] which is well-known for its chaotic behaviour.

```
while $\star$ do
    $x \leftarrow rx(1 - x)$
end while
```

The single variable x is defective due to its non-linear self-dependency. For most values of r and initial values of x the logistic map does not admit an analytical solution and well-behaved polynomials in x do not exist [28]. Hence, our invariant synthesis technique provides no solution for candidates of fixed degrees.

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 Sect. 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., [2, 3].

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

6 Synthesising solvable loops from unsolvable loops

In previous sections, we introduced a new technique to compute invariants for unsolvable loops; that is, loops containing defective variables. An orthogonal challenge is to synthesise a solvable loop from an unsolvable loop that preserves or over-approximates given specifications.

In this section we establish, with Algorithm 2, a new method to synthesise a solvable loop $\mathcal{P}'$ from an unsolvable loop $\mathcal{P}$. The solvable loop $\mathcal{P}'$ over-approximates the behaviour of $\mathcal{P}$ in the sense that every polynomial invariant of $\mathcal{P}'$ is an invariant of $\mathcal{P}$. Moreover, we show the invariants among effective variables of $\mathcal{P}$ and $\mathcal{P}'$ coincide. The following example illustrates the main idea leading to Algorithm 2.

Example 20 Example 18 showed how to synthesise the polynomial $S = x + y$ of defective variables x and y for the loop in Fig. 1a such that S admits a closed-form. In this case, the polynomial of program variables S satisfies the linear inhomogeneous recurrence $S(n+1) = 2\,S(n) - 3z(n) + 3$. We can use this recurrence to construct a solvable loop from the unsolvable from Fig. 1a that captures the dynamics of the only effective variable z as well as S:

```
$z \leftarrow 0$
while $\star$ do
    $z \leftarrow 1 - z$
    $x \leftarrow 2x + y^2 + z$
    $y \leftarrow 2y - y^2 + 2z$
end while
```

$$\xmapsto{\text{Algorithm 2}}$$

```
$z \leftarrow 0$
$\mathtt{s} \leftarrow x_0 + y_0$
while $\star$ do
    $\begin{pmatrix} z \\ \mathtt{s} \end{pmatrix} \leftarrow \begin{pmatrix} 1 - z \\ 2\mathtt{s} - 3z + 3 \end{pmatrix}$
end while
```

Our algorithmic approach, as given in Algorithm 2, synthesises the solvable loop from the unsolvable loop on the left. Such a synthesis step is implemented within our tool (Sect. 9), allowing us to synthesize the above solvable loop in about 1 second.

The inputs for Algorithm 2 are an unsolvable loop $\mathcal{P}$ with recurrence operator $\mathcal{R}$ and a fixed degree $d \in \mathbb{N}$; the algorithm outputs a solvable loop $\mathcal{P}'$, if it exist. Briefly, the algorithm invokes the invariant synthesis procedure from Sect. 2 (for degree d) and constructs the loop $\mathcal{P}'$ from $\mathcal{P}$ by removing all the defective variables and then introducing a new variable $\mathtt{s}$ that models an invariant among defective monomials, if such an invariant exists. The recurrence operator $\mathcal{R}'$ associated with the synthesised loop $\mathcal{P}'$ is the canonical recurrence operator: $\mathcal{R}'$ maps every program variable to its assignment.

Lemma 21 (Soundness) *The loop $\mathcal{P}'$ returned by Algorithm 2 is solvable. Moreover, we have $E(\mathcal{P}) \subseteq \mathrm{Vars}(\mathcal{P}')$.*

Proof Using our characterisation of solvable and unsolvable loops in terms of effective and defective variables (Theorem 14), we show that all variables in $\mathcal{P}'$ are effective. The variables of $\mathcal{P}'$ consist of the effective variables of $\mathcal{P}$ as well as the fresh variable $\mathtt{s}$: $\mathrm{Vars}(\mathcal{P}') = E(\mathcal{P}) \cup \{\mathtt{s}\}$, or $\mathrm{Vars}(\mathcal{P}') = E(\mathcal{P})$ if no invariant among defective monomials with degree d exists.

First, every $x \in E(\mathcal{P})$ necessarily remains effective for the synthesised program $\mathcal{P}'$: in the dependency graph of $\mathcal{P}'$, the variable x cannot occur in a cycle containing $\mathtt{s}$, because $\mathtt{s}$ is a fresh variable and hence cannot appear in the recurrence $\mathcal{R}[y]$ for any $y \in E(\mathcal{P})$ (Algorithm 2 line 15). Hence, if $z \in E(\mathcal{P}) \cap D(\mathcal{P}')$, then there must exist a cycle in the dependency graph of $\mathcal{P}'$ with a non-linear edge (x, y) (i.e., $\mathcal{L}(x, y) = N$) such that every vertex in the cycle is in $E(\mathcal{P})$. Consequently, this cycle is also present in the dependency graph of the original program $\mathcal{P}$. This means that not all variables in $E(\mathcal{P})$ are effective, which is a contradiction.

Algorithm 2 Solvable Loop Synthesis

```
Input: Unsolvable loop $\mathcal{P}$ with recurrence operator $\mathcal{R}$, degree $d \in \mathbb{N}$
Output: Solvable loop $\mathcal{P}'$
  Compute $E(\mathcal{P})$ and $D(\mathcal{P})$ using Algorithm 1.
  Fix candidate polynomial $S(n)$ of degree $d$ (as in Section 5 (2)).
  Solve for coefficients in
    $S(n+1) = \kappa S(n) + \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M$ (as in Section 5 (3)).
  initial = []
  body_left = []
  body_right = []
  ▷ Add all effective variables to new loop
  for $x \in E(\mathcal{P})$ do
      initial.append($x \leftarrow x_0$)
      body_left.append($x$)
      body_right.append($\mathcal{R}[x]$)
  end for
  ▷ Add well-behaved combination of defective monomials
  if $S \neq 0$ then
      Choose a fresh symbol s.
      initial.append(s $\leftarrow S(0)$)
      body_left.append(s)
      body_right.append($\kappa$s $+ \sum_{M \in \mathcal{E}(\mathcal{P})} c_M M$)
  end if
  $\mathcal{P}' \leftarrow$ initial "while ⋆ do" body_left $\leftarrow$ body_right "end while"
  return $\mathcal{P}'$
```

Second, the variable s is effective. Because s is a fresh variable, the only incoming edge of s in the dependency graph of $\mathcal{P}'$ is a linear self-loop (Algorithm 2 line 18). Hence s cannot occur in a cycle with a non-linear edge. Moreover, all outgoing edges point to effective variables. Thus s $\in E(\mathcal{P}')$ is effective. □

With the next two lemmas, we prove that the loop $\mathcal{P}'$ synthesised by Algorithm 2 over-approximates the invariants of the unsolvable loop $\mathcal{P}$. Let $\operatorname{Inv}(\mathcal{P})$ and $\operatorname{Inv}(\mathcal{P}')$ denote the invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$ respectively. The next lemma establishes that the synthesised loop $\mathcal{P}'$ is *complete with respect to invariants among effective variables.*

Lemma 22 (Completeness with respect to effective variables) *The invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$ coincide when restricted on the effective variables of $\mathcal{P}$. That is,*

$$\operatorname{Inv}(\mathcal{P}) \cap \overline{\mathbb{Q}}[E(\mathcal{P})] = \operatorname{Inv}(\mathcal{P}') \cap \overline{\mathbb{Q}}[E(\mathcal{P})].$$

Proof By the construction of $\mathcal{P}'$, every $x \in E(\mathcal{P})$ is also a variable of $\mathcal{P}'$. To distinguish the variable x in $\mathcal{P}$ from the variable x in $\mathcal{P}'$ we refer to the latter by x'. We will show that for every $x \in E(\mathcal{P})$ the sequences $\langle x(n) \rangle_n$ and $\langle x'(n) \rangle_n$ coincide. If this is the case, the polynomial invariants for the programs $\mathcal{P}$ and $\mathcal{P}'$ among the variables $E(\mathcal{P})$ necessarily coincide as well.

Let $\mathcal{R}$ and $\mathcal{R}'$ be the recurrence operators associated to $\mathcal{P}$ and $\mathcal{P}'$, respectively. For every $x \in E(\mathcal{P})$ we have $\mathcal{R}[x] = \mathcal{R}'[x']$ and $x_0 = x'_0$ by the construction of $\mathcal{P}'$. Furthermore, by Corollary 10, defective variables cannot occur in $\mathcal{R}[x]$ for any $x \in E(\mathcal{P})$. Moreover, the fresh variable s cannot occur in $\mathcal{R}'[x']$ by the construction of $\mathcal{P}'$. Hence the two systems of first-order recurrences $\{x(n+1) = \mathcal{R}[x] \mid x \in E(\mathcal{P})\}$ and $\{x'(n+1) = \mathcal{R}'[x'] \mid x \in E(\mathcal{P})\}$ together with the initial values $\{x_0 \mid E(\mathcal{P})\}$ and $\{x'_0 \mid E(\mathcal{P})\}$ induce the same sequences $\langle x(n) \rangle_n$ and $\langle x'(n) \rangle_n$ for all $x \in E(\mathcal{P})$. □

We note that when an invariant among defective monomials of degree d does not exist, the variables of the synthesised loop $\mathcal{P}'$ are precisely the effective variables of $\mathcal{P}$. In this case, Lemma 22 fully characterises the relationship between the invariants of $\mathcal{P}$ and $\mathcal{P}'$.

In the following, let us consider the complementary case, namely when an invariant among the defective monomials of $\mathcal{P}$ exists. Hence, the synthesised loop $\mathcal{P}'$ contains the additional fresh variable $\mathtt{s}$ modelling the behaviour of this invariant. With the next lemma, we confirm that the synthesised loop $\mathcal{P}'$ is indeed a *sound over-approximation of the unsolvable loop* $\mathcal{P}$. We show that every invariant of $\mathcal{P}'$ is also an invariant of $\mathcal{P}$. The program variable $\mathtt{s} \in \mathrm{Vars}(\mathcal{P}')$ introduced by Algorithm 2 is however not a program variable of $\mathcal{P}$; nevertheless, $\mathtt{s}$ models the polynomial S of defective variables in $\mathcal{P}$. Hence, to compare the invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$, we need to "substitute" $\mathtt{s}$ by the polynomial of defective variables it models. This can be done by adding the equation $\mathtt{s} = S$ ($\mathtt{s} - S = 0$) to the invariant ideal of $\mathcal{P}'$ and restricting the resulting ideal to $\mathrm{Vars}(\mathcal{P})$:

$$I(\mathrm{Inv}(\mathcal{P}') \cup \{\mathtt{s} - S\}) \cap \overline{\mathbb{Q}}[\mathrm{Vars}(\mathcal{P})]. \tag{4}$$

Lemma 23 (Over-approximation) *Let J be the ideal in* (4) *constructed from the invariant ideal of $\mathcal{P}'$ by replacing the program variable $\mathtt{s}$ by the polynomial of defective variables it models. Then, $J \subseteq \mathrm{Inv}(\mathcal{P})$.*

Proof As argued in the proof of Lemma 22, for every $x \in E(\mathcal{P})$, the sequences corresponding to the variable x in both $\mathcal{P}$ and $\mathcal{P}'$ coincide; that is, $\langle x(n)\rangle_n \equiv \langle x'(n)\rangle_n$. Furthermore, we have $\mathrm{Vars}(\mathcal{P}') = E(\mathcal{P}) \cup \{\mathtt{s}\}$. The fresh variable $\mathtt{s}$ in $\mathcal{P}'$ models the polynomial $S \in \overline{\mathbb{Q}}[\mathrm{Vars}(\mathcal{P})]$. Let $\langle \mathtt{s}(n)\rangle_n$ be the sequence induced by the program variable $\mathtt{s}$ in $\mathcal{P}'$ and, likewise, $\langle S(n)\rangle_n$ the sequence induced by the polynomial $S \in \overline{\mathbb{Q}}[\mathrm{Vars}(\mathcal{P})]$. Then, by the construction of $\mathcal{P}'$, we have $\langle \mathtt{s}(n)\rangle_n \equiv \langle S(n)\rangle_n$.

Let $Q \in \mathrm{Inv}(P')$. By the definition of the ideal J in (4), it holds that

$$Q \in \mathrm{Inv}(P') \iff Q\{\mathtt{s} \mapsto S\} \in J$$

(where the notation indicates that S is substituted for $\mathtt{s}$).

Now, Q is a polynomial relation among the sequences induced by the variables in $\mathrm{Vars}(\mathcal{P}')$. Because $\langle x(n)\rangle_n \equiv \langle x'(n)\rangle_n$ for every $x \in E(\mathcal{P})$ and $\langle \mathtt{s}(n)\rangle_n \equiv \langle S(n)\rangle_n$, the polynomial $Q\{\mathtt{s} \mapsto S\}$ represents a relation among the sequences induced by the variables in $\mathrm{Vars}(\mathcal{P})$. Hence we have $Q \in \mathrm{Inv}(\mathcal{P})$. □

Remark 24 Our loop synthesis procedure given in Algorithm 2 computes a single invariant among defective monomials (if such an invariant exists) of an unsolvable loop $\mathcal{P}$. The results in this section naturally generalise to multiple invariants among defective monomials, as follows: for every invariant I, add a fresh variable $\mathtt{s}_I$ modelling the behaviour of I to the synthesised program $\mathcal{P}'$. Note that with each additional invariant I added, the dynamics of the synthesised loop $\mathcal{P}'$ more closely resembles that of the unsolvable loop $\mathcal{P}$.

7 Adjustments for unsolvable operators in probabilistic programs

7.1 Defective variables in probabilistic loop models

The works [9, 29] defined recurrence operators for probabilistic loops. Specifically, a 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 a recurrence operator for a probabilistic loop models *(higher) moments* of program variables. As illustrated in Example 5, the recurrences of a probabilistic loop are taken over expected values of program variable monomials.

The authors of [9, 29] explicitly excluded the case of circular non-linear dependencies to guarantee computability. However, in contrast to our notions in Sect. 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 then recurrences. We illustrate the necessity of this adaptation in the following example.

Example 25 A probabilistic assignment $x \leftarrow a \; \{p\} \; b$ intuitively means that x is assigned a with probability p and b with probability $1-p$. Consider the following probabilistic loop and associated set of first-order recurrence relations in terms of the expectation operator $\mathbb{E}(\,\cdot\,)$.

```
while $\star$ do
    $y \leftarrow 4y(1-y)$
    $x \leftarrow x - y \; \{1/2\} \; x + y$
end while
```

$$
\begin{aligned}
\mathbb{E}(y_{n+1}) &= 4\mathbb{E}(y_n) - 4\mathbb{E}(y_n^2) \\
\mathbb{E}(x_{n+1}) &= \mathbb{E}(x_n) \\
\mathbb{E}(x_{n+1}^2) &= \frac{1}{2}\mathbb{E}((x_n - y_{n+1})^2) + \frac{1}{2}\mathbb{E}((x_n + y_{n+1})^2) \\
&= \mathbb{E}(x_n^2) + \mathbb{E}(y_{n+1}^2)
\end{aligned}
$$

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 25, we adapt our notion of *variable dependency*, for probabilistic loops. 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 $x, y \in \mathrm{Vars}(\mathcal{P})$. 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 6.

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 [9, 29]. For defective variables this property will generally fail For instance, in Example 25, 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 Sect. 5 is as follows: instead of program variable monomials, we consider expected values of program variable monomials. Now, our invariant synthesis technique from Sect. 5 can also be applied to probabilistic loops to synthesise combinations of expected values of defective monomials that do satisfy a linear recurrence.

7.2 Synthesising solvable probabilistic loops

In Algorithm 2 we introduced a procedure, utilising our new invariant synthesis technique from Sect. 5, to over-approximate an unsolvable loop by a solvable loop. The inputs to Algorithm 2 are an unsolvable loop with a recurrence operator and a natural number specifying a fixed degree. As mentioned, our invariant synthesis procedure is also applicable to probabilistic loops using the recurrence operator modelling moments of program variables introduced in [9, 29]. Hence, Algorithm 2 can also be used to synthesise solvable loops from unsolvable probabilistic loops. In the probabilistic case, however, the invariants computed by our approach are over *moments of program variables*. Therefore, the invariant ideal of probabilistic loops describes polynomial relations among a given set of moments of program variables, such as the expected values. Consequently, the loop synthesised by Algorithm 2 for a given probabilistic loop will be deterministic and model the dynamics of moments of program variables of the probabilistic loop.

Example 26 Recall the program $\mathcal{P}_{\mathrm{MC}}$ of Example 5. An invariant synthesised by our approach in Sect. 5 with degree 1 is $\mathbb{E}(x_n - y_n) = \frac{5^n}{6^n}(x_0 - y_0)$. Hence the solvable loop synthesised by Algorithm 2 for $\mathcal{P}_{\mathrm{MC}}$ with input degree 1 is

```
f ← $x_0 + y_0$
while ⋆ do
    $s \leftarrow \frac{1}{2}$
    f ← $\frac{5}{6}$f
end while
```

where f is the fresh variable introduced by Algorithm 2 modelling $\mathbb{E}(x_n - y_n)$. Our approach from Algorithm 2 synthesises this solvable loop from $\mathcal{P}_{\mathrm{MC}}$, using less than 0.5 s within our implementation (Sect. 9).

8 Applications of unsolvable operators towards invariant generation

Our approach automatically generates invariants for programs with defective variables (Sect. 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 for reasoning about solvable loops, complementing thus existing methods, see e.g., [2, 3, 5, 6], in the area. As such, the class of unsolvable loops that can be handled by our work extends (aforementioned) existing approaches on polynomial invariant generation. The experimental results of our approach (see Sect. 9) 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 27 (Moments of probabilistic programs [25]) As mentioned in Example 26, $\mathbb{E}(x_n - y_n) = \frac{5^n}{6^n}(x_0 - y_0)$ is an invariant for the program $\mathcal{P}_{\mathrm{MC}}$ introduced in Example 5. 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 dth moment of $x(n) - y(n)$. While the work in [25] 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 $\mathcal{P}_{\text{MC}}$ are precisely x and y as can be seen from their mutual non-linear interdependence. Namely, we have $D(\mathcal{P}_{\text{MC}}) = \{x, y\}$ and $E(\mathcal{P}_{\text{MC}}) = \{s\}$.

Example 28 (`non-lin-markov-2`) We give a second example of a non-linear Markov chain. We analyse the moments of this probabilistic program in the next section.

```
$x, y \leftarrow 0, 1$
**while** $\star$ **do**
    $s \leftarrow \text{Bernoulli}(1/2)$
    **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 29 (Biological systems [30]) A model for the decision-making process of swarming bees choosing one nest-site from a selection of two is introduced in [30] and further studied in [31, 32]. Previous works have computed probability distributions for this model [32]. The (unsolvable) loop is a discrete-time model with five classes of bees (each represented by a program variable). The coefficient Δ is the length of the time-step in the model and the remaining coefficients parameterise the rates of change. All coefficients here are symbolic.

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

We note that the model in [32] uses truncated Normal distributions, as [32] 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 exhibits non-linear self-dependence, and so the variables are partitioned into $D(\mathcal{P}) = \{x, y_1, y_2, z_1, x_2\}$ 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 (Sect. 5). Namely, we generate the following closed-form solutions over expected values of program variables:

$$\mathbb{E}(x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n)) = 1045,$$

$$\mathbb{E}((x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))^2) = 3{,}277{,}349/3, \quad \text{and}$$

$$\mathbb{E}((x(n)+y_1(n)+y_2(n)+z_1(n)+z_2(n))^3) = 1{,}142{,}497{,}455.$$

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 the 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 [30], because our approach is able to process all coefficients (most importantly Δ) as symbolic constants.

Example 30 (Probabilistic transition systems [25]) Consider the following probabilistic loop modelling a *probabilistic transition system* from [25]:

```
while $\star$ do
    $\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} \text{Normal}(0,1) \\ \text{Normal}(0,1) \end{pmatrix}$
    $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + axy \\ y + bxy \end{pmatrix}$
end while
```

While [25] 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.

The next example demonstrates an unsolvable loop whose recurrence operator cannot (yet) be handled by our work.

Example 31 (Trigonometric updates) As our approach is limited to polynomial updates of the program variables, the loop below cannot be handled by our work:

```
while $\star$ 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.

Examples 32–35 (below) are custom-made benchmarks. We have tailored these benchmarks to demonstrate the flexibility and applicability of our method to the current state of the art. Our experimental analysis is delayed to Sect. 9.

Example 32 (`squares+`)

```
$s, x, y, z \leftarrow 0, 2, 1, 0$
while $\star$ do
    $s \leftarrow \text{Bernoulli}(1/2)$
    $z \leftarrow z - 1 \; \{1/2\} \; z + 2$
    $x \leftarrow 2x + y^2 + s + z$
    $y \leftarrow 2y - y^2 + 2s$
end while
```

Example 33 (`prob-squares`)

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

Example 34 (`squares-squared`)

```
while $\star$ 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 35 (`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
    $z \leftarrow \mathrm{Normal}(0, 1)$
    $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 2x^d + z + z^2 \\ 3x^d + z + z^2 + z^3 \end{pmatrix}$
end while
```

The following set of examples are taken from the literature on the theory of trace maps. Arguably the most famous example is the classical *Fibonacci Trace Map* $f : \mathbb{R}^3 \to \mathbb{R}^3$ given by $f(x, y, z) = (y, z, 2yz - x)$ (Example 36 below); said map has garnered the attention of researchers in fields as diverse as invariant analysis, representation theory, geometry, and mathematical physics (cf. the survey papers [33, 34]). From a computational viewpoint, trace maps arise from substitution rules on matrices (see, again, the aforementioned survey papers). Given two matrices $A, B \in \mathrm{SL}(2, \mathbb{R})$ (the group of 2×2 matrices with unit determinant), consider the following substitution rule on strings of matrices: $A \mapsto B$ and $B \mapsto AB$. The classical Fibonacci Trace Map is determined by the action of this substitution on the traces of the matrices; i.e.,

$$f\big(\mathrm{tr}(A), \mathrm{tr}(B), \mathrm{tr}(AB)\big) = \big(\mathrm{tr}(B), \mathrm{tr}(AB), -\mathrm{tr}(A) + 2\,\mathrm{tr}(B)\,\mathrm{tr}(AB)\big).$$

Further examples of trace maps (Examples 37 and 38 below) are constructed from similar substitution rules on strings of matrices.
Let

$$\begin{aligned} I_1(x, y, z) &= x^2 + y^2 + z^2 - 2xyz, \\ I_2(x, y, z) &= 4x^2y - 2xz - y, \\ I_3(x, y, z) &= x^2 + y^2 + z^2 - xyz - xy - xz - yz - x - y - z, \end{aligned}$$

Then our work generates the cubic polynomial invariants $I_1(x, y, z) - I_1(x_0, y_0, z_0) = 0$, $I_2(x, y, z) - I_2(x_0, y_0, z_0) = 0$, and $I_3(x, y, z) - I_3(x_0, y_0, z_0) = 0$ for Example 36, Example 37, and Example 38, respectively.

Example 36 (`fib1`)

```
while ⋆ do
  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ z \\ 2yz - x \end{pmatrix}$
end while
```

Example 37 (`fib2`) A Generalised Fibonacci Trace Map

```
while ⋆ do
  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 2xz - y \\ 4xyz - 2x^2 - 2y^2 + 1 \end{pmatrix}$
end while
```

Example 38 (`fib3`) A second Generalised Fibonacci Trace Map

```
while ⋆ do
  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} 1 + x + y + xy - z \\ x \\ y \end{pmatrix}$
end while
```

Examples 39–40 are while loops that generate *Markov triples* [35, Chapter II.3]; that is, at every iteration each loop variable takes an integer value that appears in a Diophantine solution of the Markov equation $x^2 + y^2 + z^2 = 3xyz$.

Example 39 (`markov-triples-toggle`) A while loop that generates an infinite sequence of nodes on the Markov tree (the walk alternates between 'upper' and 'lower' branches).

```
x,y,z = 1,1,2;
branch = 0;
while ⋆ do
  if branch = 0 then
    $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x \\ 3xy - z \\ y \end{pmatrix}$;
    branch = 1;
  else
    $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 3yz - x \\ z \end{pmatrix}$;
    branch = 0;
  end if
end while
```

For this example, our approach generates the polynomial invariant, the *Markov equation*, given by $x^2 + y^2 + z^2 - 3xyz = 0$.

Example 40 (`markov-triples-random`) A while loop that simulates a Bernoulli walk on the Markov tree.

```
x,y,z = 1,1,2
while ⋆ do
```

```
    $p \leftarrow \text{Bernoulli}(1/2)$
    if $p = 1$ then
        $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x \\ 3xy - z \\ y \end{pmatrix}$
    else
        $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 3yz - x \\ z \end{pmatrix}$
    end if
end while
```

For this benchmark, our technique generates both closed-forms and invariants in (higher) moments of program variables such as $\mathbb{E}(x_n^2+y_n^2+z_n^2-3x_ny_nz_n) = 0$ and $\mathbb{E}(x_n-z_n) = 2^{-n}$.

The next two benchmarks concern a special class of polynomial automorphisms, the *Yagzhev* maps, (sometimes *cubic-homogeneous* maps) introduced (independently) by Yagzhev [36] and Bass et al. [37] in the context of the Jacobian conjecture.

Recall that Yagzhev maps $f: \mathbb{C}^n \rightarrow \mathbb{C}^n$ take the form $f(x) = x - g(x)$ such that $\det f'(x) = 1$ for all x, and $g: \mathbb{C}^n \rightarrow \mathbb{C}^n$ is a homogeneous polynomial mapping of degree 3.

De Bondt exhibited a Yagzhev mapping in 10 dimensions that has no linear invariants (providing a counterexample to the "linear dependence conjecture" for the class of maps) [38]. Zampieri demonstrated that De Bondt's example has quadratic and cubic invariants [39] (see Example 41 for such a Yagzhev map in 9 dimensions). Work by Santos Freire, Gorni, and Zampieri [40] exhibited a Yagzhev mapping in 11 dimensions that has neither linear invariants nor quadratic invariants (see Example 42).

Example 41 (`yagzhev9` [39])

```
while $\star$ do
    $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_1x_7x_9 + x_2x_9^2 \\ x_2 - x_1x_7^2 - x_2x_7x_9 \\ x_3 + x_3x_7x_9 + x_4x_9^2 \\ x_4 - x_3x_7^2 - x_4x_7x_9 \\ x_5 + x_5x_7x_9 + x_6x_9^2 \\ x_6 - x_5x_7^2 - x_6x_7x_9 \\ x_7 + (x_1x_4 - x_2x_3)x_9 \\ x_8 + (x_3x_6 - x_4x_5)x_9 \\ x_9 + (x_1x_4 - x_2x_3)x_8 - (x_3x_6 + x_4x_5)x_7 \end{pmatrix}$
end while
```

For this example, our work generates an invariant quadratic homogeneous polynomial in six variables and three symbolic constants, which we can interpret in terms of determinants (as given below):

$$\begin{aligned} &a\begin{vmatrix} x_1(n) & x_2(n) \\ x_3(n) & x_4(n) \end{vmatrix} + b\begin{vmatrix} x_3(n) & x_4(n) \\ x_5(n) & x_6(n) \end{vmatrix} + c\begin{vmatrix} x_1(n) & x_2(n) \\ x_5(n) & x_6(n) \end{vmatrix} \\ &= a\begin{vmatrix} x_1(0) & x_2(0) \\ x_3(0) & x_4(0) \end{vmatrix} + b\begin{vmatrix} x_3(0) & x_4(0) \\ x_5(0) & x_6(0) \end{vmatrix} + c\begin{vmatrix} x_1(0) & x_2(0) \\ x_5(0) & x_6(0) \end{vmatrix}. \end{aligned}$$

Our computation confirms previous work by the authors of [39]. Those authors demonstrated that this example has no linear invariants, but does admit the above quadratic invariant.

Example 42 (`yagzhev11` [40])

$$\textbf{while} \star \textbf{do}$$
$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{pmatrix} \leftarrow \begin{pmatrix} x_1 - x_3 x_{10}^2 \\ x_2 - x_2 x_{11}^2 \\ x_3 + x_1 x_{11}^2 - x_2 x_{10}^2 \\ x_4 - x_6 x_{10}^2 \\ x_5 - x_6 x_{11}^2 \\ x_6 + x_4 x_{11}^2 - x_5 x_{10}^2 \\ x_7 - x_9 x_{10}^2 \\ x_8 - x_9 x_{11}^2 \\ x_9 + x_7 x_{11}^2 - x_8 x_{10}^2 \\ x_{10} + (x_3 x_5 - x_2 x_6) x_7 + (x_1 x_6 - x_3 x_4) x_8 + (x_2 x_4 - x_1 x_5) x_9 \\ x_{11} - x_{10}^3 \end{pmatrix}$$
$$\textbf{end while}$$

For this example, our approach generates an invariant cubic homogeneous polynomial, which we can interpret as the determinant of a matrix in the program variables:

$$\begin{vmatrix} x_1(n) & x_2(n) & x_3(n) \\ x_4(n) & x_5(n) & x_6(n) \\ x_7(n) & x_8(n) & x_9(n) \end{vmatrix} = \begin{vmatrix} x_1(0) & x_2(0) & x_3(0) \\ x_4(0) & x_5(0) & x_6(0) \\ x_7(0) & x_8(0) & x_9(0) \end{vmatrix}.$$

For the avoidance of doubt, the above polynomial was previously found by the authors of [40]. Indeed, our implementation confirms previous results: there are neither linear nor quadratic invariants for this example.

Example 43 (`nagata` [41]) Let us now consider the classical *Nagata automorphism* introduced by Nagata [41, pg. 41] (see also [42]).

$$\textbf{while} \star \textbf{do}$$
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x - 2(xz + y^2)y - (xz + y^2)^2 z \\ y + (xz + y^2)z \\ z \end{pmatrix}$$
$$\textbf{end while}$$

For the Nagata Automorphism, it is easy to see that the variable z is effective and so contributes a linear invariant. When used to search for quadratic closed-forms, our method generates the polynomial

$$c(x(n)z(n) + y(n)^2) = az(0) + bz(0)^2 - az(n) - bz(n)^2 + c(x(0)z(0) + y(0)^2)$$

where a, b, and c are symbolic constants. Here we note that $x(n)z(n)$ and $y(n)^2$ are defective monomials. Our computation confirms the invariants and closed-forms established in [41].

9 Experiments

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

9.1 Implementation

Algorithm 1, our method for synthesising invariants involving defective variables, and Algorithm 2, for the synthesis of solvable loops from unsolvable loops, are implemented in the

Polar tool.[3] We use python3 and the sympy package [43] for symbolic manipulations of algebraic expressions.

9.2 Benchmark selection

While previous works [2, 4, 5, 9, 16, 44] 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 [2, 4, 5, 9, 16, 44]. 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.

In our work we present a set of 23 examples of unsolvable loops, as listed in Table 1.[4] Common to all 23 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 [25, 26, 31, 32]; 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 5 (and hence Example 27), Example 30, and Example 29 from Sect. 8. Eight further benchmarks, as described in Examples 36–43, are drawn from the theoretical physics and pure mathematics literature (references are given in Sect. 8).

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.

Experimental setup

We evaluate our approach in Polar on the examples from Table 1. All our experiments were performed on a machine with a 1.80 GHz 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: high 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 our invariant synthesis technique lies in its ability to handle ‘unsolvable’ loop programs: those with cyclic inter-dependencies and non-linear self-dependencies in the loop body. Regarding Algorithm 2, specifying our method for synthesising solvable from unsolvable loops, the main complexity lies in constructing an invariant involving defective variables. Once, such an invariant has been found, the remaining complexity of Algorithm 2

[3] https://github.com/probing-lab/polar.

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

Table 1 Features of the benchmarks

Benchmark	Var	Def	Term	Deg	Cand- 7	Eqn- 7
`squares (Fig. 1a)`	3	2	8	2	35	113
`squares+ (Ex.32)`	4	2	12	2	35	204
`non-lin-markov-1 (Ex. 5)`	2	2	11	2	35	64
`non-lin-markov-2 (Ex. 28)`	2	2	11	2	35	64
`prob-squares (Ex. 1b)`	3	3	4	3	119	337
`pts (Ex. 30)`	4	2	6	3	35	57
`squares-squared (Ex. 18)`	4	4	15	4	329	–
`bees (Ex. 29)`	5	5	21	5	791	–
`deg-5 (Ex. 35)`	3	2	8	5	35	42
`deg-6 (Ex. 35)`	3	2	8	6	35	42
`deg-7 (Ex. 35)`	3	2	8	7	35	42
`deg-8 (Ex. 35)`	3	2	8	8	35	43
`deg-9 (Ex. 35)`	3	2	8	9	35	43
`deg-500 (Ex. 35)`	3	2	8	500	35	43
`fib1 (Ex. 36)`	3	3	4	2	119	204
`fib2 (Ex. 37)`	3	3	7	3	119	368
`fib3 (Ex. 38)`	3	3	7	2	119	204
`markov-triples-toggle (Ex. 39)`	3	3	10	2	119	454
`markov-triples-random (Ex. 40)`	3	3	9	2	119	254
`yagzhev9 (Ex. 41)`	9	9	29	3	11, 439	–
`yagzhev11 (Ex. 42)`	11	11	30	3	31, 823	–
`nagata (Ex. 43)`	3	2	9	4	35	1313

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 s)

is linear in the number of program variables. Hence, the experimental results for our invariant synthesis technique also show the feasibility of our new method synthesising solvable from unsolvable loops. 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 44 Consider the question: *does the unsolvable loop program* `deg-9` in Table 1 (i.e. Example 35) possess a cubic invariant? 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 cubic polynomial loop invariant, 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 nth iteration is given by

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

Table 2 The time taken to search for polynomial candidates with closed-forms (results in seconds)

BENCHMARK	Candidate degree						
	1	2	3	4	5	6	7
`squares (Fig. 1a)`	*0.95	1.09	1.66	2.72	4.98	11.67	19.87
`squares+ (Ex. 32)`	*0.67	1.02	1.92	3.77	7.39	15.66	28.78
`non-lin-markov-1 (Ex. 5)`	*0.42	*0.68	*1.04	*2.35	*4.02	*9.81	*13.58
`non-lin-markov-2 (Ex. 28)`	*0.38	*0.59	*1.15	*2.40	*3.19	*5.91	*14.91
`prob-squares (Ex. 33)`	*0.76	1.50	4.69	20.85	–	–	–
`squares-and-cube (Fig. 1b)`	0.32	*0.51	*1.20	*4.21	*19.49	–	–
`pts (Ex. 30)`	*0.35	*0.49	*0.71	*1.13	*1.90	*3.42	*5.98
`squares-squared (Ex. 34)`	*0.48	*1.55	*8.92	–	–	–	–
`bees (Ex. 29)`	*0.69	*3.27	*42.16	–	–	–	–
`deg-5 (Ex. 35)`	*0.46	*0.65	*1.01	*1.94	*4.04	*8.20	*19.28
`deg-6 (Ex. 35)`	*0.54	*0.69	*1.62	*1.91	*4.32	*8.24	*19.75
`deg-7 (Ex. 35)`	*0.49	*0.98	*1.06	*1.84	*4.42	*8.98	*19.39
`deg-8 (Ex. 35)`	*0.45	*0.62	*1.08	*2.04	*4.07	*8.93	*20.97
`deg-9 (Ex. 35)`	*0.47	*0.65	*1.11	*1.84	*4.31	*7.85	*19.67
`deg-500 (Ex. 35)`	*0.47	*0.65	*1.08	*1.96	*4.29	*8.58	*21.05
`fib1 (Ex. 36)`	0.31	0.41	*0.91	2.02	2.97	*5.47	14.66
`fib2 (Ex. 37)`	0.31	0.49	*1.34	3.06	8.64	*17.24	–
`fib3 (Ex. 38)`	0.31	0.48	*1.73	3.46	16.92	–	–
`markov-triples-toggle (Ex. 39)`	0.39	0.61	*1.41	2.63	5.84	*9.45	22.71
`markov-triples-random (Ex. 40)`	*0.44	*0.62	*1.65	*3.52	*7.93	*21.39	*71.23
`yagzhev9 (Ex. 41)`	0.47	*2.44	*24.31	–	–	–	–
`yagzhev11 (Ex. 42)`	0.59	7.12	*39.27	–	–	–	–
`nagata (Ex. 43)`	0.33	*1.04	*3.49	*8.19	*40.03	–	–

– = Timeout (75 s); * = Found invariant of the corresponding degree

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). Three of the examples in Table 1 exhibit timeouts (60 s) in the final column. The property common to each of these examples is the high number of monomial terms in any polynomial candidate of degree 7. In turn, this property feeds into a large system of simultaneous equations, which we solve to test for invariants. 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 29). This suggests that mutual dependencies between program variables in the loop assignment explain this phenomenon: such inter-relations lead to the construction of larger systems of equations, which itself feeds into the problem of resolving the recurrence equation associated with a candidate.

Experimental summary

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

10 Conclusion

We establish a new technique that synthesises invariants for loops with unsolvable recurrence operators and show its applicability for deterministic and probabilistic programs. Our work is further extended to translate unsolvable loops into solvable ones, by ensuring that the polynomial loop invariants of the solvable loop are invariants of the given unsolvable loop. Our work is based on a new characterisation of unsolvable loops in terms of effective and defective variables: the presence of defective variables is equivalent to unsolvability. In order to generate 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.

Acknowledgements This research was supported by the Vienna Science and Technology Fund WWTF 10.47379/ICT19018 grant ProbInG, the European Research Council Consolidator Grant ARTIST 101002685, the Austrian Science Fund (FWF) project W1255-N23, and the SecInt Doctoral College funded by TU Wien. This research was also funded in part by the Austrian Science Fund (FWF) [10.55776/F85, 10.55776/W1255], for open access purposes; the authors have applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. We thank the authors of [45] for alerting us to an error in an early version of our manuscript and the anonymous reviewers for their constructive feedback.

Funding Open access funding provided by TU Wien (TUW).

Data availibility The tool's source code, data, and benchmarks used in this work are available in `Polar`'s repository https://github.com/probing-lab/polar.

References

1. Amrollahi D, Bartocci E, Kenison G, Kovács L, Moosbrugger M, Stankovic M (2022) Solving invariant generation for unsolvable loops. In: Singh G, Urban C (eds) Static analysis—29th international symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, proceedings. Lecture notes in computer science, vol 13790, pp 19–43. Springer, Cham. https://doi.org/10.1007/978-3-031-22308-2_3
2. Rodríguez-Carbonell E, Kapur D (2004) Automatic generation of polynomial loop invariants: algebraic foundations. In: Proceedings of the ISSAC, pp 266–273
3. Kovács L (2008) Reasoning algebraically about P-solvable loops. In: Proceedings of the TACAS, pp 249–264
4. Oliveira S, Bensalem S, Prevosto V (2016) Polynomial invariants by linear algebra. In: Proceedings of the ATVA, pp 479–494
5. Kincaid Z, Cyphert J, Breck J, Reps TW (2018) Non-linear reasoning for invariant synthesis. In: Proceedings of the POPL, pp 54–15433
6. Humenberger A, Jaroschek M, Kovács L (2018) Invariant generation for multi-path loops with polynomial assignments. In: Proceedings of the VMCAI, pp 226–246
7. Huang Z, Fan C, Mereacre A, Mitra S, Kwiatkowska MZ (2014) Invariant verification of nonlinear hybrid automata networks of cardiac cells. In: Proceedings of the CAV, pp 373–390
8. Kaminski BL, Katoen J, Matheja C, Olmedo F (2016) Weakest precondition reasoning for expected run-times of probabilistic programs. In: Proceedings of the ESOP, pp 364–389
9. Bartocci E, Kovács L, Stankovic M (2019) Automatic generation of moment-based invariants for prob-solvable loops. In: Proceedings of the ATVA, pp 255–276
10. Müller-Olm M, Seidl H (2004) Computing polynomial program invariants. Inf Process Lett 91(5):233–244
11. Hrushovski E, Ouaknine J, Pouly A, Worrell J (2020) On strongest algebraic program invariants. J ACM (to appear)
12. Elspas B, Green M, Levitt K, Waldinger R (1972) Research in interactive program-proving techniques. Technical report, SRI
13. Katz S, Manna Z (1976) Logical analysis of programs. Commun ACM 19(4):188–206
14. Everest G, Poorten A, Shparlinski I, Ward T (2003) Recurrence sequences, vol 104. Mathematical surveys and monographs. American Mathematical Society, Providence, p 318
15. Kauers M, Paule P (2011) The concrete tetrahedron. Texts and monographs in symbolic computation. Springer, Vienna, p 203
16. Rodríguez-Carbonell E, Kapur D (2007) Generating all polynomial invariants in simple loops. J Symb Comput 42:443–476
17. Farzan A, Kincaid Z (2015) Compositional recurrence analysis. In: FMCAD, pp 57–64
18. Bartocci E, Kovács L, Stankovic M (2020) Analysis of Bayesian networks via prob-solvable loops. In: Proceedings of the ICTAC, pp 221–241
19. Frohn F, Hark M, Giesl J (2020) Termination of polynomial loops. In: Proceedings of the SAS, pp 89–112 (2020)
20. Chakarov A, Sankaranarayanan S (2013) Probabilistic program analysis with martingales. In: Sharygina N, Veith H (eds) Computer aided verification. Springer, Berlin, pp 511–526
21. Lattner C, Adve VS (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the CGO, pp 75–88

22. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580
23. Kauers M, Zimmermann B (2008) Computing the algebraic relations of C-finite sequences and multisequences. J Symb Comput 43:787–803
24. Humenberger A, Jaroschek M, Kovács L (2017) Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In: Proceedings of the ISSAC, pp 221–228
25. Schreuder A, Ong C-L (2019) Polynomial probabilistic invariants and the optional stopping theorem. CoRR. arXiv:1910.12634
26. Chakarov A, Voronin Y-L, Sankaranarayanan S (2016) Deductive proofs of almost sure persistence and recurrence properties. In: Proceedings of the TACAS, pp 260–279
27. May RM (1976) Simple mathematical models with very complicated dynamics. Nature. https://doi.org/10.1038/261459a0
28. Maritz MF (2020) A note on exact solutions of the logistic map. Chaos Interdiscip J Nonlinear Sci 10(1063/1):5125097
29. Moosbrugger M, Stankovič M, Bartocci E, Kovács L (2022) This is the moment for probabilistic loops. Proc ACM Program Lang. https://doi.org/10.1145/3563341
30. Britton NF, Franks NR, Pratt SC, Seeley TD (2002) Deciding on a new home: How do honeybees agree? Proc R Soc Lond Ser B Biol Sci 269(1498):1383–1388
31. Dreossi T, Dang T, Piazza C (2016) Parallelotope bundles for polynomial reachability. In: Proceedings of the HSCC, pp 297–306
32. Sankaranarayanan S, Chou Y, Goubault E, Putot S (2020) Reasoning about uncertainties in discrete-time dynamical systems using polynomial forms. In: Proceedings of the NeurIPS, pp 17502–17513
33. Baake M, Grimm U, Joseph D (1993) Trace maps, invariants, and some of their applications. Int J Mod Phys B 7(6–7):1527–1550. https://doi.org/10.1142/S021797929300247X
34. Roberts JAG, Baake M (1994) Trace maps as 3D reversible dynamical systems with an invariant. J Stat Phys 74(3–4):829–888. https://doi.org/10.1007/BF02188581
35. Cassels JWS (1972) An introduction to diophantine approximation. Cambridge tracts in mathematics and mathematical physics, No. 45. Hafner Publishing Co., New York, p 169. Facsimile reprint of the 1957 edition
36. Jagžev AV (1980) On a problem of O.-H. Keller. Sibirsk Mat Zh 21(5):141–150191
37. Bass H, Connell EH, Wright D (1982) The Jacobian conjecture: reduction of degree and formal expansion of the inverse. Bull Am Math Soc 7(2):287–330. https://doi.org/10.1090/S0273-0979-1982-15032-7
38. Bondt M (2006) Quasi-translations and counterexamples to the homogeneous dependence problem. Proc Am Math Soc 134(10):2849–2856. https://doi.org/10.1090/S0002-9939-06-08335-3
39. Zampieri G (2008) Homogeneous polynomial invariants for cubic-homogeneous functions. Univ Iagel Acta Math 46:99–103
40. Santos Freire R Jr, Gorni G, Zampieri G (2008) Search for homogeneous polynomial invariants and a cubic-homogeneous mapping without quadratic invariants. Univ Iagel Acta Math 46:7–13
41. Nagata M (1972) On automorphism group of $k[x, y]$. Kinokuniya Book Store Co., Ltd., Tokyo, p 53. Department of Mathematics, Kyoto University, Lectures in Mathematics, No. 5
42. van den Essen A, Peretz R (2003) Polynomial automorphisms and invariants. J Algebra 269(1):317–328. https://doi.org/10.1016/S0021-8693(03)00424-1
43. Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka V, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A (2017) SymPy: symbolic computing in Python. PeerJ Comput Sci 3:103
44. Humenberger A, Jaroschek M, Kovács L (2018) Aligator.jl—A Julia package for loop invariant generation. In: Proceedings of the CICM, pp 111–117
45. Bayarmagnai E, Mohammadi F, Prébet R (2024) Algebraic tools for computing polynomial loop invariants. In: Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, ISSAC 2024 (To Appear). https://doi.org/10.1145/3666000.3669710