Sound and Complete Witnesses for Template-based Verification of LTL Properties on Polynomial Programs

KRISHNENDU CHATTERJEE, Institute of Science and Techology Austria, Austria
AMIR KAFSHDAR GOHARSHADY, The Hong Kong University of Science and Technology (HKUST), China
EHSAN KAFSHDAR GOHARSHADY, Institute of Science and Techology Austria, Austria
MEHRDAD KARRABI, Institute of Science and Techology Austria, Austria
ĐORĐE ŽIKELIĆ[*], Singapore Management University, Singapore

Most problems in program verification are undecidable due to Rice's theorem. A classical approach to sidestep the undecidability is template-based verification. Template-based methods first define a sound and complete notion of witness or certificate for the desired property. Informally speaking, a witness is a mathematical object whose existence certifies that the program satisfies the specification. They then focus on programs and witnesses of a specific form (template), e.g. linear or polynomial, to obtain an automated synthesis algorithm that finds a witness. For example, ranking functions are sound and complete witnesses for termination. Although it is undecidable whether a program has a ranking function, there are automated approaches that check whether an imperative program with linear/polynomial assignments, guards and invariants has a linear/polynomial ranking function. Thus, this important special case of the problem is actually decidable. Similar template-based approaches exist in the literature for the classical problems of safety (invariant generation) and reachability (safety violation). While finding sound and complete witness notions, or sound an incomplete witnesses that are easy to synthesize is straightforward, the key challenge is finding a notion of witness that is both sound and complete on the one hand and amenable to automated template-based synthesis on the other.

In this work, we study the classical problem of verifying programs with respect to formal specifications given in the linear temporal logic (LTL). LTL is a rich and expressive logic that can specify important properties of programs. This includes, but is not limited to, termination, safety, liveness, progress, recurrence and reach-avoid properties. We first present novel sound and complete witnesses for LTL verification over imperative programs. Our witnesses are applicable to both universal (all runs) and existential (some run) settings. We then consider polynomial arithmetic programs, i.e. programs in which every assignment and guard consists only of polynomial expressions, with specifications as LTL formulas in which atomic propositions are polynomial constraints. For this setting, we provide an efficient algorithm to automatically synthesize such LTL witnesses. Our synthesis procedure is both sound and semi-complete, i.e. complete for any fixed polynomial degree in the templates. In other words, we provide the first template-based approach for polynomial programs that can handle any LTL formula as its specification. Our approach has termination guarantees with sub-exponential time complexity and generalizes and unifies previous methods for termination, safety and reachability, since they are expressible in LTL. Finally, we present experimental results demonstrating the effectiveness of our approach and that it can handle programs which were beyond the reach of previous state-of-the-art tools.

1 INTRODUCTION

Static Analysis of Programs. Given an input program and a specification that describes a desired property about the program's traces, static analysis aims to automatically verify whether

[*]Part of the work done while the author was at the Institute of Science and Technology Austria

Authors' addresses: Krishnendu Chatterjee, krishnendu.chatterjee@ist.ac.at, Institute of Science and Techology Austria, P.O. Box 1212, Klosterneuburg, Austria; Amir Kafshdar Goharshady, goharshady@cse.ust.hk, The Hong Kong University of Science and Technology (HKUST), Hong Kong, China; Ehsan Kafshdar Goharshady, ehsan.goharshady@ist.ac.at, Institute of Science and Techology Austria, P.O. Box 1212, Klosterneuburg, Austria; Mehrdad Karrabi, mehrdad.karrabi@ist.ac.at, Institute of Science and Techology Austria, P.O. Box 1212, Klosterneuburg, Austria; Đorđe Žikelić, dzikelic@smu.edu.sg, Singapore Management University, Singapore.

the program satisfies the specification [1]. Important and well-studied specifications include termination, safety, liveness, progress, reach-avoid and many other classical properties. To unify the verification approaches, specifications are often formalized by logical formulas. A logic provides a precise and concise description of specifications. A required aspect of the logic is expressiveness, i.e., the ability to express the classical temporal properties of the programs. A rich and expressive temporal logic for program behaviors is the linear-time temporal logic (LTL).

Linear-time Temporal Logic. The Linear-time Temporal Logic (LTL) [2] is one of the most classical and well-studied frameworks for formal specification, model checking and program verification [3]. In LTL, we consider a set AP of atomic propositions and an infinite trace which tells us which propositions in AP hold at any given time. LTL formulas are then able to not only express propositional logical operations, but also modalities referring to the future. For example, $\mathsf{X}\ p$ requires that p holds in the next timeslot, whereas $\mathsf{F}\ q$ means q should hold at some time in the future and $\mathsf{G}\ r$ requires that r holds at every time in the future. This allows LTL to express common verification tasks such as termination, liveness, fairness and safety.

Since its introduction in 1977, LTL has been widely studied by the verification, programming languages and model checking communities. In its most classical format, the traces are runs of a given Kripke structure K, which is a simplified model of a program, and the model checking problem is to decide whether all such runs satisfy a given LTL formula φ. The satisfiability problem asks whether such a K exists, given a formula φ, and the synthesis problem asks for a concrete K. Both model checking and satisfiability are known to be PSPACE-complete for LTL, whereas synthesis is 2EXPTIME-complete [4–6]. There is also a natural and elegant correspondence between LTL formulas and Non-deterministic Büchi automata (NBW) [7, 8], as well as a wide variety of approaches and tools that perform the LTL-to-NBW translation [9–19].

Program Analysis and LTL. LTL is a prominent logic in static program analysis and its model checking problem over infinite-state systems and programs has been a well-studied research area. Traditionally, LTL specifications were applied to sequential programs [2] and concurrent programs [20]; and reactive synthesis with LTL specification were also considered [4]. Two classical and widely-studied special cases of LTL are safety specifications [3], which intuitively cover all properties that require the program never does something undesirable, and liveness specifications [21–23], which informally describe properties that the program (eventually) does something desirable. There are many approaches for automated verification of LTL over various families of programs. The techniques in the literature include predicate abstraction [24, 25], compositional reasoning [26], symbolic execution [27, 28], and temporal prophecies [29].

Witnesses. Given a specification φ and a program P, a *witness* is a mathematical object whose existence proves that the specification φ is satisfied by P. We say that a witness family is *sound and complete* when for every program P and specification φ, we have $P \models \varphi$ if and only if there is a witness in the family that certifies it. For example, ranking functions [1] are well-known sound and complete witnesses for termination.

Motivation for Witnesses. Witnesses are especially useful in dealing with undecidable problems in verification and static analysis, which includes all non-trivial semantic properties [30]. This is because although the general case of the problem is undecidable, having a sound and complete notion of a witness can lead to algorithms that check for the existence of witnesses of a special form. For example, while termination is undecidable [31], and hence so is the equivalent problem of deciding the existence of a ranking function, there are nevertheless sound and complete algorithms for synthesis of *linear* ranking functions [32]. Similarly, while reachability (safety violation) is undecidable, there exist sound and semi-complete procedures for synthesis of witnesses in linear and polynomial forms [33]. This highlights the importance of witnesses in verification and program analysis.

Polynomial Programs. In this work, we mainly focus on imperative programs with polynomial arithmetic. More specifically, our programs have real variables and the right-hand-side of every assignment is a polynomial expression with respect to program variables. Similarly, the guard of every loop or branch is also a boolean combination of polynomial inequalities over the program variables. See Figure 3 as an example.

Motivation for Polynomial Programs with LTL specifications. There are several reasons to consider polynomial programs with LTL specifications:

- Many real-world families of programs, such as, programs for cyber-physical systems and smart contracts, can be modeled in this framework [34–36].
- Polynomial programs are one of the most general families for which finding polynomial witnesses for reachability and safety are known to be decidable [33, 37, 38]. Hence, they provide a desirable tradeoff between decidability and generality.
- Using abstract interpretation, non-polynomial behavior in a program can be removed or replaced by non-determinism. Moreover, one can approximate any continuous function up to any desired level of accuracy by a polynomial. This is due to the Stone–Weierstrass theorem [39]. Thus, analysis of polynomial programs can potentially be applied to many non-polynomial programs through either abstract interpretation or numerical approximation of the program's behavior.
- Previous works have studied (a) linear/affine programs with termination, safety, and reachability specifications [32, 40, 41], and (b) polynomial programs with termination, safety and reachability properties [33, 37, 38, 42]. Since LTL subsumes all these specifications, polynomial program analysis with LTL provides a unifying and general framework for all these previous works.
- LTL Verification over polynomial programs is considered in [43, 44] which provide a sound and relatively-complete algorithm. They have a general framework with integer variables and recursive functions. Thus, they have no guaranteed complexity bounds.

Our Contributions. In this work, our contributions are threefold:

- *Sound and complete witness.* First, by exploiting the connections to Büchi automata, we propose a novel family of sound and complete witnesses for general LTL formulas. This extends and unifies the known concepts of ranking functions [1], inductive reachability witnesses [33] and inductive invariants [40], which are sound and complete witnesses for termination, reachability and safety, respectively. Our witness characterization result is not limited to polynomial programs and is applicable to any program. Crucially, it is also well-suited for template-based synthesis.
- *Witness synthesis algorithm.* On the algorithmic side, we consider polynomial programs and present a sound and semi-complete template-based algorithm to synthesize polynomial LTL witnesses. A template represents polynomials of a given degree. The semi-completeness guarantee signifies that there is a witness polynomial of large enough degree, and hence by iteratively increasing the degrees of the templates, our algorithm handles all polynomial programs and LTL specifications. This algorithm is a generalization of the template-based approaches in [32, 33, 40] which considered termination, reachability and safety. It has sub-exponential runtime for any fixed degree of the template polynomials.
- *Experimental results.* Finally, on the experimental side, we provide a prototype implementation of our approach and comparisons with state-of-the-art LTL model checking tools. Our experiments show that our approach is applicable in practice and can handle

many instances that were beyond the reach of previous methods. Thus, our algorithm, besides the theoretical semi-completeness guarantees, also presents an effective practical method that is applicable to new instances that were not solvable using existing methods.

Organization. Section 2 discusses related work. We then present two motivating examples for LTL program analysis in Section 3. Section 4 sets up the arena, establishes the connection between LTL and Büchi automata in our setting, and formally defines the problems we study. In Section 5, we provide our witnesses for LTL and prove that they are sound and complete. This is followed by our synthesis algorithm in Section 6. Finally, Section 7 provides experimental results.

2 RELATED WORKS

We now provide an overview of previous related works and compare them with our approach.

Related Works on Linear Programs. There are many approaches focusing on linear witness synthesis for important special cases of LTL formulas. For example, [32, 45] consider the problem of synthesizing linear ranking functions (termination witnesses) over linear arithmetic programs. The works [40, 41] synthesize linear inductive invariants (safety witnesses), while [46] considers probabilistic reachability witnesses. The notion of invariants has also been used for error localization [47]. The work [48] handles a larger set of verification tasks and richer settings, such as context-sensitive interprocedural program analysis. All these works rely on the well-known Farkas lemma [49] and can handle programs with linear/affine arithmetic and synthesize linear/affine witnesses. In comparison, our approach is (i) applicable to general LTL formulas and not limited to a specific formula such as termination or safety, and (ii) able to synthesize *polynomial* witnesses for *polynomial* programs with soundness and completeness guarantees. Thus, our setting is more general in terms of (a) formulas, (b) witnesses, and (c) programs that can be supported.

Related Works on Polynomial Programs. Similar to the linear case, there is a rich literature on synthesis of polynomial witnesses over polynomial programs. However, these works again focus on specific special formulas only and are not applicable to general LTL. For example, [42, 50–53] consider termination analysis, [38] extends the invariant generation (safety witness synthesis) algorithm of [40] to the polynomial case and [54] further adds support for probabilistic programs. The works [55–57] consider alternative types of witnesses for safety, i.e. barriers, and obtain similarly successful synthesis algorithms. In contrast, [33, 58] synthesize reachability (safety violation) witnesses. Finally, [59] considers the problem of template-based synthesis of a program, when the specification and a skeleton of the desired program are given as input.

Since we can handle any arbitrary LTL formula, our approach can be seen as a unification and generalization of all template-based verification algorithms mentioned above. In both cases above, some of the previous works are incomparable to ours since they consider probabilistic programs, whereas our setting has only non-probabilistic polynomial programs. Note that we do allow non-determinism.

Related Works on LTL Model Checking. There are thousands of works on LTL model checking and there is no way we can do justice to all of them. We refer to [60, 61] for an excellent treatment of the finite-state cases. Some of the works that provide LTL model checking over infinite-state systems/programs are as follows:

- A prominent technique in this area is predicate abstraction [24, 25, 62], which uses a finite set of abstract states defined by an equivalence relation based on a finite set of predicates to soundly, but not completely, reduce the problem to the finite-state case.
- [26] uses a compositional approach to falsify LTL formulas and find an indirect description of a path that violates the specification.

- There are several symbolic approaches including [27] which is focused on fairness and [28] which is applicable to LLVM. Another work in this category is [63], whose approach is to repeatedly rule out infeasible finite prefixes in order to find a run of the program that satisfies/violates the desired LTL formula. The work [64] uses CTL-based approaches that might report false counter-examples when applied to LTL. It then identifies and removes such spurious counterexamples using symbolic determinization.
- The work [65] presents a framework for proving liveness properties in multi-threaded programs by using well-founded proof spaces.
- The recent work [29] uses temporal prophecies, inspired by classical prophecy variables, to provide significantly more precise reductions from general temporal verification to the special case of safety.
- There are many tools for LTL-based program analysis. For example, T2 [66] is able to verify a large family of liveness and safety properties, nuXmv [67] is a symbolic model checker with support for LTL, F3 [26] proves fairness in infinite-state transition systems, and Ultimate LTLAutomizer [63] is a general-purpose tool for verification of LTL specifications over a wide family of programs with support for various types of variables.
- Finally, we compare against the most recent related work [44]. This work provides relative-completeness guarantees for general programs with LTL specifications. Since it considers integer programs with recursive functions, there is no complexity guarantee provided. The earlier work [43] provides several special cases where termination is guaranteed. However, no runtime bounds are established. In contrast, our approach has both termination guarantees and sub-exponential time complexity for fixed degree. We provide an experimental comparison in Section 7.

As shown by our experimental results in Section 7, our completeness results enable our tool to handle instances that other approaches could not. On the other hand, our method is limited to polynomial programs and witnesses. Thus, there are also cases in which our approach fails but some of the previous tools succeed, e.g. when the underlying program requires a non-polynomial witness. In particular, Ultimate LTLAutomizer [63] is able to handle non-polynomial programs and witnesses, too.

Constraint solving. Our algorithm relies on constraint-solving methods. Many template-based synthesis methods use constraint-solving, e.g. [33, 40]. Moreover, constraint-solving has also been used in context of probabilistic programs [68, 69]. However to the best of our knowledge such methods have not been applied to general LTL model-checking of polynomial programs.

3 MOTIVATING EXAMPLES FOR LTL PROGRAM ANALYSIS

In this work, we consider two variants of LTL program analysis: *Universal* analysis asks whether *every run* of a given program satisfies a given LTL specification φ, whereas *existential* analysis aims to decide if *there exist* a run satisfying φ. We illustrate these variants on two motivating examples. Both examples consider pairs of programs with non-determinism and LTL specifications whose satisfaction we wish to analyze. For each program pair, only some (but not all) runs of the first program satisfy the LTL specification, whereas all runs of the second program satisfy the LTL specification. Moreover, in both examples, we consider LTL specifications that cannot be reduced to termination, reachability or safety. Thus, the goal in each of these examples cannot be simply reduced to termination, reachability or safety analysis and hence, previous approaches for these classical problems are inapplicable here. In other words, LTL specifications not only generalize safety, liveness, termination and reachability, but are also strictly more expressive.

```
Precondition($l_{init}$) : $n \geq 1, i = 1, s = 0$
$l_1$:  while  $i \leq n$  do
            assert($s \leq n \cdot (n-1)/2$)
            if  *  then
$l_2$:              $s = s + i$
            else
$l_3$:              skip
            fi
      done
$l_t$:

LTL  specification:
$\left(\mathsf{F}\ at(l_t)\right) \wedge \left(\mathsf{G}\ (at(l_1) \Rightarrow s \leq n \cdot (n-1)/2)\right)$
```

```
Precondition($l_{init}$) : $n \geq 1, i = 1, s = 0$
$l_1$:  while  $i \leq n$  do
            assert($s \leq n \cdot (n+1)/2$)
            if  *  then
$l_2$:              $s = s + i$
            else
$l_3$:              skip
            fi
      done
$l_t$:

LTL  specification:
$\left(\mathsf{F}\ at(l_t)\right) \wedge \left(\mathsf{G}\ (at(l_1) \Rightarrow s \leq n \cdot (n+1)/2)\right)$
```

Fig. 1. Our motivating example program pair for Existential and Universal LTL Program Analysis with reach-avoid specifications. The reach-avoid specification requires that every run in the program terminates while satisfying all assertions shown in blue. Only some (but not all) runs of the program in the left satisfy the reach-avoid specification. In contrast, all runs of the program in the right satisfy the reach-avoid specification.

Example 1 (Reach-avoid specifications). *Our first example program pair illustrates the Existential and Universal LTL Program Analyses for* reach-avoid specifications. *Intuitively, a run is said to satisfy a reach-avoid specification if it reaches some "target" set of program states while avoiding some "unsafe" set of program states. A common scenario in which reach-avoid specifications arise in program analysis is if we require all runs in a program to terminate while satisfying all assertions specified by the programmer. This example illustrates one such scenario. Reach-avoid specifications are also extremely common in the analysis of dynamical or hybrid systems [70–72].*

*Consider the program pair shown in Figure 1. Both programs initialize variable i to 1, variable s to 0, and consist of a loop that is executed as long as $i \leq n$. In each loop iteration, the programs may or may not increment the value of s by i. Whether s is incremented or not is subject to a non-deterministic choice, indicated in the program syntax by a command **if** $*$ **then**. At the end of each loop iteration, in both programs i is incremented by 1. The two programs only differ in the assertion by which they are annotated. The program on the left is annotated by **assert**($s \leq n(n-1)/2$) at the location l_1 of the loop head, whereas the program on the right is annotated by **assert**($s \leq n(n+1)/2$) at the same location. Assertions in both programs are highlighted in blue.*

In both programs, a run satisfies the reach-avoid specification if it terminates (i.e. reaches l_t) while satisfying assertions in all visited program locations. This specification for programs in Figure 1 left and right is formalized via the following LTL formulas:

$$\phi_{left} = \left[\left(\mathsf{F}\ at(l_t)\right) \wedge \left(\mathsf{G}\ (at(l_1) \Rightarrow s \leq n \cdot (n-1)/2)\right)\right],$$
$$\phi_{right} = \left[\left(\mathsf{F}\ at(l_t)\right) \wedge \left(\mathsf{G}\ (at(l_1) \Rightarrow s \leq n \cdot (n+1)/2)\right)\right],$$

where we use $at(l)$ to denote the atomic proposition that a program run is in location l. We will formalize the syntax and semantics of LTL in Section 4. The two specifications are different due to different assertions by which the two programs in Figure 1 are annotated.

Note that all runs of the program in Figure 1 (right) satisfy ϕ_{right}, whereas only some runs of the program in Figure 1 (left) satisfy ϕ_{left}. Indeed, the value of s along a run is maximized if the run always takes the then-branch of the non-deterministic choice, resulting in the value $\sum_{i=1}^{n} i = n \cdot (n+1)/2$ upon termination. This run violates the assertion in the program in Figure 1 (left) and therefore it violates

```
Precondition($l_{init}$): $n = 100, x = 0, y = 0$
$l_1$:  while true do
            $n = 100$
            $x = 1$
            if * then
                $y = -1$
            elif * then
                $y = 0$
            else
                $y = 1$
            fi

$l_2$:      while $n > 0$ do
                $n = n - y$
            done
$l_3$:      $x = 0$
        done
$l_t$:

LTL specification:
G $(x = 1 \Rightarrow (\mathsf{F}\ x = 0))$
```

```
Precondition($l_{init}$): $n = 100, x = 0, y = 0$
$l_1$:  while true do
            $n = 100$
            $x = 1$
            if * then
                $y = -1$
            elif * then
                $y = 0$
            else
                $y = 1$
            fi
$l_2$:      assume($y \geq 1$)
$l_3$:      while $n > 0$ do
                $n = n - y$
            done
$l_4$:      $x = 0$
        done
$l_t$:

LTL specification:
$(\mathsf{G}\ (at(l_2) \Rightarrow y \geq 1)) \Rightarrow (\mathsf{G}\ (x = 1 \Rightarrow (\mathsf{F}\ x = 0)))$
```

Fig. 2. Our motivating example program pair for Existential and Universal LTL Program Analysis with progress specifications. The progress specification requires that if a run satisfies all assume statements shown in blue, whenever the run reaches a program state with $x = 1$, it must also subsequently reach a program state with $x = 0$. Only some (but not all) runs of the program in the left satisfy the progress specification. In contrast, all runs of the program in the right satisfy the progress specification.

ϕ_{left}. On the other hand, all runs satisfy the assertion in the program in Figure 1 (right) and also satisfy ϕ_{right}.

The goal of the Universal LTL Program Analysis for this example is to show that all runs *of the program in Figure 1 (right) satisfy ϕ_{right}. On the other hand, the goal of the Existential LTL Program Analysis is to show that* not all runs *of the program in Figure 1 (left) satisfy ϕ_{left}. The latter is achieved by using the Existential LTL Program Analysis to analyze the negated LTL specification*

$$\neg\phi_{left} = \left[\left(\mathsf{G}\ \neg at(l_t)\right) \vee \left(\mathsf{F}\ (at(l_1) \wedge s > n \cdot (n-1)/2)\right)\right].$$

Hence, the Universal LTL Program Analysis **proves** *a reach-avoid specification for all runs, whereas the Existential LTL Program Analysis* **refutes** *a reach-avoid specification for some run.*

Example 2 (Progress specifications). *Our second example program pair illustrates the Existential and Universal LTL Program Analyses for* progress specifications. *Progress specifications are a subclass of liveness specifications. Intuitively, given two sets of program states A and B, a run is said to satisfy the progress specification for A and B if whenever the run reaches a state in A it must also subsequently reach a state in B [73]. Examples of progress specifications include starvation freedom, fairness or recurrence [74]. For our example, we consider a variant of the program and the progress specification in Section 2 of [75] with finite non-determinism.*

Consider the program pair shown in Figure 2. Both programs initialize variable n to 100, variables x and y to 0, and consist of a loop that is executed indefinitely. In each loop iteration, both programs first set the value of x to 1. They then non-deterministically choose a value from the set $\{-1, 0, 1\}$ and assign it to y. Then, both programs execute an inner loop that decrements n by y as long as $n > 0$. Finally, *upon inner loop termination, both programs set the value of x to 0. The only difference between the two programs is that the program in Figure 2 (right) is also annotated by an assumption **assume**($y \geq 1$) at the location l_2 before entering the inner loop. The assumption in the program is highlighted in blue.*

As in [75], we consider the progress specification for sets of states in which $x = 1$ and in which $x = 0$. In other words, a run satisfies the specification if it either violates some assumption in the program, or if whenever it reaches a state in which $x = 1$ it must also subsequently reach a state in which $x = 0$. This specification for programs in Figure 2 is formalized via the following LTL formulas:

$$\phi_{left} = \left[\mathsf{G}\left(x = 1 \Rightarrow (\mathsf{F}\ x = 0)\right)\right],$$
$$\phi_{right} = \left[\left(\mathsf{G}\ (at(l_2) \Rightarrow y \geq 1)\right) \Rightarrow \left(\mathsf{G}\ (x = 1 \Rightarrow (\mathsf{F}\ x = 0))\right)\right].$$

Note that all runs of the program in Figure 2 (right) satisfy ϕ_{right}. Indeed, if a run satisfies all assume statements, then the program must assign $y = 1$ in every outer loop iteration. This means that the inner loop must always terminate as each inner loop iteration decrements n by 1, therefore the run reaches a state with $x = 0$ infinitely many times. On the other hand, only some runs of the program in Figure 2 (left) satisfy ϕ_{left}. Any run which in some outer loop iteration assigns $y = -1$ or $y = 0$ enters the inner loop that never terminates since n is never decremented, so the value of x is 1 indefinitely.

The goal of the Universal LTL Program Analysis for this example is to show that all runs *of the program in Figure 1 (right) satisfy ϕ_{right}. On the other hand, the goal of the Existential LTL Program Analysis is to show that* not all runs *of the program in Figure 1 (left) satisfy ϕ_{left}. The latter is achieved by using the Existential LTL Program Analysis to analyze the negated LTL specification*

$$\neg\phi_{left} = \left[\mathsf{F}\left(x = 1 \wedge \mathsf{G}\, x \neq 0\right)\right].$$

Hence, the Universal LTL Program Analysis **proves** *a progress specification for all runs, whereas the Existential LTL Program Analysis* **refutes** *a progress specification for some run.*

The above examples highlight the importance of analysis of non-linear programs with respect to specifications described as LTL formulas, which are beyond safety and termination properties. This is precisely the focus of the current work.

4 TRANSITION SYSTEMS, LTL AND BÜCHI AUTOMATA

Notation. We use $\mathbb{N}$ and $\mathbb{R}$ to denote the sets of natural and real numbers, respectively. We define $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$, $[n] := \{0, 1, 2, \ldots, n-1\}$ and Λ as the empty string. When $e \in \mathbb{R}^n$, we use e_i to denote the i-th component of e. Given a finite set $\mathcal{V}$ of real-valued variables, a variable valuation $e \in \mathbb{R}^{|\mathcal{V}|}$ and a boolean predicate φ over $\mathcal{V}$, we write $e \models \varphi$ when φ evaluates to true upon substituting variables by the values given in e.

We consider imperative numerical programs with real-valued variables, containing standard programming constructs such as assignments, branching and loops. In addition, our programs can have finite non-determinism. We denote non-deterministic branching in our syntax by **if** $*$ **then**. See Figure 3 for an example. We use transition systems to formally model programs.

Transition systems. An infinite-state *transition system* is a tuple $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$, where:

- $\mathcal{V} = \{x_0, \ldots, x_{n-1}\}$ is a finite set of real-valued *program variables.*
- L is a finite set of *locations.*
- $l_{init} \in L$ is the *initial location.*
- $\theta_{init} \subseteq \mathbb{R}^n$ is a set of *initial variable valuations.*
- $\mapsto$ is a finite set of *transitions.* Each transition $\tau \in \mapsto$ is of the form $\tau = (l, l', G_\tau, U_\tau)$, where l is the source location, l' is the target location, G_τ is the guard of the transition, which is a boolean predicate over $\mathcal{V}$, and $U_\tau : \mathbb{R}^n \to \mathbb{R}^n$ is the update function of the transition.

```
Precondition ($l_{init}$): $x_0 \geq 0$
$l_1$:  while $x_0 \geq 0$ do
          if * then
$l_2$:        $x_0 = x_0^2 + 1$
          else
$l_3$:        $x_0 = x_0^2 - 1$
          fi
        done
$l_t$:
```

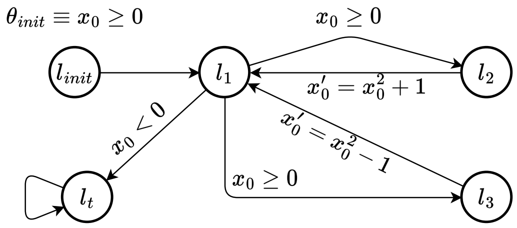

Fig. 3. An example program (left) and its transition system (right). Note that there is non-determinism at l_1.

Translating programs into transition systems is a standard process. Thus, we omit the details. In the rest of the paper, we assume we are given a transition system $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ of the program that we wish to analyze.

States and Successors. A *state* in $\mathcal{T}$ is an ordered pair (l, e) with $l \in L$ and $e \in \mathbb{R}^n$. A state (l, e) is said to be *initial* if $l = l_{init}$ and $e \in \theta_{init}$. We use $\mathscr{S}$ and $\mathscr{S}_{init}$ to denote the sets of all states and initial states, respectively. We also assume the existence of a special *terminal location* l_t with a single outgoing transition which is a self-loop $(l_t, l_t, \text{true}, Id)$ with $Id(e) = e$ for each $e \in \mathbb{R}^n$. A state (l', e') is a *successor* of (l, e), denoted as $(l, e) \mapsto (l', e')$, if there exists a transition $\tau = (l, l', G_\tau, U_\tau) \in \mapsto$ such that $e \models G_\tau$ and $e' = U_\tau(e)$. We assume each state has at least one successor. This is without loss of generality, since we can always introduce self-loop transitions to a terminal location. We impose this assumption to ensure all runs of the program are infinite, since LTL semantics are defined on infinite traces.

Schedulers. We resolve non-determinism in transition systems using the standard notion of schedulers. A *scheduler* is a function σ that chooses the initial state in $\mathscr{S}_{init}$ in which the program execution should start, and then for each state specifies the successor state to which the program execution should proceed. Formally, a scheduler is a function $\sigma\colon \mathscr{S}^* \to \mathscr{S}$ such that $\sigma(\Lambda) \in \mathscr{S}_{init}$ and $s \mapsto \sigma(\pi \cdot s)$ for every sequence of states π and state s. We use Σ to denote the set of all schedulers in $\mathcal{T}$. A scheduler is *memory-less* if its choice of a successor state only depends on the final state in the sequence and does not depend on prior history, i.e. if $\sigma(\pi \cdot s) = \sigma(\pi' \cdot s)$ for every state s and pair of sequences π and π'.

Runs. A *run* in $\mathcal{T}$ is an infinite sequence of successor states starting in $\mathscr{S}_{init}$. More precisely, $\pi\colon \mathbb{N}_0 \to \mathscr{S}$ is a run of $\mathcal{T}$ if $\pi(0) \in \mathscr{S}_{init}$ and $\pi(i-1) \mapsto \pi(i)$ for every $i \in \mathbb{N}$. Every finite prefix of a run is called a *trajectory*. We denote the trajectory containing the first k states in π by π^k and write π^{k+} for the remaining suffix of the run π. A scheduler σ naturally induces a run π_σ in the transition system, defined as $\pi_\sigma(0) = \sigma(\Lambda)$ and $\pi_\sigma(i) = \sigma(\pi_\sigma^i)$ for every $i \in \mathbb{N}$.

Example. Figure 3 (right) shows a transition system with $\mathscr{S}_{init} = \{(l_{init}, x_0) \mid x_0 \geq 0\}$. The transition from l_1 to l_t has guard $x_0 < 0$ and no updates, while the transitions from l_1 to l_2 and l_3 have the same guards and the scheduler has to choose between them whenever $x \geq 0$ is satisfied at l_1.

We now review the Linear-time Temporal Logic (LTL). Since LTL is standard, we keep our description brief and refer to [74] for a more detailed treatment.

Linear-time Temporal Logic. Let AP be a finite set of atomic propositions. LTL formulas are inductively defined as follows:

- If $p \in \mathsf{AP}$, then p is an LTL formula.
- If φ and ψ are LTL formulas, then $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathsf{X}\ \varphi$, $\mathsf{G}\ \varphi$, $\mathsf{F}\ \varphi$ and $\varphi\ \mathsf{U}\ \psi$ are all LTL formulas.

$\neg$, $\vee$ and $\wedge$ are the propositional negation, disjunction and conjunction while $\mathsf{X}, \mathsf{G}, \mathsf{F}$ and U are the *next, globally, finally* and *until* temporal operators.

Atomic Propositions. To use LTL over the transition system $\mathcal{T}$, we first need to specify a finite set of atomic propositions AP. In this work, we let the set AP consist of (i) finitely many constraints of the form $\exp(\mathbf{x}) \geq 0$ where $\exp\colon \mathcal{V} \to \mathbb{R}$ is an arithmetic expression over $\mathcal{V}$, and (ii) an atomic proposition $at(l)$ for each location l in $\mathcal{T}$. Intuitively, our atomic propositions can either (i) check whether an arithmetic inequality holds over the program variables, or (ii) check if the program/transition system is at a particular location l. Note that unlike classical LTL settings, our atomic propositions are not necessarily independent. For example, if we have $p_1 := x \geq 0$ and $p_2 := x + 1 \geq 0$, it is impossible to have $p_1 \wedge \neg p_2$ at any point in time.

Semantics. Each atomic proposition p in AP determines a set of states in $\mathcal{T}$ at which p is satisfied. We write $(l, e) \models p$ to denote such satisfaction. Specifically, if p is of the form $\exp(\mathbf{x}) \geq 0$ then $(l, e) \models p$ if $\exp(e) \geq 0$, and if p is of the form $at(l')$ then $(l, e) \models p$ if $l = l'$. Our LTL semantics are then defined in the usual way. Given an infinite sequence $\pi = (l_0, e_0), (l_1, e_1), \dots$ of states, we write $\pi \models \varphi$ to denote that it satisfies φ and have:

- For every $p \in \mathsf{AP}$, $\pi \models p$ iff $(l_0, e_0) \models p$;
- $\pi \models \mathsf{X}\, \varphi$ iff $\pi^{1+} \models \varphi$;
- $\pi \models \mathsf{F}\, \varphi$ iff $\pi^{i+} \models \varphi$ for some $i \in \mathbb{N}_0$;
- $\pi \models \mathsf{G}\, \varphi$ iff $\pi^{i+} \models \varphi$ for all $i \in \mathbb{N}_0$;
- $\pi \models \varphi\, \mathsf{U}\, \psi$ iff there exists $k \in \mathbb{N}_0$ s.t. $\pi^{i+} \models \varphi$ for all $0 \leq i < k$ and, additionally, $\pi^{k+} \models \psi$.

Büchi Specifications. A Büchi specification is simply a subset $\mathcal{B} \subseteq \mathcal{S}$ of states. A run π is *$\mathcal{B}$–Büchi* if it visits $\mathcal{B}$ infinitely many times, i.e. if the set $\{i \mid \pi(i) \in \mathcal{B}\}$ is infinite. A scheduler is $\mathcal{B}$–Büchi if its run is.

Program Analysis with LTL Specifications. We now define the LTL program analysis problems that we consider in this work. Given a transition system $\mathcal{T}$ and an LTL formula φ, we are interested in schedulers σ for which the induced run π_σ satisfies φ. In particular, we consider the following two decision problems:

(1) *Existential LTL Program Analysis (ELTL-PA).* Given a transition system $\mathcal{T}$ and an LTL formula φ in $\mathcal{T}$, prove that *there exists a scheduler* σ for which the induced run π_σ satisfies φ.
(2) *Universal LTL Program Analysis (ULTL-PA).* Given a transition system $\mathcal{T}$ and an LTL formula φ in $\mathcal{T}$, prove that *for every scheduler* σ the induced run π_σ satisfies φ.

Example. Consider the transition system in Figure 3 and the LTL formula $\varphi = [\mathsf{G}(at(l_3) \Rightarrow (\mathsf{F}\, at(l_2))]$. The run that starts at $(l_{init}, 1)$ and chooses l_2 if $x_0 = 0$ and l_3 whenever $x_0 = 1$, satisfies φ. Therefore, in this case, the answer to the ELTL-PA problem is positive. Additionally, deciding termination of a program with terminal location l_t is equivalent to the ULTL-PA problem of $[\mathsf{F}\ \mathsf{G}\ at(l_t)]$ on the same program.

Program Analysis with Büchi Specifications. Similar to LTL, Büchi specifications give rise two main decision problems as follows:

(1) *Existential Büchi Program Analysis (EB-PA).* Given a transition system $\mathcal{T}$ and a Büchi specification $\mathcal{B}$ on $\mathcal{T}$, prove that *there exists a scheduler* σ for which the induced run π_σ is $\mathcal{B}$-Büchi.
(2) *Universal Büchi Program Analysis (UB-PA).* Given a transition system $\mathcal{T}$ and a Büchi specification $\mathcal{B}$ on $\mathcal{T}$, prove that *for every scheduler* σ the induced run π_σ is $\mathcal{B}$-Büchi.

Büchi Automata [74, 76]. A *non-deterministic Büchi automaton (NBW)* is an ordered tuple $N = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is a finite alphabet, $\delta\colon Q \times A \to 2^Q$ is a transition relation, q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. An infinite word $a_0, a_1, \dots$

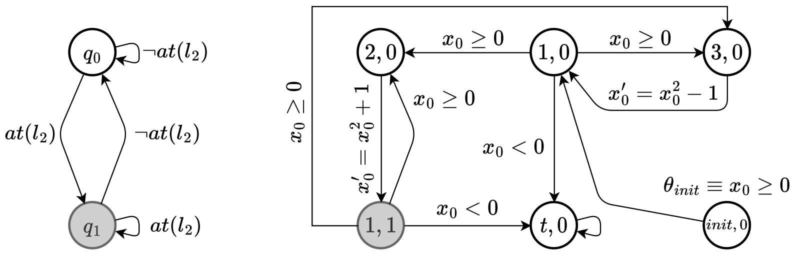

Fig. 4. An NBW accepting G F $at(l_2)$ with gray accepting nodes (left) and the product of the transition system in Figure 3 and this NBW (right). A node labeled i, j represents location (l_i, q_j). Unreachable locations have been removed.

of letters in the alphabet A is accepted by N if it gives rise to at least one accepting run in N, i.e. if there exists a run $q_0, q_1, \dots$ such that $q_{i+1} \in \delta(q_i, a_i)$ for each i and F is visited infinitely many times in this run.

From LTL Formulas to Büchi Automata. It is a classical result in model checking that for every LTL formula φ defined over atomic predicates AP there exists a non-deterministic Büchi automaton N with alphabet 2^{AP} which accepts exactly those traces that satisfy φ [60]. Here, 2^{AP} is the set of subsets of AP.

We now define the product of a transition system $\mathcal{T}$ and an NBW N over the alphabet 2^{AP}. Note that at every step of a run of a transition system, we are always at a unique location. Hence, if a letter $\alpha \in 2^{\mathsf{AP}}$ contains both $at(l)$ and $at(l')$ for two different locations $l \neq l'$, then it will never be realized by any run. Thus, without loss of generality, we can remove any transitions with such an α. A similar argument applies if α has no atomic proposition of the form $at(\cdot)$[*].

Product of a Transition System and a Büchi Automaton. Let $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ be a transition system and $N = (Q, 2^{\mathsf{AP}}, \delta, q_0, F)$ an NBW. The *product* of $\mathcal{T}$ and N results in a new transition system $\mathcal{T} \times N := (\mathcal{V}, L', l'_{init}, \theta_{init}, \rightsquigarrow)$ and a set of states $\mathcal{B}_N^{\mathcal{T}}$ in $\mathcal{T} \times N$ where:

- L' is the Cartesian product $L \times Q$.
- $l'_{init} = (l_{init}, q_0)$.
- For each transition $\tau = (l, l', G_\tau, U_\tau) \in \mapsto$, states $q, q' \in Q$ and $\alpha \in 2^{\mathsf{AP}}$ such that $q' \in \delta(q, \alpha)$ and $at(l)$ is the unique proposition of the form $at(\cdot)$ in α, we include a transition $\big((l, q), (l', q'), G_\tau \wedge \alpha^{-at}, U_\tau\big)$ in $\rightsquigarrow$. Here, α^{-at} is the conjunction of all atomic propositions in α except $at(l)$.
- $\mathcal{B}_N^{\mathcal{T}} = L \times F \times \mathbb{R}^n$, i.e. $\mathcal{B}_N^{\mathcal{T}}$ is the set of states in $\mathcal{T} \times N$ whose NBW state component is contained in F.

Let N be an NBW accepting the traces that satisfy φ. Then, in order to solve the ELTL-PA, it suffices to solve the EB-PA problem over the Cartesian product $\mathcal{T} \times N$ with the set $\mathcal{B}_N^{\mathcal{T}}$ as the Büchi specification:

LEMMA 4.1 (ELTL-PA TO EB-PA [63], PROOF IN APPENDIX A). *Let $\mathcal{T}$ be a transition system, φ an LTL formula for $\mathcal{T}$ and N an NBW that accepts the same language as φ. The ELTL-PA problem of $\mathcal{T}$ and φ is equivalent to EB-PA problem of $\mathcal{T} \times N$ and $\mathcal{B}_N^{\mathcal{T}}$.*

Example. Figure 4 (left) shows an example Büchi automaton N accepting the runs satisfying [G F $at(l_2)$] with q_1 as the only accepting node. Figure 4 (right) shows the product of the transition system in Figure 3 and N with the Büchi specification $\mathcal{B}_N^{\mathcal{T}}$ shown in gray.

[*]If $\delta(q, \alpha) = \emptyset$, i.e. if the transition is removed, then any run that is at q and reads the letter α will be a rejecting run. Equivalently, we can create a non-accepting sink state $\perp$ and let $\delta(q, \alpha) = \perp$. The sink state will only have self-loops, i.e. $\delta(\perp, \beta) = \perp$ for every $\beta \in 2^{\mathsf{AP}}$.

Remark. Based on the lemma above, instead of designing witnesses for the ELTL-PA problem, we only need to find sound and complete witnesses for EB-PA. Moreover, it is easy to see that ULTL-PA is reducible to ELTL-PA since all runs of $\mathcal{T}$ satisfy φ if and only if there is no run that satisfies $\neg\varphi$. So, finding sound and complete witnesses for EB-PA will theoretically solve both existential and universal variants of LTL-PA.

An analogous lemma exists to reduce ULTL-PA to UB-PA. However, the UB-PA problem of $\mathcal{T} \times N$ and $\mathcal{B}_N^T$ is stronger than the ULTL-PA of $\mathcal{T}$ and φ. Nevertheless, if we limit the LTL formula φ to admit a *deterministic* Büchi automaton (DBW), then we have the following lemma:

Lemma 4.2 (ULTL-PA to UB-PA, Proof in Appendix B). *Let $\mathcal{T}$ be a transition system, φ an LTL formula for $\mathcal{T}$ and D a DBW that accepts the same language as φ. The ULTL-PA problem of $\mathcal{T}$ and φ is equivalent to the UB-PA problem of $\mathcal{T} \times D$ and $\mathcal{B}_D^{\mathcal{T}}$.*

Remark. Lemma 4.2 is more restrictive than Lemma 4.1 since it only applies to deterministic Büchi automata. If the LTL formula φ in a ULTL-PA instance does not admit a deterministic Büchi automaton, then we should apply the sequence of reductions in the previous remark and first reduce it to ELTL-PA and then to EB-PA. However, if φ does admit a DBW, then the reduction of Lemma 4.2 is preferable in practice. Thus, we will provide witness concepts for both EB-PA and UB-PA. Moreover, note that DBWs are expressive enough for virtually all common verification tasks. Thus, Lemma 4.2 will be invoked more often in practice.

5 SOUND AND COMPLETE B-PA WITNESSES

Let $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ be a transition system and $\mathcal{B} \subseteq \mathcal{S}$ a set of states in $\mathcal{T}$. In this section, we introduce our sound and complete witnesses for the EB-PA and UB-PA problems.

5.1 Sound and Complete Witnesses for Existential B-PA

Our witness concept for the EB-PA problem is a function that assigns a real value to each state in $\mathcal{T}$. The witness function is required to be non-negative in at least one initial state of $\mathcal{T}$, to preserve non-negativity in at least one successor state and to strictly decrease in value in at least one successor state whenever the current state is contained in $\mathcal{B}$ and the value of the witness function in the current state is non-negative. Hence, starting in an initial state in which the witness function is non-negative, one can always select a successor state in which the witness function is non-negative and furthermore ensure that $\mathcal{B}$ is eventually reached due to the strict decrease condition, which will also be referred to as the *Büchi-ranking condition.*

Definition 5.1 (EBRF). Given two states $s_1, s_2 \in \mathcal{S}$, a function $f \colon \mathcal{S} \to \mathbb{R}$ is said to Büchi-rank (s_1, s_2) where $s_1 \mapsto s_2$, if it satisfies one of the following:

- $s_1 \in \mathcal{B} \wedge \left[f(s_1) \geq 0 \Rightarrow f(s_2) \geq 0\right]$; or
- $s_1 \notin \mathcal{B} \wedge \left[f(s_1) \geq 0 \Rightarrow 0 \leq f(s_2) \leq f(s_1) - 1\right]$.

f is called a *$\mathcal{B}$-Existential Büchi Ranking Function* ($\mathcal{B}$-EBRF) if it satisfies the following conditions:

- $\exists s_{init} \in \mathcal{S}_{init}$ where $f(s_{init}) \geq 0$.
- For every $s_1 \in \mathcal{S}$, there exists $s_2 \in \mathcal{S}$ such that $s_1 \mapsto s_2$ and (s_1, s_2) is Büchi-ranked by f.

Example. The following function is a $\{(l_1, q_1, *)\}$-EBRF for the transition system in Figure 4:

$$f(l, x_0) = \begin{cases} x_0 + 3 & \text{if } l = (l_{init}, q_0) \\ x_0 + 2 & \text{if } l = (l_1, q_0) \\ x_0 + 1 & \text{if } l = (l_2, q_0) \\ 0 & \text{if } l = (l_1, q_1) \\ -1 & \text{otherwise} \end{cases}.$$

For example, the state $s_0 = ((l_1, q_0), 1)$ has two successors in the transition system: $s_1 = ((l_2, q_0), 1)$ and $s_2 = ((l_3, q_0), 1)$. It is easy to see that $0 \leq f(s_1) \leq f(s_0) - 1$ which shows that transition from s_0 to s_1 is Büchi-ranked by f.

The following lemmas establish the soundness and completeness of EBRFs for the EB-PA problem. To prove them, we use $\mathcal{B}$-EBRF to exhibit a $\mathcal{B}$-Büchi run in $\mathcal{T}$.

Lemma 5.2 (Soundness). *If there exists a $\mathcal{B}$-EBRF f for $\mathcal{T}$, then the answer to the EB-PA problem on $\mathcal{T}$ and $\mathcal{B}$ is positive.*

Proof. Let $e_{init} \in \theta_{init}$ be such that $f(l_{init}, e_{init}) \geq 0$ and $(l, e) \in \mathcal{S}$ be any state of $\mathcal{T}$. By definition of EBRFs, there exists $(l', e') \in \mathcal{S}$ such that $(l, e) \mapsto (l', e')$ and f Büchi-ranks $\big((l, e), (l', e')\big)$. We define the memory-less scheduler σ as follows: $\sigma_f(\Lambda) = (l_{init}, e_{init})$ and $\sigma_f(l, e) = (l', e')$. Next we prove that σ_f is $\mathcal{B}$-Büchi by showing that $\pi = \pi_{\sigma_f}$ is $\mathcal{B}$-Büchi.

As f is an EBRF, $f(\pi(0)) \geq 0$ and it can be inductively proved that $f(\pi(i)) \geq 0$ for each $i \in \mathbb{N}$. Also, if $\pi(i) \notin \mathcal{B}$, then $f(\pi(i+1)) \leq f(\pi(i)) - 1$. Suppose, for the sake of contradiction, that π reaches $\mathcal{B}$ only finite number of times. Then there exists $k \in \mathbb{N}$ such that π^{k+} does not reach $\mathcal{B}$ at all, meaning that the value of f is non-negative at $\pi(k)$ and decreases by 1 infinitely while remaining non-negative. The contradiction shows that π is $\mathcal{B}$-Büchi. □

Lemma 5.3 (Completeness). *If there exists a $\mathcal{B}$-Büchi scheduler σ for $\mathcal{T}$, then there exists a $\mathcal{B}$-EBRF for $\mathcal{T}$.*

Proof. Suppose that there exists a $\mathcal{B}$-Büchi scheduler σ for $\mathcal{T}$. To prove completeness, we first observe that if there exists a $\mathcal{B}$-Büchi scheduler σ for $\mathcal{T}$, then there also exists a memory-less $\mathcal{B}$-Büchi scheduler for $\mathcal{T}$. Due to space restrictions, we defer the proof of this claim to Appendix C.

Let σ be a memory-less $\mathcal{B}$-Büchi scheduler for $\mathcal{T}$. It gives rise to a $\mathcal{B}$-EBRF f for $\mathcal{T}$ as follows: Let π_σ be the run in $\mathcal{T}$ induced by σ. Since σ is $\mathcal{B}$-Büchi, the run π_σ must visit states in $\mathcal{B}$ infinitely many times. Thus, for every $i \in \mathbb{N}_0$, let $dist_{\pi_\sigma}(i)$ be the smallest number $k \in \mathbb{N}_0$ such that $\pi_\sigma(i+k) \in \mathcal{B}$. We define f as follows: $f(l, e) = \begin{cases} -1, & \text{if } (l, e) \notin \pi_\sigma \\ dist_{\pi_\sigma}(i), & \text{if } (l, e) = \pi_\sigma(i) \end{cases}$. In other words, f is equal to -1 in every state that is not contained in the run π_σ, and for every state contained in π_σ we define f to be equal to the number of subsequent states in π_σ before the next visit to $\mathcal{B}$. One can easily check by insepction that this function satisfies the defining conditions of EBRFs. □

Lemmas 5.2 and 5.3 together imply the following theorem, which is the main result of this section and establishes soundness and completeness of EBRFs for the EB-PA problem. Hence, since we showed in Lemma 4.1 that one can reduce the ELTL-PA problem to EB-PA, as a corollary it also follows that EBRFs provide sound and complete certificates for ELTL-PA.

Theorem 5.4 (Soundness and Completeness of EBRFs for EB-PA). *There exists a $\mathcal{B}$-EBRF f for $\mathcal{T}$ with Büchi specification $\mathcal{B}$ if and only if the answer to the EB-PA problem of $\mathcal{T}$ and $\mathcal{B}$ is positive.*

Corollary 5.5. *The answer to the ELTL-PA problem of $\mathcal{T}$ and φ is positive if and only if there exists a $\mathcal{B}_N^{\mathcal{T}}$-EBRF for $\mathcal{T} \times N$, where N is the NBW accepting φ.*

5.2 Sound and Complete Witnesses for Universal B-PA

Similarly to EBRFs, we can define a witness function for the UB-PA problem. The difference compared to EBRFs is that we now impose the Büchi ranking condition for *every* successor state of a state in which the witness function is non-negative. In contrast, in EBRFs we imposed the Büchi ranking condition only for *some* successor state.

Definition 5.6 (UBRF). A function $f\colon \mathcal{S} \to \mathbb{R}^n$ is called a *$\mathcal{B}$-Universal Büchi Ranking Function* ($\mathcal{B}$-UBRF) if it satisfies the following conditions:

- $f(s) \geq 0$ for **every** $s \in \mathcal{S}_{init}$
- For **every** $s_1, s_2 \in \mathcal{S}$ such that $s_1 \mapsto s_2$, (s_1, s_2) is Büchi-ranked by f.

We have the following theorem, which establishes that UBRFs provide a sound and complete certificate for the UB-PA problem. The proof is similar to the existential case and thus deferred to Appendix D. The subsequent corollary then follows from Lemma 4.2 which shows that the ULTL-PA problem can be reduded to the UB-PA problem if φ admits a deterministic Büchi automaton.

Theorem 5.7 (Soundness and Completeness of UBRFs for UB-PA). *There exists a $\mathcal{B}$-UBRF f for $\mathcal{T}$ with Büchi specification $\mathcal{B}$ if and only if the answer to the UB-PA problem of $\mathcal{T}$ and $\mathcal{B}$ positive.*

Corollary 5.8. *If φ is an LTL formula that admits a DBW D, the answer to the ULTL-PA problem of $\mathcal{T}$ and φ is positive if and only if there exists a $\mathcal{B}_D^{\mathcal{T}}$-UBRF for $\mathcal{T} \times D$.*

6 TEMPLATE-BASED SYNTHESIS OF POLYNOMIAL WITNESSES

We now present our fully automated algorithms to synthesize polynomial EBRFs and UBRFs in polynomial transition systems. A transition system $\mathcal{T}$ is said to be *polynomial* if guards and updates of all transitions in $\mathcal{T}$ are polynomial expressions over program variables $\mathcal{V}$. Given a polynomial transition system $\mathcal{T}$ and a Büchi specification $\mathcal{B}$, which was obtained from an LTL formula as above, our approach synthesizes polynomial EBRFs and UBRFs of any desired degree, assuming that they exist. Our algorithms follow a template-based synthesis approach, similar to the methods used for reachability and termination analysis [33, 38]. In particular, both EBRF and UBRF synthesis algorithms first fix a symbolic polynomial template function for the witness at each location in $\mathcal{T}$. The defining conditions of EBRFs/UBRFs are then expressed as entailment constraint of the form

$$\exists c \in \mathbb{R}^m \ \ \forall e \in \mathbb{R}^n \ \ (\phi \Rightarrow \psi), \tag{1}$$

where ϕ and ψ are conjunctions of polynomial inequalities. We show that this translation is sound and complete. However, such constraints are notoriously difficult to solve due to the existence of a quantifier alternation. Thus, we use the sound and semi-complete technique of [33] to eliminate the quantifier alternation and translate our constraints into a system of purely existentially quantified quadratic inequalities. Finally, this quadratic programming instance is solved by an SMT solver. We note that a central technical difficulty here is to come up with sound and complete witness notions whose synthesis can be reduced to solving entailment constraints of the form (1). While [33, 38] achieved this for termination and reachability, our EBRF and UBRF notions significantly extend these results to arbitrary LTL formulas.

As is common in static analysis tasks, we assume that the transition system comes with an invariant θ_l at every location l in $\mathcal{T}$. Invariant generation is an orthogonal and well-studied problem. In our setting of polynomial programs, the invariant can be automatically generated using the tools in [38, 77, 78]. Alternatively, one can encode the inductive invariant as part of the constraints of the form (1). This has the extra benefit of ensuring that we always find an invariant that leads to a witness for our LTL formula, if such a witness exists, and thus do not sacrifice completeness due to potentially loose invariants. See [38] for details of the encoding. This is the route we took in our tool, i.e. our tool automatically generates the invariants it requires using the sound and complete method of [38]. For brevity, we removed the invariant generation part from the description of the algorithms below.

Synthesis of Polynomial EBRFs. We now present our algorithm for synthesis of a polynomial EBRF, given a polynomial transition system $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ and Büchi specification $\mathcal{B}$ obtained from an LTL formula with polynomial inequalities in AP. The algorithm has five steps:

(1) *Fixing Symbolic Templates.* Let $M_{\mathcal{V}}^D = \{m_1, m_2, \ldots, m_k\}$ be the set of all monomials of degree at most D over the set of variables $\mathcal{V}$. In the first step, the algorithm generates a symbolic

Location	Template	Location	Template
(l_{init}, q_0)	$c_{init,0,0} + c_{init,0,1} \cdot x_0 + c_{init,0,2} \cdot x_0^2$	(l_2, q_0)	$c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2$
(l_1, q_0)	$c_{1,0,0} + c_{1,0,1} \cdot x_0 + c_{1,0,2} \cdot x_0^2$	(l_3, q_0)	$c_{3,0,0} + c_{3,0,1} \cdot x_0 + c_{3,0,2} \cdot x_0^2$
(l_1, q_1)	$c_{1,1,0} + c_{1,1,1} \cdot x_0 + c_{1,1,2} \cdot x_0^2$	(l_t, q_0)	$c_{t,0,0} + c_{t,0,1} \cdot x_0 + c_{t,0,2} \cdot x_0^2$

Table 1. The template EBRF generated for the transition system in Figure 4

polynomial template for the EBRF at each location $l \in L$ as follows: $f_l(x) = \Sigma_{i=1}^{k} c_{l,i} \cdot m_i$. Here, all the c-variables are fresh symbolic template variables that represent the coefficients of polynomial expressions in f. The goal of our synthesis procedure is to find a concrete valuation of c variables for which f becomes a valid $\mathcal{B}$-EBRF for $\mathcal{T}$.

Example. Consider the transition system in Figure 4 and suppose $D = 2$, i.e. the goal of the algorithm is to generate a polynomial EBRF of degree at most 2 for the given transition system. Table 1 shows the template generated for each location. Note that the $c_{l,i}$ variables are treated as unknowns and the goal of the algorithm is to find suitable valuations for them so that they create a $\mathcal{B}$-EBRF.

(2) *Generating Entailment Constraints.* For every location $l \in L$ and variable valuation $x \models \theta_l$, there must exist an outgoing transition τ such that $x \models G_\tau$ and τ is Büchi-ranked by f in x. The algorithm symbolically writes this condition down as an entailment constraint: $\forall x \in \mathbb{R}^n \;\; x \models (\phi_l \Rightarrow \psi_l)$ with ϕ_l and ψ_l symbolically computed as follows: $\phi_l := \theta_l \wedge f_l(x) \geq 0$ and $\psi_l \equiv \bigvee_{\tau \in Out_l} G_\tau \wedge \mathcal{B}\text{–}Rank(\tau)$, where for each $\tau = (l, l', G_\tau, U_\tau)$ the predicate $\mathcal{B}\text{–}Rank$ is defined as follows:

$$\mathcal{B}\text{–}Rank(\tau) \equiv \begin{cases} f_{l'}(U_\tau(x)) \geq 0 \wedge f_{l'}(U_\tau(x)) \leq f_l(x) - 1 & l \notin \mathcal{B} \\ f_{l'}(U_\tau(x)) \geq 0 & l \in \mathcal{B} \end{cases}$$

The algorithm then writes ψ_l in disjunctive normal form as $\vee_{i=1}^{k} \psi_{l,i}$. Next, the algorithm rewrites $\phi_l \Rightarrow \psi_l$ equivalently as:

$$(\phi_l \wedge \textstyle\bigwedge_{i=1}^{k-1} \neg \psi_{l,i}) \Rightarrow \psi_{l,k} \tag{2}$$

This rewriting makes sure that we can later manipulate the constraint in (2) to fit in the standard form of (1)[†]. Intuitively, (2) ensures that whenever l was reached and each of the first $k-1$ outgoing transitions were either unavailable or not Büchi-ranked by f, then the last transition has to be available and Büchi-ranked by f. Our algorithm populates a list of all constraints and adds the constraint (2) to this list before moving to the next location and repeating the same procedure. Note that in all of the generated constraints of the form (2), both the LHS and the RHS of the entailment are boolean combinations of polynomial inequalities over program variables.

Example (Continued). We give the entailment constraints for location (l_2, q_0). Entailment constraints from other locations can be derived similarly.
Let $\theta_{(2,0)} \equiv x_0 \geq 0$ be the invariant for (l_2, q_0). The algorithm symbolically computes the following:

$$\begin{gathered} \left[x_0 \geq 0 \wedge c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2 \geq 0\right] \Rightarrow \\ \left[c_{1,1,0} + c_{1,1,1} \cdot (x_0^2+1) + c_{1,1,2} \cdot (x_0^2+1)^2 \geq 0 \;\; \wedge \right. \\ \left. c_{1,1,0} + c_{1,1,1} \cdot (x_0^2+1) + c_{1,1,2} \cdot (x_0^2+1)^2 \leq c_{2,0,0} + c_{2,0,1} \cdot x_0 + \cdot c_{2,0,2} \cdot x_0^2 - 1\right] \end{gathered} \tag{3}$$

This is to formulate that whenever a state $s = (l_2, q_0, x_0)$ is reached and $f(s)$ is non-negative, there exists a successor state s' such that $0 \leq f(s') \leq f(s) - 1$. As (l_2, q_0) has only one successor in $\mathcal{T}$, this has the same format as (2).

[†]We have to find values for c-variables that satisfy all these constraints conjunctively. This is why we have an extra existential quantifier in (1).

(3) *Reduce Constraints to Quadratic Inequalities.* To solve the constraints generated in the previous step, we directly integrate the technique of [33] into our algorithm. This is a sound and semi-complete approach based on Putinar's Positivstellensatz. We will provide an example below, but refer to [33] for technical details and proofs of soundness/completeness of this step.

In this step, for each constraint of the form $\Phi \Rightarrow \Psi$, the algorithm first rewrites Φ in disjunctive normal form as $\phi_1 \vee \cdots \vee \phi_t$ and Ψ in conjunctive normal form as $\Psi \equiv \psi_1 \wedge \cdots \wedge \psi_r$. Then for each $1 \le i \le t$ and $1 \le j \le r$ the algorithm uses Putinar's Positivstellensatz in the exact same way as in [33] to generate a set of quadratic inequalities equivalent to $\phi_i \Rightarrow \psi_j$. The algorithm keeps track of a quadratic programming instance Γ and adds these new inequalities to it conjunctively.

Example (Continued). The algorithm symbolically computes the following entailments as $\phi_i \Rightarrow \psi_j$ expressions from (3):

$$(i)\ \ x_0 \ge 0 \wedge c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2 \ge 0 \Rightarrow c_{1,1,0} + c_{1,1,1} \cdot (x_0^2 + 1) + c_{1,1,2} \cdot (x_0^2 + 1)^2 \ge 0$$

$$(ii)\ x_0 \ge 0 \wedge c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2 \ge 0 \Rightarrow$$
$$c_{1,1,0} + c_{1,1,1} \cdot (x_0^2 + 1) + c_{1,1,2} \cdot (x_0^2 + 1)^2 \le c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2 - 1$$

We show how the first constraint above is handled. The goal is to write the RHS of (i) as the sum of several non-negative expressions derived from multiplying the LHS expressions by sum-of-squares (SOS) polynomials. The idea is that SOS polynomials are always non-negative and the polynomials on the LHS are also assumed to be non-negative, so if we can combine them to achieve the RHS, then the constraint is proven[‡]. To do this, the algorithm generates three template SOS polynomials. For simplicity, suppose they are of the form $\lambda_{i,0} + \lambda_{i,1} \cdot x_0^2$ for $i = 0, 1, 2$ where all $\lambda_{i,j}$'s are non-negative. Next, it symbolically computes the following equality:

$$\begin{aligned}
&(\lambda_{0,0} + \lambda_{0,1} \cdot x_0^2)\\
&+ (\lambda_{1,0} + \lambda_{1,1} \cdot x_0^2) \cdot (x_0)\\
&+ (\lambda_{2,0} + \lambda_{2,1} \cdot x_0^2) \cdot (c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2)\\
&= c_{1,1,0} + c_{1,1,1} \cdot (x_0^2 + 1) + c_{1,1,2} \cdot (x_0^2 + 1)^2
\end{aligned} \tag{4}$$

The polynomial equality in (4) has to be satisfied for all values of the program variable x_0. This means the monomials x_0^4, x_0^3, x_0^2, x_0, and 1 should have the same coefficients on both sides. Equating the coefficients, the algorithm obtains the following quadratic constraints which are equivalent to the original constraint:

$$\begin{aligned}
\lambda_{0,0} + \lambda_{2,0} \cdot c_{2,0,0} &= c_{1,1,0} + c_{1,1,1} + c_{1,1,2}\\
\lambda_{1,0} + \lambda_{2,0} \cdot c_{2,0,1} &= 0\\
\lambda_{0,1} + \lambda_{2,0} \cdot c_{2,0,2} + \lambda_{2,1} \cdot c_{2,0,0} &= c_{1,1,1} + 2c_{1,1,2}\\
\lambda_{1,1} + \lambda_{2,1} \cdot c_{2,0,1} &= 0\\
\lambda_{2,1} \cdot c_{2,0,2} &= c_{1,1,2}
\end{aligned}$$

(4) *Handling Initial Conditions.* Additionally, for every variable $x \in \mathcal{V}$, the algorithm introduces another symbolic template variable t_x, modeling the initial value of x in the program, and adds the constraint $[\theta_{init}(t) \wedge f_{l_{init}}(t) \ge 0]$ to Γ to impose that there exists an initial state in $\mathcal{T}$ at which the value of the EBRF f is non-negative.

[‡]This method is not only sound, but also complete. This is due to Putinar's Positivstellensatz [79]. See [33] for details.

Example (Continued). To ensure existence of $s_{init} \in \mathscr{S}_{init}$ such that the generated EBRF has non-negative value on s_{init} (as required by the definition of an EBRF), the algorithm adds $[t_{x_0} \geq 0 \wedge f_{(init,0)}(t_{x_0}) \geq 0]$ to Γ.

(5) *Solving the System.* Finally, the algorithm uses an external solver (usually an SMT solver) to compute values of t and c variables for which Γ is satisfied. If the solver succeeds in solving the system of constraints Γ, the computed values of c and t variables give rise to a concrete instance of an $\mathcal{B}$-EBRF for $\mathcal{T}$. This implies that the answer to the EB-PA problem is positive, and the algorithm return "Yes". Otherwise, the algorithm returns "Unknown", as there might exist a $\mathcal{B}$-EBRF for $\mathcal{T}$ of higher maximum polynomial degree D or a non-polynomial $\mathcal{B}$-EBRF.

Example (Continued). The EBRF shown in the Example of Section 5 satisfies the above constraints and corresponds to the following solution to the QP instance Γ :

$$\begin{array}{lllllll} c_{0,0,0}=3 & c_{0,0,1}=1 & c_{0,0,2}=0 & c_{1,0,0}=2 & c_{1,0,1}=1 & c_{1,0,2}=0 & \\ c_{2,0,0}=1 & c_{2,0,1}=1 & c_{2,0,2}=0 & c_{1,1,0}=0 & c_{1,1,1}=0 & c_{1,1,2}=0 & \\ c_{3,0,0}=-1 & c_{3,0,1}=0 & c_{3,0,2}=0 & c_{t,0,0}=-1 & c_{t,0,1}=0 & c_{t,0,2}=0 & t_{x_0}=0 \end{array}$$

This solution was obtained by passing Γ to the Z3 SMT solver.

Theorem 6.1 (Existential Soundness and Semi-Completeness). *The algorithm above is a sound and semi-complete reduction to quadratic programming for the problem of synthesis of an EBRF in a polynomial transition system $\mathcal{T}$ given a Büchi specification $\mathcal{B}$ obtained from an LTL formula with polynomial inequalities in* AP. *Moreover, for any fixed D, the algorithm has sub-exponential complexity.*

Proof. Steps 1 and 2 of the algorithm are clearly sound and complete since they simply encode the EBRF conditions in a specific equivalent format. The same applies to the initial conditions in Step 4. See [33] for details of Step 3 and its soundness and semi-completeness. We are using the technique of [33] as a black box in our Step 3 and hence the same arguments apply in our case. Our algorithm inherits semi-completeness and its dependence on polynomial degrees from [33].

For any fixed degree of the template polynomials, our algorithm above provides a PTIME reduction from the problem of synthesizing EBRFs/URBFs to Qudratically-constrained Quadratic Programming (QP). It is well-known that QP is solvable in sub-exponential time [80]. Thus, the complexity of our approach is sub-exponential, too. □

In the above theorem, soundness means that every solution to the QP instance is a valid EBRF and semi-completeness means that if a polynomial EBRF exists and the chosen maximum degree D is large enough, then the QP instance will have a solution.

In practice, we simply pass the QP instance to an SMT solver. Since it does not include a quantifier alternation, the SMT solvers have dedicated heuristics and are quite efficient on QP instances.

Synthesis of Polynomial UBRFs. Our algorithm for synthesis of UBRFs is almost the same as our EBRF algorithm, except that the constraints generated in Steps 2 and 4 are slightly different.

Changes to Step 2. Step 2 is the main difference between the two algorithms. In this step, for each location $l \in L$ and each transition $\tau \in Out_l$ the UBRF algorithm adds $(\phi_{l,\tau} \Rightarrow \psi_{l,\tau})$ to the set of constraints, where we have $\phi_{l,\tau} \equiv \theta_l \wedge G_\tau \wedge f_l(x) \geq 0$ and $\psi_{l,\tau} \equiv \mathcal{B}\text{–}Rank(\tau)$. The intuition behind this step is that whenever a transition is enabled, it has to be Büchi-ranked by f.

Changes to Step 4. In this step, instead of searching for a suitable initial valuation for program variables, the algorithm adds the quadratic inequalities equivalent to $(\theta_{init} \Rightarrow f_{l_{init}}(x) \geq 0)$ to Γ. The quadratic inequalities are obtained exactly as in Step 3. This is because the value of the UBRF must be non-negative on every initial state of the transition system.

In the universal case, we have a similar theorem of soundness and semi-completeness whose proof is exactly the same as Theorem 6.1.

Theorem 6.2 (Universal Soundness and Semi-Completeness). *The algorithm above is a sound and semi-complete reduction to quadratic programming for the problem of synthesis of an UBRF in a polynomial transition system $\mathcal{T}$ given a Büchi specification $\mathcal{B}$ obtained from an LTL formula with polynomial inequalities in* AP. *Moreover, for any fixed D, the algorithm has sub-exponential complexity.*

7 EXPERIMENTAL RESULTS

General Setup of Experiments. We implemented a prototype of our UBRF and EBRF synthesis algorithms in Java and used Z3 [81], Barcelogic [82] and MathSAT5 [83] to solve the generated systems of quadratic inequalities. More specifically, after obtaining the QP instance, our tool calls all three SMT solvers in parallel. We also used ASPIC [77] for invariant generation for benchmarks that are linear programs. All experiments were performed on a Debian 11 machine with a 2.60GHz Intel E5-2670 CPU and 6 GB of RAM with a timeout of 1800 seconds.

Baselines. We compare our tool with Ultimate LTLAutomizer [63], nuXmv [67], and MuVal [44] as well as with a modification of our method that instead of using Putinar's Positivstellensatz simply passes entailment constraints to the SMT-solver Z3 [81]:

- Ultimate LTLAutomizer makes use of "Büchi programs", which is a similar notion to our product of a transition system and a Büchi Automaton, to either prove that every lasso shaped path in the input program satisfies the given LTL formula, or find a path that violates it. However, in contrast to our tool, it neither supports non-linear programs nor provides completeness.
- nuXmv is a symbolic model checker with support for finite and infinite transition systems. It allows both existential and universal LTL program analysis and supports non-linear programs. It does not provide any completeness guarantees.
- MuVal [44] is a fixed-point logic validity checker based on pfwCSP solving [43]. It supports both linear and non-linear programs with integer variables and recursive functions.
- When directly applying Z3, instead of the dedicated quantifier elimination method (Step 3 of our algorithm), we directly pass the quantified formula (1) to the solver, which will in turn apply its own generic quantifier elimination. This is an ablation experiment to check whether the technique of Step 3 is needed in practice.

Benchmarks. We gathered benchmarks from two sources:

- 297 benchmarks from the "Termination of C-Integer Programs" category of TermComp'22 [84][§]. Among these, 287 programs only contained linear arithmetic which is supported by all comparator tools, whereas 10 programs (Table 4) contained polynomial expressions not supported by Ultimate.
- 21 non-linear benchmarks from the "ReachSafety-Loops `nl-digbench`" category of SV-COMP'22 [85][¶]. As these benchmarks are all non-linear, none of them are supported by Ultimate.

LTL specifications. We used the four LTL specifications shown in Table 2. In all four considered specifications, x represents the alphabetically first variable in the input program. The motivation behind the choice of our specifications is as follows:

[§]There were originally 355 benchmarks, but we had to remove benchmarks with unbounded non-determinism and those without any variables, since they cannot be translated to transition systems and are not supported in our setting.

[¶]The original benchmark set contains 28 programs, but 7 of them contain unsupported operators such as integer mod and are thus not expressible in our setting.

Name	Formula	Pre-condition θ_{init}
RA	$(F\ at(l_{term})) \wedge (G\ x \geq 0)$	$\forall x \in \mathcal{V}, 0 \leq x \leq 64$
OV	$F\ (at(l_{term}) \vee x < -64 \vee x > 63)$	$\forall x \in \mathcal{V}, -64 \leq x \leq 63$
RC	$G\ F\ (x \geq 0)$	$\forall x \in \mathcal{V}, -64 \leq x \leq 63$
PR	$G\ (x < -5 \Rightarrow F\ (x > 0))$	$\forall x \in \mathcal{V}, -64 \leq x \leq 63$

Table 2. LTL specifications used in our experiments.

Formula	Ours				Ultimate				nuXmv				MuVal				Z3			
	Yes	No	Tot.	U.	Yes	No	Tot.	U.	Yes	No	Tot.	U.	Yes	No	Tot.	U.	Yes	No	Tot.	U.
RA	141	114	255	5	142	121	263	7	76	91	137	0	118	76	194	0	56	36	92	0
OV	199	47	246	4	212	55	267	5	110	50	160	0	205	47	252	3	48	27	75	0
RC	87	187	274	0	86	194	280	0	83	183	266	0	86	191	277	0	44	71	115	0
PR	43	222	265	1	45	237	282	0	44	227	271	0	42	235	277	0	29	77	106	0
Avg. Runtime (s)	5.4	81.5	47.2	-	5.4	4.1	4.7	-	248.9	13.5	98.7	-	48.8	8.43	26.4	-	18.5	160.6	95.7	-

Table 3. Experimental results on linear benchmarks. We report in how many cases the tool could successfully prove the formula (Yes) or refute it (No), total number of cases proved by the tool (Tot.), number of instances uniquely solved by each tool and no other tools (U.), and average runtime of each tool on programs that were successfully proved as correct with respect to each specification.

- *Reach-avoid (RA) specifications.* The first specification is an example of reach-avoid specifications that we discussed in Section 3. In particular, the reach-avoid specification that we consider in our experimental evaluation specifies that a program run should terminate without ever making x negative. As discussed in Section 3, reach-avoid specifications are standard in the analysis of dynamical and hybrid systems [70–72]. Another common example of reach-avoid specifications is requiring program termination while satisfying all assertions specified by the programmer.
- *Overflow (OV) specifications.* Intuitively, we want to evaluate whether our approach is capable of detecting variable overflows. The second specification specifies that each program run either terminates or the value of the variable x overflows. Specifically, suppose that an overflow is handled as a runtime error and ends the program. The negation (refutation) of this specification models the existence of a run that neither terminates nor overflows and thus converges.
- *Recurrence (RC) specifications.* The third specification is an instance of a recurrence specification, which is an important class of specifications specifying that a program run visits a set of states infinitely many times [86]. This requires that a program run contains infinitely many visits to states in which x has a non-negative value.
- *Progress (PR) specifications.* The fourth specification is an example of progress specifications that we discussed in Section 3. The progress specification that we consider in our experimental evaluation specifies that a program run always makes progress from states in which the value of x is less than -5 to states in which the value of x is strictly positive.

Results on Linear Programs. Table 3 summarizes our results over linear benchmarks to which all tools are applicable. First, we observe that in all cases our tool outperforms the method that uses Z3 for quantifier elimination, showing that our Step 3 is a crucial and helpful part of the algorithm. Compared to nuXmv, our tool proves more instances in all but two existential and one universal LTL program analysis cases, i.e. the "No" column for the OV and PR specifications and the "Yes" column for the PR specification. On the other hand, our prototype tool is on par with Ultimate and MuVal, while proving 10 unique instances. Note that Ultimate is a state of the art and well-maintained competition tool that is highly optimized with heuristics that aim at the linear case. In contrast, it cannot handle polynomial instances. Our results shown in Table 3 demonstrate that our prototype tool is very competitive already on linear benchmarks, even though our main contribution is to provide practically-efficient semi-complete algorithms for the polynomial case.

	Benchmark	Ours				nuXmv				MuVal			
		RA	*OV*	*RC*	*PR*	*RA*	*OV*	*RC*	*PR*	*RA*	*OV*	*RC*	*PR*
TermComp Benchmarks	`...2008-aaron12 true-termination.c`	F	✔	✗	✗	F	F	✗	✗	✗	F	✗	✗
	`ComplInterv.c`	✗	✔	✗	✗	F	F	✗	F	F	✔	✗	F
	`DoubleNeg.c`	✔	✔	✗	✗	F	F	✗	✗	✔	✔	✗	✗
	`Factorial.c`	✗	F	✔	✔	F	✔	F	F	F	✔	✔	✔
	`LogMult.c`	✔	✔	✔	✔	F	✔	✔	✔	✔	✔	✔	✔
	`svcomp_ex1.c`	✔	✔	F	F	F	✔	✗	✗	✔	✔	✗	✗
	`svcomp_ex2.c`	F	✔	✔	✔	F	✔	✔	✔	✔	✔	✔	F
	`svcomp_ex3a.c`	✔	✔	✔	✗	F	✔	✔	✗	✔	✔	✔	✗
	`svcomp_ex3b.c`	✔	✔	✔	✗	F	✔	✔	✗	✔	✔	✔	✗
	`svcomp_fermat.c`	✔	✔	✔	✔	F	F	F	F	F	✔	✔	✔
SV-Comp Benchmarks	`bresenham-ll.c`	✔	✔	✗	✗	F	F	F	F	F	✔	✗	✗
	`cohencu-ll.c`	✔	✔	✗	✗	F	F	✗	✗	F	✔	F	F
	`cohendiv-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`dijkstra-u.c`	✔	✔	✔	F	F	F	✔	✗	F	✔	F	✗
	`divbin.c`	F	✔	F	F	F	F	F	F	F	F	F	F
	`egcd2-ll.c`	✔	F	F	✔	F	F	✗	✔	F	F	F	F
	`egcd3-ll.c`	✔	F	F	✔	F	F	✗	✔	F	F	F	F
	`egcd-ll.c`	✗	✔	✗	✗	F	F	✗	✗	F	✔	F	F
	`geo1-ll.c`	✔	✔	✔	✔	F	F	✔	✔	✔	✔	✔	✔
	`geo2-ll.c`	✔	✔	✔	✔	F	F	✔	✔	✔	✔	✔	✔
	`geo3-ll.c`	✔	✔	✗	✗	F	F	✗	✗	F	F	✗	✗
	`hard-ll.c`	✔	F	F	F	F	F	F	F	✔	✔	✗	✗
	`lcm1.c`	F	F	✔	✔	F	F	✔	✔	F	F	F	F
	`lcm2.c`	✔	✔	✔	✔	F	F	✔	✔	✔	✔	✔	F
	`mannadiv.c`	✔	✔	✔	✔	F	F	✔	✔	✔	✔	✔	F
	`ps2-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`ps3-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`ps4-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`ps5-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`ps6-ll.c`	✔	✔	✔	✗	F	F	✔	✗	✔	✔	✔	✗
	`sqrt1-ll.c`	✔	✔	✔	✗	✔	✔	✔	✗	✔	✔	✔	✗
	Total Successful Instances	27	26	26	27	1	7	26	25	19	25	24	21
	Unique Instances	8	2	0	1	0	0	2	0	1	0	1	1
	Average Time (s)	26.0	29.2	20.2	53.4	16.6	164.8	0.2	0.3	216.5	43.9	71.3	26.5

Table 4. Experimental results on polynomial benchmarks. ✔ denotes successful proof of the property and ✗ denotes successful refutation. F means the tool failed to decide the LTL specification.

Unique Instances. An important observation is that our tool successfully handles 10 unique *linear* instances that no other tool manages to prove or refute. Thus, our evaluation shows that our method handles not only polynomial, but even linear benchmarks that were beyond the reach of the existing methods. This shows that our algorithm, besides the desired theoretical guarantee of semi-completeness, provides an effective automated method. Future advances in invariant generation and SMT solving will likely improve the performance of our method even further.

Runtimes. Our tool and Ultimate are the fastest tools for proving universal LTL analysis instances with an equal average runtime of 5.4 seconds. For existential LTL analysis, our tool is slower than other tools. However, note that there are many heuristics that can be added to our method for ELTL-PA and we only implemented our algorithm in Section 6 without any heuristics.

Results on Non-Linear Programs. Table 4 shows the performance of our tool, nuXmv and MuVal on the non-linear benchmarks. Ultimate does not support non-linear arithmetic and Z3 timed out on every benchmark in this category. Here, compared to nuXmv, our tool succeeded in solving strictly more instances in all but one formula, i.e. *RC*, where both tools solve the same number of instances. In comparison with MuVal, our tool proves more instances for all four formulas. Moreover, the fact that Z3 timed out for every program in this table is further confirmation of the practical necessity of Step 3 (Quantifier Elimination Procedure of [33]) in our algorithm. Note that our prototype could prove 11 instances that none of the other tools could handle.

Summary. Our experiments demonstrate that our automated algorithms are able to synthesize both universal and existential LTL witnesses for a wide variety of programs. Our technique clearly outperforms the previous methods when given non-linear polynomial programs (Table 4). Additionally, even in the much more widely-studied case of linear programs, we are able to handle instances that were beyond the reach of previous methods and to solve the number of instances that is close to the state-of-the-art tools (Table 3).

8 DISCUSSION AND CONCLUSION

Limitations and Future Directions. One limitation of our synthesis algorithm is that it is only applicable to polynomial programs and witnesses, since it uses theorems from real algebraic geometry for its quantifier elimination phase. On the one hand, generalizing the analysis to non-polynomial programs is an interesting extension of our work. On the other hand, using the Stone–Weierstrass theorem [39] to approximate non-polynomial program behavior and then applying our method is another future direction. Another limitation is that our method only supports finitely-branching non-determinism for universal LTL. Our existential approach does not require this and handles arbitrary infinitely-branching non-deterministic behavior.

Finally, our method is restricted to numerical programs and does not allow heap-manipulating operations. A common approach to handling heap-manipulating operations is to construct numerical abstractions of programs [87, 88] and perform the analysis on numerical abstractions. Thus, coupling such approaches, e.g. [75], with our method is a compelling future direction.

Concluding Remarks. We presented a novel family of sound and complete witnesses for template-based LTL verification. Our approach is applicable to both universal (all-paths) and existential (some-path) model checking. It unifies and significantly generalizes previous works targeting special cases of LTL, e.g. termination, safety and reachability. We also showed that our LTL witnesses can be synthesized in a sound and semi-complete manner by a reduction to quadratic programming. Our reduction works when the program and the witness are both polynomial. This generalizes previous algorithms for linear and polynomial programs with termination, safety, and reachability properties.

REFERENCES

[1] Robert W Floyd. Assigning meanings to programs. *Program Verification: Fundamental Issues in Computer Science*, pages 65–81, 1993.

[2] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[3] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 2012.

[4] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[5] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[6] Javier Esparza, Jan Kretínský, Jean-François Raskin, and Salomon Sickert. From LTL and limit-deterministic büchi automata to deterministic parity automata. In *TACAS*, pages 426–442, 2017.

[7] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[8] Pierre Wolper. Temporal logic can be more expressive. In *FOCS*, pages 340–348, 1981.

[9] Xavier Thirioux. Simple and efficient translation from LTL formulas to buchi automata. In *FMICS*, pages 145–159, 2002.

[10] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In *CAV*, pages 248–263, 2000.

[11] Lai-Xiang Shan, Xiaomin Du, and Zheng Qin. Efficient approach of translating LTL formulae into büchi automata. *Frontiers Comput. Sci.*, 9(4):511–523, 2015.

[12] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In *CAV*, pages 53–65, 2001.

[13] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to büchi automata. In *FORTE*, pages 308–326, 2002.

[14] Jacek Cichon, Adam Czubak, and Andrzej Jasinski. Minimal büchi automata for certain classes of LTL formulas. In *DepCoS-RELCOMEX*, pages 17–24, 2009.

[15] Irina V. Shoshmina and Alexey B. Belyaev. Symbolic algorithm for generation büchi automata from LTL formulas. In *PaCT*, pages 98–109, 2011.

[16] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In *TACAS*, pages 95–109, 2012.

[17] Shohei Mochizuki, Masaya Shimakawa, Shigeki Hagihara, and Naoki Yonezaki. Fast translation from LTL to büchi automata via non-transition-based automata. In *ICFEM*, pages 364–379, 2014.

[18] Salomon Sickert and Jan Kretínský. Mochiba: Probabilistic LTL model checking using limit-deterministic büchi automata. In *ATVA*, pages 130–137, 2016.

[19] Simon Jantsch, David Müller, Christel Baier, and Joachim Klein. From LTL to unambiguous büchi automata via disambiguation of alternating automata. In *FM*, pages 262–279, 2019.

[20] Arthur J. Bernstein and Paul K. Harter. Proving real-time properties of programs with temporal logic. In *SOSP*, pages 1–11, 1981.

[21] Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *TOPLAS*, 4(3):455–495, 1982.

[22] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *POPL*, pages 265–276, 2007.

[23] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. In *POPL*, pages 26:1–26:33, 2018.

[24] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[25] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In *CAV*, pages 271–291, 2016.

[26] Alessandro Cimatti, Alberto Griggio, and Enrico Magnago. LTL falsification in infinite-state systems. *Inf. Comput.*, 289:104977, 2022.

[27] Byron Cook, Heidy Khlaaf, and Nir Piterman. Fairness for infinite-state systems. In *TACAS*, pages 384–398, 2015.

[28] Petr Bauch, Vojtech Havel, and Jiri Barnat. LTL model checking of LLVM bitcode with symbolic data. In *MEMICS*, pages 47–59, 2014.

[29] Oded Padon, Jochen Hoenicke, Kenneth L. McMillan, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Temporal prophecy for proving temporal properties of infinite-state systems. *Formal Methods Syst. Des.*, 57(2):246–269, 2021.

[30] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the AMS*, 74 (2):358–366, 1953.

[31] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58 (345-363):5, 1936.

[32] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[33] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. Polynomial reachability witnesses via stellensätze. In *PLDI*, pages 772–787, 2021.

[34] Thomas Gurriet, Andrew Singletary, Jacob Reher, Laurent Ciarletta, Eric Feron, and Aaron D. Ames. Towards a framework for realizable safety critical control through active set invariance. In *ICCPS*, pages 98–106, 2018.

[35] Nathan Fulton. *Verifiably safe autonomy for cyber-physical systems.* PhD thesis, Carnegie Mellon University, 2018.

[36] Zhuo Cai, Soroush Farokhnia, Amir Kafshdar Goharshady, and S. Hitarth. Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts. In *OOPSLA*, 2023.

[37] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

[38] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*, pages 672–687, 2020.

[39] Louis De Branges. The Stone-Weierstrass theorem. *Proceedings of the AMS*, 10(5):822–824, 1959.

[40] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[41] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, pages 53–68, 2004.

[42] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through positivstellensatz's. In *CAV*, pages 3–22, 2016.

[43] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In *CAV*, pages 742–766, 2021.

[44] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. Modular primal-dual fixpoint logic solving for temporal verification. In *POPL*, pages 2111–2140, 2023.

[45] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear ranking for linear lasso programs. In *ATVA*, pages 365–380, 2013.

[46] Florian Funke, Simon Jantsch, and Christel Baier. Farkas certificates and minimal witnesses for probabilistic reachability constraints. In *TACAS*, pages 324–345, 2020.

[47] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *FM*, pages 187–201, 2012.
[48] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.
[49] Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik*, 1902(124):1–27, 1902.
[50] Eike Neumann, Joël Ouaknine, and James Worrell. On ranking function synthesis and termination for polynomial programs. In *CONCUR*, pages 15:1–15:15, 2020.
[51] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The probabilistic termination tool amber. In *FM*, pages 667–675, 2021.
[52] Liyong Shen, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. *J. Syst. Sci. Complex.*, 26(2):291–301, 2013.
[53] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. Proving non-termination by program reversal. In *PLDI*, pages 1033–1048, 2021.
[54] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In *ATVA*, pages 400–416, 2017.
[55] Andrew Clark. Verification and synthesis of control barrier functions. In *CDC*, pages 6105–6112, 2021.
[56] Yifan Zhang, Zhengfeng Yang, Wang Lin, Huibiao Zhu, Xin Chen, and Xuandong Li. Safety verification of nonlinear hybrid systems based on bilinear programming. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2768–2778, 2018.
[57] Qiuye Wang, Mingshuai Chen, Bai Xue, Naijun Zhan, and Joost-Pieter Katoen. Synthesizing invariant barrier certificates via difference-of-convex programming. In *CAV*, pages 443–466, 2021.
[58] Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and repulsing supermartingales for reachability in randomized programs. *TOPLAS*, 43(2):5:1–5:46, 2021.
[59] Amir Kafshdar Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit J. Motwani. Algebro-geometric algorithms for template-based synthesis of polynomial programs. In *OOPSLA*, pages 727–756, 2023.
[60] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer, 2018.
[61] Jan Strejcek. *Linear temporal logic: Expressiveness and model checking*. PhD thesis, Masaryk University, 2004.
[62] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144, 2005.
[63] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In *CAV*, pages 49–66, 2015.
[64] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *POPL*, pages 399–410, 2011.
[65] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196, 2016.
[66] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *TACAS*, pages 387–393, 2016.
[67] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. Extending nuXmv with timed transition systems and timed temporal properties. In *CAV*, pages 376–386, 2019.
[68] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. Efficient synthesis with probabilistic constraints. In *CAV*, pages 278–296, 2019.
[69] Aws Albarghouthi and Justin Hsu. Constraint-based synthesis of coupling proofs. In *CAV*, pages 327–346, 2018.
[70] Yiming Meng and Jun Liu. Lyapunov-barrier characterization of robust reach-avoid-stay specifications for hybrid systems, 2022. URL https://arxiv.org/abs/2211.00814.
[71] Sean Summers and John Lygeros. Verification of discrete time stochastic hybrid systems: A stochastic reach-avoid decision problem. *Autom.*, pages 1951–1961, 2010.
[72] Đorđe Žikelić, Mathias Lechner, Thomas A. Henzinger, and Krishnendu Chatterjee. Learning control policies for stochastic systems with reach-avoid guarantees. In *AAAI*, pages 11926–11935, 2023.
[73] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992. ISBN 978-3-540-97664-6. doi: 10.1007/978-1-4612-0931-7. URL https://doi.org/10.1007/978-1-4612-0931-7.
[74] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
[75] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *PLDI*, pages 219–230, 2013.
[76] J Richard Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. In *Studies in Logic and the Foundations of Mathematics*, volume 44, pages 1–11. 1966.
[77] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with aspic and c2fsm. *Electron. Notes Theor. Comput. Sci.*, pages 3–13, 2010.

[78] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. Non-linear reasoning for invariant synthesis. In *POPL*, pages 54:1–54:33, 2018.

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

[80] Dmitrii Grigorev and Nicolai Vorobjov. Solving systems of polynomial inequalities in subexponential time. *Journal of symbolic computation*, 5(1-2):37–64, 1988.

[81] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[82] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *CAV*, pages 294–298, 2008.

[83] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *TACAS*, pages 93–107, 2013.

[84] Florian Frohn, Jürgen Giesl, Georg Moser, Albert Rubio, Akihisa Yamada, et al. Termination competition 2022, 2021. URL https://termination-portal.org/wiki/Termination_Competition_2022.

[85] Dirk Beyer. Progress on software verification: SV-COMP 2022. In *TACAS*, pages 375–402, 2022.

[86] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, pages 377–410, 1990.

[87] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. *Formal Methods Syst. Des.*, 38(2):158–192, 2011.

[88] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

A PROOF OF LEMMA 4.1

Proof. First, suppose that there exists a scheduler σ in $\mathcal{T}$ such that $\pi_\sigma = (l_0, e_0), (l_1, e_1), \ldots$ satisfies φ. For each i let $\alpha_i \in 2^{\mathsf{AP}}$ be the set of all atomic predicates that are true in (l_i, e_i). Then, since N accepts the same language as φ, we know that $\alpha_0, \alpha_1, \ldots$ is an infinite word in the alphabet 2^{AP} which is accepting in N. Let $q_0, q_1, \ldots$ be an accepting run in N induced by this infinite word, and consider the following sequence of states in $\mathcal{T} \times N$:

$$\pi' = (l_0, q_0, e_0), (l_1, q_1, e_1), \ldots$$

We claim that this sequence is a run, i.e. that $(l_i, q_i, e_i) \rightsquigarrow (l_{i+1}, q_{i+1}, e_{i+1})$ for each i where $\rightsquigarrow$ is the set of transitions in $\mathcal{T} \times N$. To see this, note that since $\pi_\sigma = (l_0, e_0), (l_1, e_1), \ldots$ is a run in $\mathcal{T}$, for each i there exists a transition $\tau = (l_i, l_{i+1}, G_\tau, U_\tau)$ under which (l_{i+1}, e_{i+1}) is a successor of (l_i, e_i). On the other hand, we also know that $q_{i+1} \in \delta(q_i, \alpha_i)$ where δ is the transition function of N. Finally, since α_i is the set of all atomic predicates in AP that are true at (l_i, e_i), we must have $at(l_i) \in \alpha_i$. Hence, we have that $(l_i, q_i, e_i) \rightsquigarrow (l_{i+1}, q_{i+1}, e_{i+1})$ under transition $((l_i, q_i), (l_{i+1}, q_{i+1}), G_\tau \wedge \alpha^{-at}, U_\tau)$ and the claim follows. Finally, since $q_0, q_1, \ldots$ is accepting in N we have that $\pi' = (l_0, q_0, e_0), (l_1, q_1, e_1), \ldots$ contains infinitely many states in $\mathcal{B}_N^{\mathcal{T}}$, Hence, taking a scheduler in $\mathcal{T} \times N$ which induces the run π' solves the EB-PA problem for $\mathcal{T} \times N$ and $\mathcal{B}_N^{\mathcal{T}}$.

For the opposite direction, suppose that there exists a scheduler in $\mathcal{T} \times N$ which yields a run $\pi' = (l_0, q_0, e_0), (l_1, q_1, e_1), \ldots$ that visits $\mathcal{B}_N^{\mathcal{T}}$ infinitely many times. The definition of transitions in the product transition system $\mathcal{T} \times N$ implies that $(l_0, e_0), (l_1, e_1), \ldots$ is a run in $\mathcal{T}$, that $q_0, q_1, \ldots$ is an accepting run in N and that there exists an infinite accepting word $\alpha_0, \alpha_1, \ldots$ in 2^{AP} such that $at(l_i) \in \alpha_i$ for each i. Since $\alpha_0, \alpha_1, \ldots$ is accepting in N and $at(l_i) \in \alpha_i$ for each i and since N and φ accept the same language, it follows that $(l_0, e_0), (l_1, e_1), \ldots$ satisfies φ. Hence, taking a scheduler in $\mathcal{T}$ which induces the run $(l_0, e_0), (l_1, e_1), \ldots$ solves the ELTL-PA problem for $\mathcal{T}$ and φ. □

B PROOF OF LEMMA 4.2

Proof. First suppose that for every scheduler σ of $\mathcal{T}$, the induced run π_σ satisfies φ. Let $\pi = (l_0, e_0, q_0), (l_1, e_1, q_1), \ldots$ be a run in $\mathcal{T} \times D$, then $\eta = (l_0, e_0), (l_1, e_1), \ldots$ is a run in $\mathcal{T}$, hence satisfying φ. As D is deterministic, $q_0, q_1, \ldots$ is the unique run of D corresponding to η, so it has to be accepting. This shows that π is $\mathcal{B}_D^T$-Büchi. Therefore, every run of $\mathcal{T} \times D$ is $\mathcal{B}_D^T$-Büchi and the answer to the UB-PA problem is positive.

Now, suppose that every run of $\mathcal{T} \times D$ is $\mathcal{B}_D^T$-Büchi. Let $\pi = (l_0, e_0), (l_1, e_1), \ldots$ be a run in $\mathcal{T}$ and let $q_0, q_1, \ldots$ be the corresponding run in D. From the definition of product transition system, it is followed that $(l_0, e_0, q_0), (l_1, e_1, q_1), \ldots$ is a run of $\mathcal{T} \times D$ and therefore $\mathcal{B}_D^{\mathcal{T}}$-Büchi. This means that $q_0, q_1, \ldots$ is an accepting run in D, showing that π satisfies φ. Therefore, every run of $\mathcal{T}$ satisfies φ and the answer to the ULTL-PA problem is positive. □

C SUFFICIENCY OF MEMORY-LESS SCHEDULERS

The scheduler generated in the proof of Lemma 5.2 is memory-less. We now show that the existence of memory-less schedulers that generate $\mathcal{B}$-Büchi runs is equivalent to the existence of generic (history dependent) schedulers that generate $\mathcal{B}$-Büchi runs. We start by defining the cycle-decomposition of a trajectory. Trajectories are finite sub-sequences of runs and cycles are trajectories starting and ending in the same state.

Definition C.1 (Cycle-Decomposition of a Trajectory). For a trajectory τ starting at s_1 and ending at s_2, we define its Cycle-Decomposition as a tuple (C, β) where C is the set of all cycles appearing as a sub-trajectory of τ and β is the underlying non-cyclic trajectory in τ going from s_1 to s_2. See Figure 5 for an example.

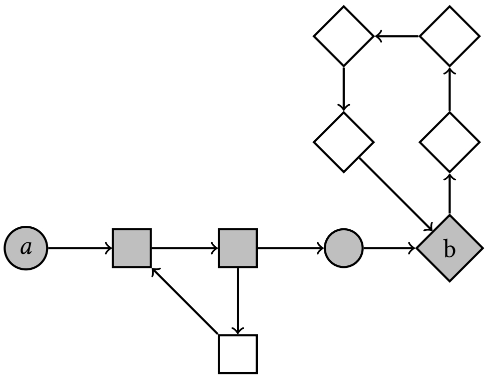

Fig. 5. An example finite path between two states a to b with its cycle-decomposition (C, β), where square and diamond states specify the cycles in C and the gray-scaled states specify β.

In order to construct the cycle-decomposition (C, β) of τ, we go through τ and push the states one by one into a stack. Whenever we visit a state s already in the stack, we pop all the elements between the two occurrences of s (including only one of the occurrences) and add them as a cycle to C. Finally, β will be the trajectory constructed by the states remaining in the stack at the end.

Lemma C.2. *There exists a* memory-less *$\mathcal{B}$-Büchi scheduler for $\mathcal{T}$* if and only if *there exists a $\mathcal{B}$-Büchi scheduler for $\mathcal{T}$.*

Proof. Necessity is trivial. For sufficiency, suppose σ is a $\mathcal{B}$-Büchi scheduler. We construct a memory-less scheduler σ' based on σ. Let $\pi = \pi_\sigma$ and let (C_i, β_i) be the cycle-decomposition of π^i for all $i \geq 0$. One of the following cases can happen:

(1) π has no cycles. In this case, σ was not using the history of the run to generate π. Therefore, we can define σ' to be the same as σ on π and arbitrary on other states. $\pi_{\sigma'}$ will then be the same as π_σ proving that σ is $\mathcal{B}$-Büchi.
(2) π has a finite number of cycles. So, there exists k such that π^{k+} is non-cyclic which means $\beta_k \cdot \pi^{k+}$ is a $\mathcal{B}$-Büchi run with no cycles where we can apply case (i).
(3) π has an infinite number of cycles. Here we would have two sub-cases:
    - π contains a cycle c starting at $\pi(b)$ which intersects with $\mathcal{B}$. Let k be the smallest index where $\pi(k) = \pi(b)$, then the run $\pi' = \beta_k \cdot c^\omega$ is easily seen to be $\mathcal{B}$-Büchi. As each state in π' has a unique successor in the run, there exists a memory-less scheduler σ' where $\pi_{\sigma'} = \pi'$.
    - None of the cycles of π intersect with $\mathcal{B}$. Removing all cycles from π produces a non-cyclic $\mathcal{B}$-Büchi run π'. The rest was handled in case (i).

□

D SOUNDNESS AND COMPLETENESS OF UNIVERSAL WITNESSES

Lemma D.1. *Every memory-less scheduler of $\mathcal{T}$ is $\mathcal{B}$-Büchi, if and only if every scheduler of $\mathcal{T}$ is $\mathcal{B}$-Büchi*

Proof. Necessity is trivial. For sufficiency, suppose every memory-less scheduler of $\mathcal{T}$ is $\mathcal{B}$-Büchi and let σ be a non-$\mathcal{B}$-Büchi scheduler for $\mathcal{T}$. Let $\pi = \pi_\sigma$ and let (C_i, β_i) be the cycle-decomposition of π^i for all $i \geq 0$. One of the following cases happens:

(1) π has no cycles. In this case, σ does not use the history of the run to generate π. Therefore, we can define σ' to be the same as σ on π and arbitrary on other states. It is then easy to see that $\pi_\sigma = \pi_{\sigma'}$ which means σ' is a non-$\mathcal{B}$-Büchi memory-less scheduler which is a contradiction.
(2) π has a finite number of cycles. In this case, there exists k such that π^{k+} is non-cyclic and does not visit $\mathcal{B}$. Therefore, $\beta_k \cdot \pi^{k+}$ is a non-cyclic non-$\mathcal{B}$-Büchi run of $\mathcal{T}$. Similar to the previous case, this yields a contradiction.
(3) π has an infinite number of cycles. As π reaches $\mathcal{B}$ only a finite number of times, there must exists a cycle in π that does not reach π. Suppose $c = [\pi(i), \pi(i+1), \dots, \pi(j)]$ is one such cycle. Then $\beta_{i-1} \cdot c^\omega$ is a non-cyclic non-$\mathcal{B}$-Büchi run of $\mathcal{T}$. Similar to the previous cases, this yields a contradiction.

□

Proof of Theorem 5.7. We are now ready to prove the soundness and completeness theorem for UBRFs.

Soundness. Let $\mathcal{T}$ be a transition system, f a $\mathcal{B}$-UBRF for $\mathcal{T}$ and π a run in $\mathcal{T}$. By the definition of UBRFs, $f(\pi(i)) \geq 0$ for all $i \geq 0$ and $f(\pi(i+1)) \leq f(\pi(i)) - 1$ if $\pi(i) \notin \mathcal{B}$. Now suppose for the sake of contradiction that π is not $\mathcal{B}$-Büchi. Then there exists k such that π^{k+} does not reach $\mathcal{B}$ at all. So, by taking a transition from $\pi(k+i)$ to $\pi(k+i+1)$ the value of f decreases by at least 1 while staying non-negative which yields a contradiction, showing that π is $\mathcal{B}$-Büchi. This proves the soundness of the witness. □

Completeness. For completeness, we must show that whenever every run of $\mathcal{T}$ is $\mathcal{B}$-Büchi, there exists a $\mathcal{B}$-UBRF f for $\mathcal{T}$. For a states $s \in \mathcal{S}$ let $d(s) : \mathcal{S} \to \mathbb{N}_0$ be the defined as follows:

$$d(s) = \begin{cases} 0 & s \in \mathcal{B} \wedge s \text{ is reachable} \\ \sup\limits_{s \mapsto s'} d(s') + 1 & s \notin \mathcal{B} \wedge s \text{ is reachable} \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

It is trivial that if d is definable, then it is a $\mathcal{B}$-UBRF for $\mathcal{T}$. So, we show that d is a well-defined. $d(s)$ is trivially well-defined in the first and last case of (2). Suppose for a moment that the co-domain of d is $\mathbb{N}_0 \cup \{\infty\}$. We show that the ∞ case will never happen. For the sake of contradiction assume that there exists a reachable state $s_0 \notin B$ such that $\sup\limits_{s_0 \mapsto s'} d(s') + 1 = \infty$. So, there must exist s_1 such that $s_0 \mapsto s_1$ and $d(s_1) = \infty$. Continuing the same procedure inductively, there must exist s_i such that $s_{i-1} \mapsto s_i$ and $d(s_i) = \infty$. Hence, if α would be the trajectory ending at s_0, the run $\alpha, s_1, s_2, \dots$ is not $\mathcal{B}$-Büchi, which is a contradiction. So, d is well-defined. □

E COMPARISON OF SMT SOLVERS

As mentioned in Section 7, in our implementation of our approach, we pass the resulting QP instance to all three SMT solvers in parallel. Table 5 shows the individual performance of different SMT solvers on linear benchmarks. We observe that MathSAT5 outperforms the other two solvers overall. However, none of the solvers outperforms others in every case. This confirms that, much like many other program analyses, LTL verification tools should use several SMT solvers in parallel to achieve the best performance.

Formula	Barcelogic		MathSAT5		Z3	
	Yes	No	Yes	No	Yes	No
RA	115	91	138	106	81	42
OV	138	38	197	46	111	28
PR	75	139	84	189	39	109

Table 5. Performance of our Approach using Different SMT Solvers