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

Krishnendu Chatterjee[1], Amir Kafshdar Goharshady[2], Ehsan Kafshdar Goharshady[1], Mehrdad Karrabi[1], and Đorđe Žikelić[3]*

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
{krishnendu.chatterjee, ehsan.goharshady, mehrdad.karrabi}@ist.ac.at
[2] The Hong Kong University of Science and Technology (HKUST), Hong Kong
goharshady@cse.ust.hk
[3] Singapore Management University, Singapore
dzikelic@smu.edu.sg

Abstract. We study the classical problem of verifying programs with respect to formal specifications given in the linear temporal logic (LTL). We first present novel sound and complete witnesses for LTL verification over imperative programs. Our witnesses are applicable to both verification (proving) and refutation (finding bugs) settings. We then consider LTL formulas in which atomic propositions can be polynomial constraints and turn our focus to polynomial arithmetic programs, i.e. programs in which every assignment and guard consists only of polynomial expressions. For this setting, we provide an efficient algorithm to automatically synthesize such LTL witnesses. Our synthesis procedure is both sound and semi-complete. 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

Linear-time Temporal Logic. The Linear-time Temporal Logic (LTL) [48] is one of the most classical and well-studied frameworks for formal specification, model checking and program verification. 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, $\texttt{X}\ p$ requires that p holds in the next timeslot, whereas $\texttt{F}\ q$ means q should hold at some time in the future. This allows LTL to express common verification tasks such as termination, liveness, fairness and safety.

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

*Part of the work done while the author was at the Institute of Science and Technology Austria (ISTA).

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. Witnesses are especially useful in dealing with undecidable problems in verification, which includes all non-trivial semantic properties [52]. 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 [59], 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 [49]. Similarly, while reachability (safety violation) is undecidable, it has sound and complete witnesses that can be automatically synthesized in linear and polynomial forms [1]. Our work subsumes both [49] and [1] and provides sound and complete witnesses for general LTL formulas.

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.

Our Contributions. In this work, our contributions are threefold:
– On the theoretical side, 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 [31], inductive reachability witnesses [1] and inductive invariants [20], which are sound and complete witnesses for termination, reachability and safety, respectively. Our theoretical result is not limited to polynomial programs.
– On the algorithmic side, we consider polynomial programs and present a sound and semi-complete template-based algorithm to synthesize polynomial LTL witnesses. This algorithm is a generalization of the template-based approaches in [49,1,20] which considered termination, reachability and safety. To the best of our knowledge, this is the most general model checking problem over polynomial programs to be handled by template-based approaches to date.
– Finally, on the experimental side, we provide an 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 completeness result pays off in practice and enables us to solve new instances.

Motivation for Polynomial Programs. There are several reasons why we consider polynomial programs:
– Many real-world families of programs, such as, programs for cyber-physical systems and smart contracts, can be modeled in this framework [38,33,10].
– They are one of the most general families for which finding polynomial witnesses for reachability and safety are known to be decidable [53,1,12]. 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 [25]. Thus, analysis of polynomial programs can potentially be applied to many non-polynomial programs via abstract interpretation or numerical approximation of the program's behavior.
– Previous works have studied (a) linear/affine programs with termination, safety, and reachability specifications [49,20,54], and (b) polynomial programs with termination, safety and reachability properties [53,12,1,11]. Since LTL subsumes all these specifications, polynomial program analysis with LTL provides a unifying and general framework for all these previous works.

Related Works on Linear Programs. There are many approaches focusing on linear witness synthesis for important special cases of LTL formulas. For example, [49,39] consider the problem of synthesizing linear ranking functions (termination witnesses) over linear arithmetic programs. The works [20,54] synthesize linear inductive invariants (safety witnesses), while [34] considers probabilistic reachability witnesses. The work [37] 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 [27] 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, [46,44,11,55,13] consider termination analysis, [12] extends the invariant generation (safety witness synthesis) algorithm of [20] to the polynomial case and [30] further adds support for probabilistic programs. The works [17,64,63] consider alternative types of witnesses for safety (barriers) and obtain similarly successful synthesis algorithms. Finally, [58,1] synthesize reachability witnesses. Since we can handle any arbitrary LTL formula, our approach can be seen as an extension and unification of all these works. Indeed, our synthesis algorithm directly builds upon and extends [1].

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. We refer to [19,56] for an excellent treatment of the finite-state cases. Some works that provide LTL model checking over infinite-state systems/programs are as follows:
– A prominent technique in this area is predicate abstraction [35,24,50], 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.

– [14] 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 [21] which is focused on fairness and [4] which is applicable to LLVM. Another work in this category is [26], 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 [22] 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 [28] presents a framework for proving liveness properties in multi-threaded programs by using well-founded proof spaces.
– The recent work [47] 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 [8] is able to verify a large family of liveness and safety properties, nuXmv [15] is a symbolic model checker with support for LTL, F3 [14] proves fairness in infinite-state transition systems, and Ultimate LTLAutomizer [26] 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 [60]. 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 [61] 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.

As shown by our experimental results in Section 5, 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 [26] is able to handle non-polynomial programs and witnesses, too.

2 Transition Systems, LTL and Büchi Automata

For a vector $e \in \mathbb{R}^n$, 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

```
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
$l_t$:
```

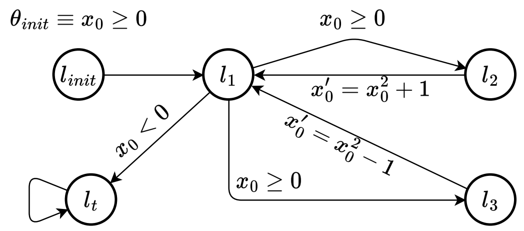

Fig. 1: An example program (left) and its transition system (right). Note that there is non-determinism at l_1.

non-deterministic branching in our syntax by **if** $*$ **then**. See Figure 1 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* with $l_{init} \in L$ 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.

Translating programs into transition systems is a standard process. In what follows, 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. An example is shown in Figure 1.

States and Runs. A *state* in $\mathcal{T}$ is a 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. We 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 so that all runs are infinite and LTL semantics are well defined. This is without loss of generality, since we can introduce transitions to the terminal location. A *run* in $\mathcal{T}$ is an infinite sequence of successor states starting in $\mathscr{S}_{init}$.

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

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

$\neg, \vee$ and $\wedge$ are the propositional negation, disjunction and conjunction while `X`, `G`, `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 $\texttt{AP}$. In this work, we let the set $\texttt{AP}$ 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 $\mathcal{T}$. 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.

The semantics of LTL is standard, refer to Appendix A for details.

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 two problems:

1. *LTL Verification of Programs (LTL-VP).* Given a transition system $\mathcal{T}$ and an LTL formula φ in $\mathcal{T}$, prove that *all possible runs* of $\mathcal{T}$ satisfy φ.
2. *LTL Refutation of Programs (LTL-RP).* Given a transition system $\mathcal{T}$ and an LTL formula φ in $\mathcal{T}$, prove that there *exists a run* that violates φ, or equivalently, satisfies $\neg\varphi$.

Remark. *LTL Verification* asks about correctness of the program while *LTL Refutation* addresses the problem of finding bugs. Both problems have been widely studied in the literature [3,26,60]. Moreover, a witness for the refutation problem can be used in counterexample-guided techniques such as CEGAR [18].

Example. Consider the transition system in Figure 1 and the LTL formula $\varphi = \neg[\texttt{G}(at(l_3) \Rightarrow \texttt{F}at(l_2))]$. The run that starts at $(l_{init}, 1)$ and chooses l_2 if $x_0 = 0$ and l_3 whenever $x_0 = 1$, does not satisfy φ. Therefore, in this case, the answer to the LTL-RP problem is positive. Additionally, deciding termination of a program with terminal location l_t is equivalent to the LTL-VP problem of $[\texttt{F}\ \texttt{G}\ at(l_t)]$ on the same program.

Program Analysis with Büchi Specifications. A Büchi specification is a subset $\mathcal{B} \subseteq \mathscr{S}$ of states. A run π is $\mathcal{B}$*−Büchi* if it visits $\mathcal{B}$ infinitely many times, i.e. if $\{i \mid \pi(i) \in \mathcal{B}\}$ is infinite. Similar to LTL, Büchi specifications give rise to two main decision problems as follows:

1. *Universal Büchi Program Analysis (UB-PA).* Given a transition system $\mathcal{T}$ and a Büchi specification $\mathcal{B}$ on $\mathcal{T}$, prove that *all possible runs* of $\mathcal{T}$ are $\mathcal{B}$*−Büchi.*
2. *Existential Büchi Program Analysis (EB-PA).* Given a transition system $\mathcal{T}$ and a Büchi specification $\mathcal{B}$ on $\mathcal{T}$, prove the *existence* of a run that is $\mathcal{B}$*−Büchi.*

Büchi Automata [2,9]. A *non-deterministic Büchi automaton (NBW)* is a 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 $a_0, a_1, \ldots$ 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 $q_0, q_1, \ldots$ such that $q_{i+1} \in \delta(q_i, a_i)$ for each i and F is visited infinitely many times. It is a classical result that for every LTL formula φ defined over atomic predicates $\texttt{AP}$ there exists a non-deterministic Büchi automaton N with alphabet 2^{AP} which accepts exactly those traces that satisfy φ [19].

Let $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ be a transition system and $N = (Q, 2^{AP}, \delta, q_0, F)$ be an NBW. In order to analyse $\mathcal{T}$ with respect to N, we utilize the Cartesian

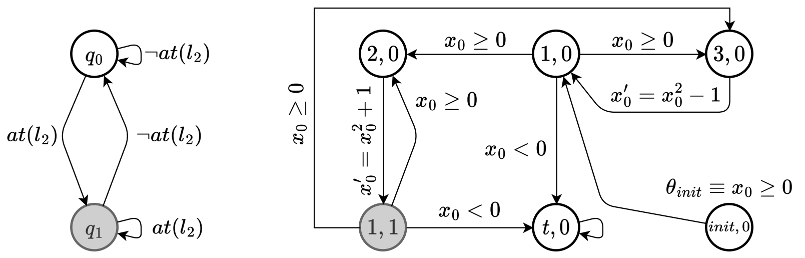

Fig. 2: An NBW accepting `G F` $at(l_2)$ with gray accepting nodes (left) and the product of the transition system in Figure 1 and this NBW (right). A node labeled i, j represents location (l_i, q_j). Unreachable locations have been removed.

product $\mathcal{T} \times N$ and the Büchi specification $\mathcal{B}_N^{\mathcal{T}} = L \times F \times \mathbb{R}^n$. The state space of $\mathcal{T} \times N$ is exactly the Cartesian product of the state spaces of $\mathcal{T}$ and N. Moreover, for $l, l' \in L$ and $q, q' \in Q$, there is a transition from (l, q) to (l', q') if there is a transition in $\mathcal{T}$ from l to l' and a transition in N from q to q'. The formal definition of the product is available in Appendix B. See Figure 2 for an example.

Lemma 1 (From LTL to Büchi Specifications, Proof in Appendix C). *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 LTL-RP problem of $\mathcal{T}$ and $\neg\varphi$ is equivalent to the EB-PA problem of $\mathcal{T} \times N$ and $\mathcal{B}_N^{\mathcal{T}}$ [26].*
– *If N is deterministic, then the LTL-VP problem of $\mathcal{T}$ and φ is equivalent to the UB-PA problem of $\mathcal{T} \times N$ and $\mathcal{B}_N^{\mathcal{T}}$.*

Remark. Based on the lemma above, instead of designing witnesses for the LTL-RP problem, we only need to find sound and complete witnesses for EB-PA. Moreover, it is easy to see that LTL-VP is reducible to LTL-RP 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 verification and refutation variants of LTL program analysis. Note that the second statement in Lemma 1 is more restrictive than the first one since it only applies to deterministic Büchi automata. Thus, if the LTL formula φ does not admit a deterministic Büchi automaton, the above sequence of reductions from LTL-VP to LTL-RP should be made and then the EB-PA witness should be used. However, if φ admits a DBW, then the reduction to UB-PA is preferable in practice. We will provide witness concepts for both EB-PA and UB-PA problems in the next section.

3 Sound and Complete B-PA Witnesses

Let $\mathcal{T} = (\mathcal{V}, L, l_{init}, \theta_{init}, \mapsto)$ be a transition system and $\mathcal{B} \subseteq \mathscr{S}$ 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.

3.1 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 not 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 1 (EBRF). *Given two states $s_1, s_2 \in \mathscr{S}$, a function $f \colon \mathscr{S} \to \mathbb{R}$ is said to Büchi-rank (s_1, s_2) where $s_1 \mapsto s_2$, if it satisfies one of the following:*
- $s_1 \in \mathcal{B} \wedge \big[f(s_1) \geq 0 \Rightarrow f(s_2) \geq 0\big]$; *or*
- $s_1 \notin \mathcal{B} \wedge \big[f(s_1) \geq 0 \Rightarrow 0 \leq f(s_2) \leq f(s_1) - 1\big]$.

f is called a $\mathcal{B}$-Existential Büchi Ranking Function ($\mathcal{B}$-EBRF) if it satisfies the following conditions:
- $\exists s_{init} \in \mathscr{S}_{init}$ *where* $f(s_{init}) \geq 0$.
- *For every $s_1 \in \mathscr{S}$, there exists $s_2 \in \mathscr{S}$ such that $s_1 \mapsto s_2$ and (s_1, s_2) is Büchi-ranked by f.*

Example. The following is a $\{(l_1, q_1, *)\}$-EBRF for the transition system in Fig. 2: $f(l, x_0) = x_0 + 3$ if $l = (l_{init}, q_0)$, $f(l, x_0) = x_0 + 2$ if $l = (l_1, q_0)$, $f(l, x_0) = x_0 + 1$ if $l = (l_2, q_0)$, $f(l, x_0) = 0$ if $l = (l_1, q_1)$ and $f(l, x_0) = 0$ otherwise.

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 theorem, proved in Appendix E, establishes the soundness and completeness of EBRFs for the EB-PA problem, which is the main result of this section. Hence, since we showed in Lemma 1 that one can reduce the LTL-RP problem to EB-PA, as a corollary it also follows that EBRFs provide sound and complete certificates for LTL-RP.

Theorem 1 (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 1. *The answer to the LTL-RP 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 $\neg\varphi$.*

3.2 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 2 (UBRF). *A function $f\colon \mathscr{S} \to \mathbb{R}^n$ is called a $\mathcal{B}$-Universal Büchi Ranking Function ($\mathcal{B}$-UBRF) if it satisfies the following conditions:*
– *$f(s) \geq 0$ for* ***every*** *$s \in \mathscr{S}_{init}$*
– *For* ***every*** *$s_1, s_2 \in \mathscr{S}$ 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 presented in Appendix G. The subsequent corollary then follows from Lemma 1 which shows that the LTL-VP problem can be reduced to the UB-PA problem if φ admits a deterministic Büchi automaton.

Theorem 2 (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 2. *If φ is an LTL formula that admits a DBW D, the answer to the LTL-VP problem of $\mathcal{T}$ and φ is positive if and only if there exists a $\mathcal{B}_D^{\mathcal{T}}$-UBRF for $\mathcal{T} \times D$.*

Remark. Note that if the transition system $\mathcal{T}$ is deterministic, (i.e. it contains no non-determinism in initial states, assignments or branches) the LTL-VP of $\mathcal{T}$ and φ will be equivalent to the LTL-RP of $\mathcal{T}$ and $\neg\varphi$. Thus, in this case, the Büchi automaton determinism assumption can be relaxed as follows: if N is an NBW that accepts the same language as φ, the answer to the LTL-VP of $\mathcal{T}$ and φ is positive if and only if there exists a $\mathcal{B}_N^{\mathcal{T}}$-EBRF for $\mathcal{T} \times N$.

4 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 [1,12]. 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

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

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 [1] 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 [1,12] 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 polynomial programs, invariants can be automatically generated using the tools in [29,12,40]. Alternatively, one can encode an inductive invariant via 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 [12] 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 [12]. 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`. We present a detailed example that illustrates the steps of the algorithm in appendix H. The algorithm has five steps:

1. *Fixing Symbolic Templates.* Let $M_{\mathcal{V}}^{D} = \{m_1, m_2, \dots, m_k\}$ 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 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 $\mathcal{T}$.
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: $\forall x \in \mathbb{R}^n \quad x \models (\phi_l \Rightarrow \psi_l)$ 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:

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

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.

3. *Reduce Constraints to Quadratic Inequalities.* To solve the constraints generated in the previous step, we directly integrate the technique of [1] 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 [1] 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 $1 \leq i \leq t$ and $1 \leq j \leq r$ the algorithm uses Putinar's Positivstellensatz in the exact same way as in [1] to generate a set of quadratic inequalities equivalent to $\phi_i \Rightarrow \psi_j$. The algorithm keeps track of a quadratic program Γ and adds these new inequalities to it conjunctively.
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 $[\theta_{init}(t) \wedge f_{l_{init}}(t) \geq 0]$ to Γ to impose that there exists an initial state in $\mathcal{T}$ at which the value of the EBRF f is non-negative.
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.

Theorem 3 (Existential Soundness and Semi-Completeness). *The algorithm above is a sound and semi-complete reduction to quadratic programming for synthesizing 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 [1] for details of Step 3 and its soundness and semi-completeness. We are using the technique of [1] as a black

[§]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).

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 [1].

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 [36]. 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 3.

Theorem 4 (Universal Soundness and Semi-Completeness). *The algorithm above is a sound and semi-complete reduction to quadratic programming for synthesizing 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.*

5 Experimental Results

General Setup of Experiments. We implemented a prototype of our UBRF and EBRF synthesis algorithms in Java and used Z3 [45], Barcelogic [6] and MathSAT5 [16] 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 [29] for invariant generation for benchmarks that are linear programs. 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 [26], nuXmv [15], and MuVal [60] 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 [45]:

– 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 [60] is a fixed-point logic validity checker based on pfwCSP solving [61]. 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 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 [32][¶]. Among these, 287 programs only contained linear arithmetic which is supported by all comparator tools, whereas 10 programs (Appendix I) contained polynomial expressions not supported by Ultimate.
– 21 non-linear benchmarks from the "ReachSafety-Loops `nl-digbench`" category of SV-COMP'22 [5][‖]. 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 1. In all four considered specifications, x represents the alphabetically first variable in the input program. The motivation behind our specifications is as follows:

– *Reach-avoid (RA) specifications.* The first specification is an example of reach-avoid specifications. 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. Such specifications are standard in the analysis of dynamical and hybrid systems [43,57,62]. Another common example is requiring program termination while satisfying all assertions in the program.
– *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

[¶]There were originally 335 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 1: 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.
Linear	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. T	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	-
Non-linear	RA	24	3	27	8	-	-	-	-	1	0	1	0	18	1	19	2	0	0	0	0
	OV	26	0	26	2	-	-	-	-	7	0	7	0	25	0	25	1	0	0	0	0
	RC	20	6	26	0	-	-	-	-	17	9	26	2	17	7	24	2	0	0	0	0
	PR	11	16	27	1	-	-	-	-	9	16	25	0	5	16	21	1	0	0	0	0
	Avg. T	10.7	99.1	32.3	-	-	-	-	-	34.6	0.3	20.0	-	109.6	14.7	84.7	-	-	-	-	-

Table 2: Summary of our experimental results. For each class of benchmarks (linear/non-linear) and each formula, 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 (Avg. T).

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 recurrence specifications which specify that a program run visits a set of states infinitely many times [42]. Our example 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. In our experimental evaluation, progress specification 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. The top rows of Table 2 summarize 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 LTL refutation and one LTL verification 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 2 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.

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 further improve the performance.

Runtimes. Our tool and Ultimate are the fastest tools for proving LTL verification instances with an equal average runtime of 5.4 seconds. For LTL refutation, our tool is slower than other tools.

Results on Non-Linear Programs. The bottom rows of Table 2 show the performance of our tool and the baselines 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 [1]) 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 LTL verification and refutation witnesses for a wide variety of programs. Our technique outperforms the previous methods when given non-linear polynomial programs (Bottom rows of Table 2). Moreover, 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 (Top Rows of Table 2).

6 Conclusion

We presented a novel family of sound and complete witnesses for template-based LTL verification. Our approach is applicable to both verification and refutation of LTL properties in programs. 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. An interesting direction of future work would be to consider non-numerical programs that allow heap-manipulating operations. A common approach to handling heap-manipulating operations is to construct numerical abstractions of programs [7,41] and perform the analysis on numerical abstractions. Thus, coupling such approaches, e.g. [23], with our method is a compelling future direction.

References

1. Asadi, A., Chatterjee, K., Fu, H., Goharshady, A.K., Mahdavi, M.: Polynomial reachability witnesses via stellensätze. In: PLDI. pp. 772–787 (2021)
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. Baresi, L., Kallehbasti, M.M.P., Rossi, M.: Efficient scalable verification of LTL specifications. In: ICSE (1). pp. 711–721. IEEE Computer Society (2015)
4. Bauch, P., Havel, V., Barnat, J.: LTL model checking of LLVM bitcode with symbolic data. In: MEMICS. pp. 47–59 (2014)
5. Beyer, D.: Progress on software verification: SV-COMP 2022. In: TACAS. pp. 375–402 (2022)
6. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: CAV. pp. 294–298 (2008)
7. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. Formal Methods Syst. Des. **38**(2), 158–192 (2011)
8. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: TACAS. pp. 387–393 (2016)
9. Büchi, J.R.: Symposium on decision problems: On a decision method in restricted second order arithmetic. In: Studies in Logic and the Foundations of Mathematics, vol. 44, pp. 1–11 (1966)
10. Cai, Z., Farokhnia, S., Goharshady, A.K., Hitarth, S.: Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts. In: OOPSLA (2023)
11. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through positivstellensatz's. In: CAV. pp. 3–22 (2016)
12. Chatterjee, K., Fu, H., Goharshady, A.K., Goharshady, E.K.: Polynomial invariant generation for non-deterministic recursive programs. In: PLDI. pp. 672–687 (2020)
13. Chatterjee, K., Goharshady, E.K., Novotný, P., Žikelić, Đ.: Proving non-termination by program reversal. In: PLDI. pp. 1033–1048 (2021)
14. Cimatti, A., Griggio, A., Magnago, E.: LTL falsification in infinite-state systems. Inf. Comput. **289**, 104977 (2022)
15. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with timed transition systems and timed temporal properties. In: CAV. pp. 376–386 (2019)
16. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: TACAS. pp. 93–107 (2013)
17. Clark, A.: Verification and synthesis of control barrier functions. In: CDC. pp. 6105–6112 (2021)
18. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV (2000)
19. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018)
20. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: CAV. pp. 420–432 (2003)
21. Cook, B., Khlaaf, H., Piterman, N.: Fairness for infinite-state systems. In: TACAS. pp. 384–398 (2015)
22. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: POPL. pp. 399–410 (2011)
23. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: PLDI. pp. 219–230 (2013)

24. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In: CAV. pp. 271–291 (2016)
25. De Branges, L.: The Stone-Weierstrass theorem. Proceedings of the AMS **10**(5), 822–824 (1959)
26. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: A new approach to LTL software model checking. In: CAV. pp. 49–66 (2015)
27. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik **1902**(124), 1–27 (1902)
28. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: LICS. pp. 185–196 (2016)
29. Feautrier, P., Gonnord, L.: Accelerated invariant generation for C programs with aspic and c2fsm. Electron. Notes Theor. Comput. Sci. pp. 3–13 (2010)
30. Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: ATVA. pp. 400–416 (2017)
31. Floyd, R.W.: Assigning meanings to programs. Program Verification: Fundamental Issues in Computer Science pp. 65–81 (1993)
32. Frohn, F., Giesl, J., Moser, G., Rubio, A., Yamada, A., et al.: Termination competition 2022 (2021), `https://termination-portal.org/wiki/Termination_Competition_2022`
33. Fulton, N.: Verifiably safe autonomy for cyber-physical systems. Ph.D. thesis, Carnegie Mellon University (2018)
34. Funke, F., Jantsch, S., Baier, C.: Farkas certificates and minimal witnesses for probabilistic reachability constraints. In: TACAS. pp. 324–345 (2020)
35. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. pp. 72–83 (1997)
36. Grigorev, D., Vorobjov, N.: Solving systems of polynomial inequalities in subexponential time. Journal of symbolic computation **5**(1-2), 37–64 (1988)
37. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. pp. 281–292 (2008)
38. Gurriet, T., Singletary, A., Reher, J., Ciarletta, L., Feron, E., Ames, A.D.: Towards a framework for realizable safety critical control through active set invariance. In: ICCPS. pp. 98–106 (2018)
39. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: ATVA. pp. 365–380 (2013)
40. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-linear reasoning for invariant synthesis. In: POPL. pp. 54:1–54:33 (2018)
41. Magill, S., Tsai, M., Lee, P., Tsay, Y.: Automatic numeric abstractions for heap-manipulating programs. In: POPL. pp. 211–222 (2010)
42. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC. pp. 377–410 (1990)
43. Meng, Y., Liu, J.: Lyapunov-barrier characterization of robust reach-avoid-stay specifications for hybrid systems (2022). https://doi.org/10.48550/ARXIV.2211.00814, `https://arxiv.org/abs/2211.00814`
44. Moosbrugger, M., Bartocci, E., Katoen, J., Kovács, L.: The probabilistic termination tool amber. In: FM. pp. 667–675 (2021)
45. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. pp. 337–340 (2008)
46. Neumann, E., Ouaknine, J., Worrell, J.: On ranking function synthesis and termination for polynomial programs. In: CONCUR. pp. 15:1–15:15 (2020)

47. Padon, O., Hoenicke, J., McMillan, K.L., Podelski, A., Sagiv, M., Shoham, S.: Temporal prophecy for proving temporal properties of infinite-state systems. Formal Methods Syst. Des. **57**(2), 246–269 (2021)
48. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57 (1977)
49. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. pp. 239–251 (2004)
50. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL. pp. 132–144 (2005)
51. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana University Mathematics Journal **42**(3), 969–984 (1993)
52. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the AMS **74**(2), 358–366 (1953)
53. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: POPL. pp. 318–329 (2004)
54. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS. pp. 53–68 (2004)
55. Shen, L., Wu, M., Yang, Z., Zeng, Z.: Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. J. Syst. Sci. Complex. **26**(2), 291–301 (2013)
56. Strejcek, J.: Linear temporal logic: Expressiveness and model checking. Ph.D. thesis, Masaryk University (2004)
57. Summers, S., Lygeros, J.: Verification of discrete time stochastic hybrid systems: A stochastic reach-avoid decision problem. Autom. pp. 1951–1961 (2010)
58. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in randomized programs. TOPLAS **43**(2), 5:1–5:46 (2021)
59. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. J. of Math **58**(345-363), 5 (1936)
60. Unno, H., Terauchi, T., Gu, Y., Koskinen, E.: Modular primal-dual fixpoint logic solving for temporal verification. In: POPL. pp. 2111–2140 (2023)
61. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: CAV. pp. 742–766 (2021)
62. Žikelić, Đ., Lechner, M., Henzinger, T.A., Chatterjee, K.: Learning control policies for stochastic systems with reach-avoid guarantees. In: AAAI. pp. 11926–11935 (2023)
63. Wang, Q., Chen, M., Xue, B., Zhan, N., Katoen, J.: Synthesizing invariant barrier certificates via difference-of-convex programming. In: CAV. pp. 443–466 (2021)
64. Zhang, Y., Yang, Z., Lin, W., Zhu, H., Chen, X., Li, X.: Safety verification of nonlinear hybrid systems based on bilinear programming. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2768–2778 (2018)

A LTL Semantics

Semantics. Each atomic proposition p in $\texttt{AP}$ 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 $\pi = (l_0, e_0), (l_1, e_1), \ldots$ of states, we write $\pi \models \varphi$ to denote that it satisfies φ and have:

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

B From LTL Formulas to Büchi Automata

It is a classical result in model checking that for every LTL formula φ defined over atomic predicates $\texttt{AP}$ there exists a non-deterministic Büchi automaton N with alphabet 2^{AP} which accepts exactly those traces that satisfy φ [19]. Here, $2^{\texttt{AP}}$ is the set of subsets of $\texttt{AP}$.

We now define the product of a transition system $\mathcal{T}$ and an NBW N over the alphabet $2^{\texttt{AP}}$. Note that at every step of a run of a transition system, we are always at a unique location. Hence, if a letter $\alpha \in 2^{\texttt{AP}}$ 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 $N = (Q, 2^{\texttt{AP}}, \delta, q_0, F)$ 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 $\alpha \in 2^{\texttt{AP}}$ 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.

[**]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 $\bot$ and let $\delta(q, \alpha) = \bot$. The sink state will only have self-loops, i.e. $\delta(\bot, \beta) = \bot$ for every $\beta \in 2^{\texttt{AP}}$.

C Proof of Lemma 1

C.1 From LTL-RP to EB-PA

Proof. First, suppose that there exists a run $\pi = (l_0, e_0), (l_1, e_1), \dots$ in $\mathcal{T}$ that satisfies φ. For each i let $\alpha_i \in 2^{\mathtt{AP}}$ 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 $\alpha_0, \alpha_1, \dots$ is an infinite word in the alphabet $2^{\mathtt{AP}}$ which is accepting in N. Let $q_0, q_1, \dots$ be an accepting run in N induced by this infinite word, and consider the following sequence of states in $\mathcal{T} \times N$:

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

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 $\pi = (l_0, e_0), (l_1, e_1), \dots$ 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 $\mathtt{AP}$ 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 $q_0, q_1, \dots$ is accepting in N we have that $\pi' = (l_0, q_0, e_0), (l_1, q_1, e_1), \dots$ contains infinitely many states in $\mathcal{B}_N^{\mathcal{T}}$.

For the opposite direction, suppose that there exists a run $\pi' = (l_0, q_0, e_0), (l_1, q_1, e_1), \dots$ in $\mathcal{T} \times N$ 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 $(l_0, e_0), (l_1, e_1), ...$ is a run in $\mathcal{T}$, that $q_0, q_1, \dots$ is an accepting run in N and that there exists an infinite accepting word $\alpha_0, \alpha_1, \dots$ in 2^{AP} such that $at(l_i) \in \alpha_i$ for each i. Since $\alpha_0, \alpha_1, \dots$ 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 $(l_0, e_0), (l_1, e_1), ...$ satisfies φ. □

C.2 From LTL-VP to UB-PA

Proof. First suppose that every run π of $\mathcal{T}$ satisfies φ. Let $\pi' = (l_0, e_0, q_0), (l_1, e_1, q_1), \dots$ be a run in $\mathcal{T} \times D$, then $\eta = (l_0, e_0), (l_1, e_1), \dots$ is a run in $\mathcal{T}$, hence satisfying φ. As D is deterministic, $q_0, q_1, \dots$ is the unique run of D corresponding to η, so it has to be accepting. This shows that π' is $\mathcal{B}_D^{\mathcal{T}}$-Büchi. Therefore, every run of $\mathcal{T} \times D$ is $\mathcal{B}_D^{\mathcal{T}}$-Büchi and the answer to the UB-PA problem is positive.

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

D Scheduler Definition

We define scheduler in this section and will use it in the proofs of Appendices E, F, and G.

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'. 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 $i \in \mathbb{N}$. For a subset $\mathcal{B} \subseteq \mathscr{S}$ of state, a scheduler is $\mathcal{B}-$Büchi if its run is.

E Proof of Theorem 1

Please see Appendix D for the definition of schedulers.

Proof (Soundness). Let $e_{init} \in \theta_{init}$ be such that $f(l_{init}, e_{init}) \geq 0$ and $(l, e) \in \mathscr{S}$ be any state of $\mathcal{T}$. By definition of EBRFs, there exists $(l', e') \in \mathscr{S}$ 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. $\square$

Proof (Completeness). 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 $\mathcal{T}$. We defer the proof of this claim to Appendix F.

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 inspection that this function satisfies the defining conditions of EBRFs. $\square$

F Sufficiency of Memory-less Schedulers

Please see Appendix D for the definition of schedulers. The scheduler generated in the proof of Theorem 1 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. 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.

Definition 3 (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 3 for an example.*

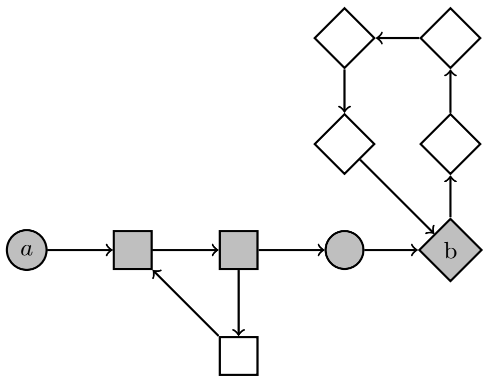

Fig. 3: An example finite path between two states a to b with its cycle-decomposition (C, β), where square and diamond states specify the cycles in C and the gray-scaled states specify β.

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 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).

G Soundness and Completeness of Universal Witnesses

Please see Appendix D for the definition of schedulers.

Lemma 3. *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 $c = [\pi(i), \pi(i+1), \ldots, \pi(j)]$ 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 2. We are now ready to prove the soundness and completeness theorem for UBRFs.

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 3: The template EBRF generated for the transition system in Figure 2

Proof (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.

Proof (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 $s \in \mathscr{S}$ let $d(s) : \mathscr{S} \to \mathbb{N}_0$ 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.

H Running Example of the Algorithm

Fixing Symbolic Templates. Consider the transition system in Figure 2 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 3 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.

Generating Entailment Constraints. 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).

Reduce Constraints to Quadratic Inequalities. The algorithm symbolically computes the following entailments as $\phi_i \Rightarrow \psi_j$ expressions from (3):

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

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

$$c_{1,1,0} + c_{1,1,1} \cdot (x_0^2+1) + c_{1,1,2} \cdot (x_0^2+1)^2 \leq c_{2,0,0} + c_{2,0,1} \cdot x_0 + c_{2,0,2} \cdot x_0^2 - 1$$

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:

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

Handling Initial Conditions. 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.

[††]This method is not only sound, but also complete. This is due to Putinar's Positivstellensatz [51]. See [1] for details.

Solving the System. The EBRF shown in the Example of Section 3 satisfies the above constraints and corresponds to the following solution to the QP instance Γ :

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

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

I Experimental results on Non-Linear Benchmarks

	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.