Making prophecies with decision predicates

(Extended Version)

Byron Cook **Eric Koskinen**

January 2011

Abstract

We describe a new algorithm for proving temporal properties expressed in LTL of infinite-state programs. Our approach takes advantage of the fact that LTL properties can often be proved more efficiently using techniques usually associated with the branching-time logic CTL than they can with native LTL algorithms. The caveat is that, in certain instances, nondeterminism in the system's transition relation can cause CTL methods to report counterexamples that are spurious with respect to the original LTL formula. To address this problem we describe an algorithm that, as it attempts to apply CTL proof methods, finds and then removes problematic nondeterminism via an analysis on the potentially spurious counterexamples. Problematic nondeterminism is characterized using *decision predicates*, and removed using a partial, symbolic determinization procedure which introduces new prophecy variables to predict the future outcome of these choices. We demonstrate—using examples taken from the PostgreSQL database server, Apache web server, and Windows OS kernel—that our method can yield enormous performance improvements in comparison to known tools, allowing us to automatically prove properties of programs where we could not prove them before.

1 Introduction

The common wisdom amongst users and developers of tools that prove temporal properties of systems is that the specification logic LTL [32] is more intuitive than CTL [10], but that properties expressed in the universal fragment of CTL ($\forall$CTL) without fairness constraints are often easier to prove than their LTL cousins [31, 43, 3][1]. Properties expressed in CTL without fairness can be proved in a purely syntax-directed manner using state-based reasoning techniques, whereas LTL requires deeper reasoning about whole sets of traces and the subtle relationships between families of them.

In this paper we aim to make an LTL prover for infinite-state programs with performance closer to what one would expect from a CTL prover. We use the observation that $\forall$CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don't always fit together: there are cases when, due to some instances

[1] Abadi and Lamport [3] make this point using the terminology of "refinement mappings" and "trace equivalence" instead of phrasing it in the context of temporal logics.

of nondeterminism in the transition system, ∀CTL alone is not powerful enough to prove an LTL property.

In these cases our LTL prover works around the problem using something we call *decision predicates*, which are used to characterize and treat such instances of nondeterminism. A decision predicate is represented as a pair of first-order logic formulae (a, b), where the formula a defines the decision predicate's presupposition (*i.e. when* the decision is made), and b characterizes the binary choice made when this presupposition holds. Any transition from state s to state s' in the system that meets the constraint $a(s) \wedge b(s')$ is distinguished by the decision predicate (a, b) from $a(s) \wedge \neg b(s')$.

We use decision predicates as the basis of a partial symbolic determinization procedure: for each predicate we introduce a new prophecy variable [3] to predict the future outcome of the decision. After partially determinizing with respect to these prophecy variables, we find that CTL proof methods succeed, thus allowing us to prove LTL properties with CTL proof techniques in cases where this strategy would have previously failed. To synthesize the decision predicates we employ a form of symbolic execution on spurious ∀CTL counterexamples together with an application of Farkas' lemma [22].

With our new approach we can automatically prove properties of infinite-state programs in minutes or seconds which were intractable using existing tools. Examples include code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Limitations. In practice, the applicability and performance of our technique is dependent on the heuristic used to choose new decision predicates when given an abstract representation of a specific point in a spurious counterexample. The predicate synthesis mechanism implemented in our tool is applicable primarily to infinite-state programs over arithmetic variables with commands that only contain linear arithmetic. However, no matter which predicate selection mechanism is used, our predicate-based determinization strategy is sound. Thus, unsound approximations to predicate synthesis could potentially be used in instances where the systems considered do not meet the constraints given above. Our technique is also based on an ∀CTL prover for infinite-state systems, which itself cannot be complete.

A further limitation is that our procedure is not well suited for finite-state model checking. The problem is that introducing prophecy variables greatly increases the number of state-holding elements required in usual finite-state encodings: Each prophecy variable must be capable of counting up to a number larger than the system's diameter [12][2]. In contrast, when using proof tools for infinite-state systems the performance cost for adding additional infinite-state variables is usually low.

Finally, our procedure critically depends on the full structure of counterexamples to ∀CTL properties, which are in the form of trees. Unfortunately, with only a few exceptions [13, 16] tools do not return whole tree counterexamples.

Related work. Our method complements more classical automata-theoretic approaches [44, 33] in which fairness constraints are used to encode linear-temporal conditions and then

[2]The problem is further exacerbated when we introduce multiple prophecy variables, as the nth prophecy variable must range over values as large as the diameter of the system which has been augmented with the first $n-1$ prophecy variables.

language emptiness—*a.k.a.* fair termination—is proved of the resulting system. The difficulty with language emptiness for infinite state systems (*e.g.* as implemented in previous work [15]) is that the mechanisms that allow us to ignore infinite executions not accepted by the fairness constraints are effectively the same as the expensive techniques used for proving termination. Thus, in practice, our previous tool [15] relies too heavily on termination proving machinery. In contrast, our new approach uses syntax-directed techniques for ∀CTL that depend much less on the performance of the underlying termination proving infrastructure. However, our strategy does rely on the assumption that, on average, the subtle correlations that are tracked only on-demand in our approach do not occur frequently. In cases where this assumption is not true, the cost of on-demand inference of decision predicates may be higher than simply using traditional techniques. We will see an example of this later in Section 6.

It is well known that determinization addresses the subtle semantic distinctions between linear-time and branching-time logics [38]. However, for infinite-state systems, open questions still remain if we hope to develop a practical determinization-based strategy: a) *what* to determinize, since complete determinization does not lead to a viable automatic tool for infinite-state systems, and b) *how* to determinize in a way that facilitates the application of current formal verification tools. We address these two questions in this paper.

Others have considered this trade-off between linear-time specifications and efficient branching-time verification procedures. For example, Cadence SMV [1] reduces LTL to CTL using additional fairness constraints [9, 14]. This technique still relies heavily on reasoning about fairness. This is a sensible engineering choice for finite-state systems for the reasons discussed above, but not for infinite-state systems. Schneider describes a method of translating an LTL formula into a semantically equivalent CTL formula [40]. However, this leads to an exponential blowup in the size of the CTL formula, and requires a modification to the model checking algorithm. Maidl identifies the subset of ∀CTL (called $\forall\text{CTL}^{\text{det}}$) which is expressible in LTL. Consequentially, for such formulae, an ∀CTL prover can be used [30]. By contrast, our decision predicate-based technique allows one to verify any LTL formula using branching-time proof techniques in such a way that performance is affected only in cases where tracking subtle correlations between traces is actually required.

Previous work has also examined different methods of representing systems [4, 6, 42] in order to facilitate proving linear-time temporal properties or proving linear-time properties of abstractions (*e.g.* pushdown systems [20, 41]). When model-checking is performed using explicit-state techniques [24, 27, 28] then the converse of our assumption is true: linear-time traces are in fact more naturally explored than branching-time executions in this context.

Our procedure uses several techniques found in the literature: namely prophecy variables [3] and Farkas' lemma [22]. We are of course not the first to use these techniques in applications related to the one addressed here. Prophecy variables have been used for many years to resolve nondeterminism in proofs, including some recent work [26, 37]. Our use of Farkas' lemma is similar to its use in rank function synthesis [34] and invariant generation [39].

```
PROVE_LTL(M, φ) :
    Ω := ∅
    let Φ = η(φ) in
    while true do
         let M^Ω = DETERMINIZE(M, Ω) in
         match PROVE_∀CTL(M^Ω, Φ) with
           | Succeed -> return Succeed
           | Fail(χ) ->
              let Ω' = REFINE(χ) in
                  if(Ω' = ∅)
                    let π ∈ χ in return Fail(π)
                  else
                    Ω := Ω ∪ Ω'
    done
```

Figure 1: Algorithm based on predicate determinization which implements LTL model checking (*i.e.* $M \models \varphi$). The procedures η, DETERMINIZE, REFINE and $\text{PROVE}_{\forall CTL}$ are defined in later sections.

2 Algorithm

Our LTL proof procedure, PROVE_{LTL}, is given in Figure 1. The algorithm is designed to iteratively find a sufficient set of decision predicates Ω such that proof tools for CTL can be used to prove an LTL property φ of the system M. The algorithm is based on four procedures which are each defined in later sections of the paper:

- η (Section 3) is a simple way of approximating an LTL formula φ with an analogous $\forall$CTL formula Φ in which universal operators are added in (*e.g.* F becomes AF, and G becomes AG). Without loss of generality we assume that negations have been pushed to the atomic propositions of the formula.

- DETERMINIZE (Section 4) takes a transition system M and a set of decision predicates Ω and returns a new partially determinized system M^Ω in which newly introduced prophecy variables are used to make predictions about the valuations of the decision predicates in Ω.

- REFINE (Section 5) takes an $\forall$CTL counterexample χ and, in the case that χ represents multiple distinct paths through the system, returns decision predicates which characterize the non-determinism that distinguishes between the different paths. In the case that χ represents only a single path through the system then REFINE returns $\varnothing$,

- $\text{PROVE}_{\forall CTL}$ (Section 6) is an $\forall$CTL-prover.

When $\Omega = \varnothing$, $\text{DETERMINIZE}(M, \Omega) = M$. Thus, on the first iteration of the loop our procedure is attempting to prove φ via a simple approximation Φ together with the original system M. When given a non-empty set of decision predicates, DETERMINIZE builds M^Ω by conjoining the original transition relation of M with a relation that specifies

the behavior of a prophecy variable for each decision predicate. For any set of decision predicates Ω, if Φ holds, then φ also holds. Thus, whenever we find a sufficient set of predicates to prove Φ, we have proved φ.

REFINE is used to determine if an $\forall$CTL-counterexample found by $\text{PROVE}_{\forall CTL}$ represents a real LTL-counterexample or something spurious. At first glance there is a formidable semantic gap between the two types of counterexamples: $\forall$CTL-counterexamples are trees, whereas LTL-counterexamples are traces. However, if all of the paths through the counterexample χ represent the same path or its prefixes, then any one of these paths is a legitimate counterexample to φ. In this case REFINE returns $\varnothing$. Otherwise, if χ represents more than one path in the program, REFINE returns a non-empty set of new decision predicates.

Example. Consider the LTL property $\mathsf{FG}(\mathsf{x} = 1)$, which informally can be read *"for every trace of the system,* $\mathsf{x} = 1$ *will eventually become true and stay true."* The meaning of the analogous $\forall$CTL property $\mathsf{AFAG}(\mathsf{x} = 1)$ is slightly more operational: *"On all paths emanating from an initial state, the system eventually reaches a state such that along all paths starting from this state,* $\mathsf{x} = 1$ *will be true and stay true."* For every transition system, if $\mathsf{AFAG}(\mathsf{x} = 1)$ holds, then $\mathsf{FG}(\mathsf{x} = 1)$ holds. Furthermore, our experience leads us to believe that proving $\mathsf{AFAG}(\mathsf{x} = 1)$ is often an efficient method of proving $\mathsf{FG}(\mathsf{x} = 1)$.

However, consider the following program, where $*$ represents nondeterministic choice:

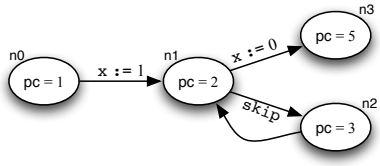
```
1    x := 1;
2    while (*) {
3        skip;
4    }
5    x := 0;
6    x := 1;
7    while (true) {
8        skip;
9    }
```

In this case $\mathsf{FG}(\mathsf{x} = 1)$ is valid, but unfortunately $\mathsf{AFAG}(\mathsf{x} = 1)$ is not. $\mathsf{FG}(\mathsf{x} = 1)$ is valid because, *for every individual program trace*, it is valid. For example, if a trace never leaves the loop at line 2, then the property is valid because $\mathsf{x} = 1$ before entering the loop. For the traces that do leave the loop, $\mathsf{x} = 1$ will become true at the command on line 6 and then remain true. The $\forall$CTL property is valid only if we can find a set of states that are eventually reached from the program's initial states such that $\mathsf{AG}(\mathsf{x} = 1)$ holds. In this case *no such set of states exists*, and tools for $\forall$CTL verification will return counterexamples to $\mathsf{AFAG}(\mathsf{x} = 1)$ that seemingly have no relation to to the original property $\mathsf{FG}(\mathsf{x} = 1)$.

The heart of the problem is the nondeterministic choice between the transition from line 2 to 3, and the transition from line 2 to 5: when we are in the loop at line 2 we cannot know if we will eventually leave the loop or not. We struggle when trying to decide if a state at location 2 is the point at which $\mathsf{x} = 1$ will be global true, as it is only *after* considering a full program trace that we would know (*i.e.* in this case we need to be looking at sets of traces, not sets of states).

We now illustrate the procedure in Figure 1 on this example. Let $\varphi = \mathsf{FG}(\mathsf{x} = 1)$ and M be the example program from above. Our procedure approximates φ with $\eta(\varphi) = \Phi =$

$\mathsf{AFAG}(\mathsf{x} = 1)$. As we described above, the program M does not respect the property Φ. The counterexample χ to Φ in M is an infinite tree which can be represented as a finite graph of transitions between program locations.



Here in this graph $\mathsf{pc} = 5$ indicates that the execution is at a state in which the program counter is at line 5. Our procedure uses REFINE to simultaneously symbolically simulate all possible paths through this graph and try to unify them into a single path through M. In this case it would begin its execution by visiting first $\mathsf{pc} = 1$ and then $\mathsf{pc} = 2$, after which it would discover that, for all paths of the graph to represent the same path, it must unify $\mathsf{pc} = 5$ and $\mathsf{pc} = 3$, which cannot be done. Thus, in this case, the $\forall$CTL counterexample χ will be deemed spurious to the LTL property and refinement Ω' will include the decision predicate $(\mathsf{pc} = 2, \mathsf{pc} = 5)$. [3] This decision predicate $(\mathsf{pc} = 2, \mathsf{pc} = 5)$ characterizes the choice: "when $\mathsf{pc} = 2$, will $\mathsf{pc}' = 5$ or not?" Notice also that, in this particular case, the predicates selected are over program locations, but this is not true in general (see Example 10 in Section 5).

The procedure then uses DETERMINIZE to generate M^Ω, which is effectively the cross product of M and a new transition relation which updates a new prophecy variable ρ based on the valuations of the decision predicate $(\mathsf{pc} = 2, \mathsf{pc} = 5)$:

$$\bigwedge \left\{ \begin{array}{l} s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) \neq 5 \Rightarrow s(\rho) \neq 0 \wedge s'(\rho) = s(\rho) - 1 \\ s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5 \Rightarrow s(\rho) = 0 \wedge s'(\rho) \in \mathbb{Z} \\ s(\mathsf{pc}) \neq 2 \Rightarrow s'(\rho) = s(\rho) \end{array} \right\}$$

We might try to express M^Ω in textual program code form as

```
ρ := *;
x := 1;
while (*) {
    assume(ρ ≠ 0);
    ρ := ρ - 1;
    skip;
}
assume(ρ = 0);
ρ := *;
x := 0;
x := 1;
while (true) {
    skip;
}
```

[3] An additional decision predicate will also be returned by our procedure, but it is not important for this example.

This new prophecy variable ρ predicts the outcomes of the decision predicate $(\mathsf{pc} = 2, \mathsf{pc} = 5)$. We initialize ρ to be an integer. For every given trace of the system, the concrete number chosen at the command "$\rho := *$" predicts the number of instances of the transition $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) \neq 5$ before we see a transition $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5$. The choice of a negative number (*e.g.* -1) represents the case where the execution will never see a $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5$ transition (*i.e.* non-termination)[4]. Whenever the program makes a transition $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) \neq 5$ it knows that $\rho \neq 0$, because the prophecy made previously does not allow it. The program also decrements ρ whenever we see a $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) \neq 5$ transition, for we know that (if we are going to see it at all) we are one step closer to seeing $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5$. If and when a $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5$ transition finally occurs, we know that $\rho = 0$. The program then predicts how many $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) \neq 5$ transitions will be visited the next time around until seeing another $s(\mathsf{pc}) = 2 \wedge s'(\mathsf{pc}) = 5$ transition (which will never occur in this example). Because the old prediction is not needed again, we can re-use the same variable ρ for the new prophecy.

With the prophecy variable ρ in place, there is now a set of states where $\mathsf{AG}(x = 1)$ holds:

$$\{s \mid (s(\rho) < 0 \wedge s(\mathsf{pc}) = 2) \vee s(\mathsf{pc}) = 6\}$$

Furthermore we can prove that this set of states is eventually reached. So we can now use $\forall$CTL where it previously failed. On the second iteration of the procedure from Figure 1, no $\forall$CTL-counterexample will be found in M^Ω and thus the LTL property φ has been proved of M.

Note that if we remove the second `x := 1` command from the example, then the property φ is false. In this case the variable ρ will uniquely determine the number of iterations through the first loop, and the counterexample returned will instead involve the second loop. This $\forall$CTL counterexample is also a valid LTL counterexample.

Preliminaries

In the later sections of this paper we define each of the sub-procedures used in Figure 1 (*i.e.* η in Section 3, DETERMINIZE in Section 4, etc). However, before moving to these more detailed descriptions we must develop some terminology and definitions that will be shared later.

States, sets, relations. We assume a domain D of states and, in the context of programs, will often treat it as a mapping from variables V to values. We will let s and t range over states, and S represent a set of states. We assume that no two states are indistinguishable. R will often be used to represent relations. When R is represented symbolically (*i.e.* expressed as a formula) it will be over the unprimed variables V and primed variables V'. For a state predicate p, the meaning $[\![p]\!]^{\mathsf{S}}$, is defined as the set of concrete states that respect p. The relational meaning of a formula p over primed and unprimed variables, $[\![p]\!]^{\mathsf{R}}$ is defined in the usual way. When it is clear from the context that we mean the real relation as opposed to the symbolic formula representation, we will drop the $[\![\ ]\!]^{\mathsf{R}}$ brackets. The notations Π^1 and Π^2 mean the first and second projection,

[4]In later sections we use a special element $\perp$ instead of negative numbers to represent non-termination, but for the purpose of this illustration negative numbers are easier.

respectively, of a relation. In cases where we are representing programs with control-flow graphs we will assume that states include a variable pc that represents the program counter and whose value is taken from a finite domain $\mathcal{L} = \{\ell_1, ..., \ell_n\}$.

Transition systems. We define a machine $M = (S, R, I)$ where $I \subseteq S$ is the set of initial states and $R \subseteq S \times S$ is the transition relation. In this paper we will be constructing new systems by adding variables and equating them to their original versions. Thus, it is convenient to build in a notion of internal and external state elements. We assume that $S = S^{\mathsf{ex}} \times S^{\mathsf{in}}$ (*i.e.* states consist of an external (visible) component and an internal component). We refer to an individual state as $\langle s, s^{\mathsf{in}} \rangle \in S$ and when a machine has no internal components, we omit the $\langle\rangle$ brackets.

Traces and paths. We define a *trace* to be a sequence of states

$$\pi = (\langle s_0, s_0^{\mathsf{in}} \rangle, \langle s_1, s_1^{\mathsf{in}} \rangle, ...)$$
$$\text{such that } \langle s_0, s_0^{\mathsf{in}} \rangle \in I \wedge \forall i \geq 0. (\langle s_i, s_i^{\mathsf{in}} \rangle, \langle s_{i+1}, s_{i+1}^{\mathsf{in}} \rangle) \in R$$

We denote $\mathsf{traces}(I, R)$ as the set of all such traces. For convenience, we do not allow finite traces – the transition relation must be such that every state s has at least one successor state. This is without a loss of generality, as final states can be encoded as states that loop back to themselves in the transition relation. With coinductive reasoning we can show that there exists an infinite trace from every state.

We use the notation $\pi|_{\mathsf{ext}}$ to denote the projection of π where internal components are removed:

$$(\langle s_0, s_0^{\mathsf{in}} \rangle, \langle s_1, s_1^{\mathsf{in}} \rangle, ...)|_{\mathsf{ext}} \triangleq (s_0, s_1, ...)$$

$\mathsf{traces}(I, R)|_{\mathsf{ext}}$ is similarly defined. We say that two systems are trace equivalent, notationally $\cong$, if their sets of projected traces are equivalent. We define an *abstract trace* to be a sequence of state abstractions. A *path* is a special case of an abstract trace in which only the pc-valuations are given. A path or an abstract trace is *spurious* if there does not exist a concrete trace from which we can construct the path via a projection.

Decision predicate vector. Formally we will treat Ω as a vector of pairs. Each element in the decision predicate vector Ω is a predicate pair denoted (a, b). We will use the vector index i to refer to a particular pair within Ω, and a_i, b_i denote the components of the ith pair. We use the notation $a_i(s)$ to indicate that s is in the set of states where a_i holds (i.e. $s \in [\![a_i]\!]$) and similar for $b_i(s)$.

3 Proving LTL with ∀CTL

In this section we describe an approximation η, which defines a sound over-approximation of LTL formulae with formulae in ∀CTL.

3.1 Linear Temporal Logic (LTL)

We use the following LTL grammar:

$$\varphi \quad ::= \quad \alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{G}\varphi \mid \mathsf{F}\varphi \mid \varphi \mathsf{W} \varphi$$

$$\frac{\alpha(\pi_0|_{\mathsf{ext}})}{\pi \vDash \alpha} \quad \frac{\pi \vDash \varphi \quad \vee \quad \pi \vDash \psi}{\pi \vDash \varphi \vee \psi} \quad \frac{\pi \vDash \varphi \quad \pi \vDash \psi}{\pi \vDash \varphi \wedge \psi}$$

$$\frac{\exists i \geq 0. \pi^i \vDash \varphi}{\pi \vDash \mathsf{F}\varphi} \quad \frac{\forall i \geq 0. \pi^i \vDash \varphi \quad \vee \quad \exists j \geq 0. \pi^j \vDash \psi \wedge \forall i < j. \pi^i \vDash \varphi}{\pi \vDash \varphi \mathsf{W} \psi}$$

Figure 2: Semantics of LTL: $\vDash$

$$\frac{\alpha(s|_{\mathsf{ext}})}{s \vDash \alpha} \quad \frac{s \vDash \Phi \quad s \vDash \Psi}{s \vDash \Phi \wedge \Psi} \quad \frac{s \vDash \Phi \quad \vee \quad s \vDash \Psi}{s \vDash \Phi \vee \Psi}$$

$$\frac{\forall (s_0, s_1, ...) \in \mathsf{traces}(I, R). \exists i \geq 0.\ s_0 = s \Rightarrow s_i \vDash \Phi}{s \vDash \mathsf{AF}\Phi}$$

$$\frac{\begin{array}{c}\forall (s_0, s_1, ...) \in \mathsf{traces}(I, R). \\ \forall i \geq 0.\ s_i \vDash \Phi \quad \vee \quad \exists j \geq 0.\ s_j \vDash \Psi \wedge \forall 0 \leq i < j.\ s_i \vDash \Phi\end{array}}{s \vDash \mathsf{A}[\Phi \mathsf{W} \Psi]}$$

Figure 3: Semantics of ∀CTL: $\vDash$

We have not included $\mathsf{U}, \mathsf{R}, \mathsf{X}$ or $\neg$. Without loss of generality we assume that negations appear only in atomic propositions (*i.e.* instances of $\neg$ have been pushed to the leaves of the formula). In the context of programs X is relatively useless and is easily subsumed by F. U and R can be encoded as follows:

$$\begin{aligned} \varphi \mathsf{U} \psi &\triangleq \mathsf{F}\psi \wedge (\varphi \mathsf{W} \psi) \\ \varphi \mathsf{R} \psi &\triangleq \psi \mathsf{W} (\varphi \wedge \psi) \end{aligned}$$

The LTL semantics, notationally $\vDash$, are given in Figure 2. The notation π^i indicates a *suffix* of a trace starting at the ith state in the sequence. We use π_0 to denote the first element in π. The superscript binds tighter than the subscript, i.e. $\pi_0^i = (\pi^i)_0$.

An atomic proposition α is from some abstract domain D, and we assume that $\mathsf{true}, \mathsf{false} \in D$ and that D is closed under negation (*i.e.* $\forall \alpha \in D.\ \exists \beta \in D.\ [\![\beta]\!]^{\mathsf{S}} = [\![\neg\alpha]\!]^{\mathsf{S}}$). The operator $\mathsf{G}\varphi$ specifies that φ globally holds along all traces. The operator $\mathsf{F}\varphi$ specifies that along every trace, eventually a suffix will be reached where φ holds. Finally, the $\varphi \mathsf{W} \psi$ operator specifies that φ holds forever or φ holds until ψ holds.

The LTL entailment relation $\vDash$ is defined on traces: the relation $\pi \vDash \varphi$ indicates that φ holds for a given trace π. We now lift $\vDash$ to machines.

Definition 3.1 (LTL Machine Entailment)**.** *Assume that $M = (S, R, I)$. We define LTL-entailment, notationally $M \vDash \varphi$, as*

$$\forall \pi \in \mathit{traces}(I, R)\ .\ \pi \vDash \varphi$$

3.2 Computation Tree Logic (∀CTL)

We now review existential-free computation tree logic:

$$\Phi ::= \alpha \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathsf{AG}\Phi \mid \mathsf{AF}\Phi \mid \mathsf{A}[\Phi \mathsf{W} \Phi]$$

The semantics of ∀CTL ⊨ are given in Figure 3. Unlike the trace-based LTL semantics, ∀CTL's semantics are state-based. By this we mean that the temporal operators are state-based in structure: the derivation of a given formula is per-state and depends on the derivation of subformulae for subsequent states. The operator $\mathsf{AF}\Phi$ specifies that across all computation sequences from the current state, that there is a reachable state in which Φ holds. Finally, the $\mathsf{A}[\Phi \mathsf{W} \Psi]$ operator specifies that Φ holds in every state where Ψ does not hold yet. Note that $\mathsf{AG}\Phi = \mathsf{A}[\Phi \ \mathsf{W} \ \mathsf{false}]$.

Definition 3.2 (∀CTL Machine Entailment)**.** *As we did for* ⊨*, we lift* ⊨ *to machines. Assume* $M = (S, R, I)$*. We define* ∀*CTL-entailment, notationally* $M \vDash \Phi$*, to be*

$$\forall s \in I \ . \ R,s \vDash \varphi$$

3.3 Over-approximating LTL with ∀CTL

We describe a simple syntactic conversion from a formula in LTL to its corresponding over-approximation in ∀CTL.

Definition 3.3. *The abstraction function* $\eta : \varphi \to \Phi$ *is defined as follows:*

$$\begin{array}{lcl} \eta(\alpha) & = & \alpha \\ \eta(\varphi \wedge \psi) & = & \eta(\varphi) \wedge \eta(\psi) \\ \eta(\varphi \vee \psi) & = & \eta(\varphi) \vee \eta(\psi) \\ \eta(\mathsf{G}\varphi) & = & \mathsf{AG}\,\eta(\varphi) \\ \eta(\mathsf{F}\varphi) & = & \mathsf{AF}\,\eta(\varphi) \\ \eta(\varphi \,\mathsf{W}\, \psi) & = & \mathsf{A}[\eta(\varphi) \ \mathsf{W}\, \eta(\psi)] \end{array}$$

Lemma 3.1. *(*∀*CTL Approximation) For a machine* M *and LTL property* φ*,*

$$M \vDash \eta(\varphi) \quad \Rightarrow \quad M \vDash \varphi$$

4 Decision Predicate Determinization

In this section we describe the procedure Determinize, which uses decision predicates as it performs a symbolic form of partial determinization.

Partially determinized machines. Figure 4 contains the definition for Determinize, which is designed to return a partially determinized machine when given a vector of predicates Ω and a machine:

$$M^{\Omega} = \textsc{Determinize}(M, \Omega)$$

The new machine M^{Ω} includes additional prophecy variables denoted ρ_i. These correspond to the predicate pairs (a_i, b_i) in the vector Ω. In accordance with I^{Ω} these variables are free to be a positive integer or zero or $\perp$ in the initial state. We will see that the choice of initial values (and the choice in Eqn. 3 from Figure 4) is the driving force behind determinization. For simplicity we used $\mathbb{Z}$ instead of $\mathbb{N}_{\perp}$ in Section 2. We also now define the update relation differently than we did in Section 2, in the sense that in Figure 4

DETERMINIZE$((S, R, I), \Omega) = (S^\Omega, R^\Omega, I^\Omega)$ where

$$
\begin{aligned}
S^\Omega &= S \times \overrightarrow{\mathbb{N}_\perp} \quad \text{denoted } \langle s, \rho \rangle \\
I^\Omega &= I \times \overrightarrow{\mathbb{N}_\perp} \\
R^\Omega &= \{(\langle s, \rho \rangle, \langle s', \rho' \rangle) \mid (s, s') \in R \wedge \forall (a_i, b_i) \in \Omega.
\end{aligned}
$$

$$
\begin{aligned}
[a_i(s) \wedge \rho_i = \perp &\Rightarrow b_i(s') \wedge \rho_i' = \perp] && (1)\\
\wedge [a_i(s) \wedge \rho_i > 0 &\Rightarrow b_i(s') \wedge \rho_i' = \rho_i - 1] && (2)\\
\wedge [a_i(s) \wedge \rho_i = 0 &\Rightarrow \neg b_i(s') \wedge \rho_i' \in \mathbb{N}_\perp] && (3)\\
\wedge [\neg a_i(s) &\Rightarrow \rho_i' = \rho_i]\} && (4)
\end{aligned}
$$

and $\mathbb{N}_\perp \triangleq \mathbb{N} \cup \{\perp\}$.

Figure 4: The DETERMINIZE procedure which, when given a vector of predicate pairs Ω, constructs the corresponding predicate-determinized machine.

the unprimed variables appear only to the left of $\Rightarrow$ and primed variables appear only to the right. While the two formalizations are equivalent, the encoding in Figure 4 is conceptually more operational and easier to implement within a tools setting, where in practice we are modifying the existing transition relation of M.

Transitions in R^Ω are made in accordance with R, but constrained by the values of ρ when states are reached that match a decision predicate (a_i, b_i) in Ω. Specifically, when a state is reached where a_i holds and the prophecy variable $\rho_i = \perp$, then b_i must hold in the next state and ρ_i is unchanged (Eqn. 1 of Figure 4). This rule corresponds to behaviors where a $a_i(s)$ state is visited infinitely often. Alternatively, if $\rho_i > 0$ (Eqn. 2) then b_i must also hold in the next state, except that ρ_i is decremented. When ρ_i reaches zero, then $\neg b_i$ must hold in the next state and ρ_i is free to take a new value from $\mathbb{N}_\perp$, starting the process all over (Eqn. 3). Finally, when a_i doesn't hold of a particular state, ρ_i is unchanged (Eqn. 4).

The prophecy variables introduced here trade *nondeterminism in the transition relation* R for a larger, *nondeterministic state space*. The state space nondeterminism is either determined at machine initialization by the initial choice of values for ρ given by I^Ω, or else later in a trace (Eqn. 3) by choosing new nondeterministic values for ρ. This lazy selection of nondeterministic values means that M^Ω needn't consist of infinitely many prophecy variables for each predicate pair. This formulation restricts us to treat programs with only countable nondeterminism. One could conceive of more powerful forms of nondeterminism, but we intend to use this technique in the context of programs for which countable nondeterminism is sufficient.

Theorem 4.1. *For all* Ω, $M^\Omega \cong M$.

Proof. The theorem holds if each of the conditions **P1**, **P2**, **P3** and **P4** and **PB** described below are met. These conditions are a variation of Proposition 5 from Abadi and Lamport [3]. Conditions **P1**, **P2**, **P3** and **P4** directly match Abadi and Lamport's conditions. We omit Condition **P5** as it involves liveness restrictions on the behavior

of machines and we assume that our machines have no liveness restrictions. We loosen the restriction of Abadi and Lamport's **P6** with **PB** (detailed below), as our prophecy variables do not respect the condition of finite nondeterminism. The new condition **PB** is in fact a consequence of **P6**: in the second part of the proof, Abadi and Lamport show that all the behaviors of M are contained within M^Ω (note that regardless of superscript, $P = M$ because $L = \mathsf{true}$). Part 2.1 defines a directed graph, and then introduces Claim 2.1, which is not true in our setting. However, Claim 2.1 is only used in conjunction with Claim 2.2 and König's Lemma in order to prove Claim 2.3. In our setting we have simply included Claim 2.3 as condition **PB**.

We now describe why each condition holds:

(P1) $S^\Omega \subseteq S \times S^P$ for some S^P. $\surd$

(P2) $I^\Omega = \Pi_p^{-1}(I)$ where Π_p^{-1} maps $S \times S^\Omega$ onto S. $\surd$

(P3) If $((s,p),(s',p')) \in R^\Omega$ then $(s,s') \in R$ or $s = s'$. This holds by construction of R^Ω from R. $\surd$

(P4) If $(s,s') \in R$ and $(s',p') \in S^\Omega$ then there exists $p \in S^\Omega$ such that $((s,p),(s',p')) \in R^\Omega$. Again, this holds by construction of R^Ω from R, case splitting on the value of p' and quantifying over $i \in \Omega$. $\surd$

(PB) For every $(s_0,s_1,...) \in \mathsf{traces}(I,R)$ there exists $(p_0,p_1,...)$ such that $((s_0,p_0),(s_1,p_1),...) \in \mathsf{traces}(I^\Omega,R^\Omega)$. Quantifying over each $i \leq |\Omega|$, consider all of the (possibly infinitely many) transitions (s_j,s_{j+1}) such that $a_i(s_j)$ holds. Now for each transition $b_i(s_{j+1})$ may or may not hold. This can be modeled by:

$$(\exists m.\ b_i^m \neg b_i)^{\infty|*}(b_i^\infty)$$

i.e. a head $(\exists m.\ b_i^m \neg b_i)^{\infty|*}$ consisting of repeated instances of finitely many b_i-states and a single $\neg b_i$-state, and a tail consisting of infinitely many b_i-states. So we can choose ρ_i accordingly, setting $\rho_i = m$ in each (potentially zero or infinitely many) instances of the head, and setting $\rho_i = \bot$ in the tail. $\surd$

□

Example 5. *(Nondeterministic Choice) Consider the following machine:*

$$\begin{array}{rcl} S & = & \left[\begin{smallmatrix}\mathbb{N}\\ \mathbb{N}\end{smallmatrix}\right] \quad \textit{denoted} \quad \left[\begin{smallmatrix}\mathsf{x}\\ \mathsf{y}\end{smallmatrix}\right] \\ I & = & \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right] \\ R & = & \{(\left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right],\left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right]),\ (\left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right],\left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right]),\ (\left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right],\left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right]),\ (\left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right],\left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right])\} \end{array}$$

In this transition relation there is nondeterminism in the first transition. We can determinize this with the predicates $a = (\mathsf{x} = 0 \wedge \mathsf{y} = 0)$ *and* $b = (\mathsf{x} = 1)$*. With these predicates we can construct the corresponding* M^Ω*.*

$$\begin{array}{rcl} S^\Omega & = & \left[\begin{smallmatrix}\mathbb{N}\\ \mathbb{N}\end{smallmatrix}\right] \times \mathbb{N}_\bot \quad \textit{denoted} \quad \langle \left[\begin{smallmatrix}\mathsf{x}\\ \mathsf{y}\end{smallmatrix}\right], \rho \rangle \\ I^\Omega & = & \langle \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right], \mathbb{N}_\bot \rangle \\ R^\Omega & = & \{(\langle \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right], 1\rangle, \langle \left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right], 0\rangle), (\langle \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right], 0\rangle, \langle \left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right], \mathbb{N}_\bot\rangle), \\ & & (\langle \left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right], 0\rangle, \langle \left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right], 0\rangle), (\langle \left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right], \mathbb{N}_\bot\rangle, \langle \left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right], \mathbb{N}_\bot\rangle)\} \end{array}$$

The first two transitions have now been determinized: from the initial state, depending on the initial choice of ρ, either $(\mathsf{x} = 1)$ or $\neg(\mathsf{x} = 1)$ will hold in the next state. In this example, since the nondeterministic transition only happens once, the (external) behaviors when $\rho > 1$ or $\rho = \bot$ in the initial state are all equivalent to $\rho = 1$ in the initial state so, for presentation purposes, we have omitted them. The additional behaviors will be used in the next example.

Example 6. *(Termination) Consider the following infinite-state system which we represent symbolically*

$$\begin{array}{rcl} S & = & \mathbb{N} \quad \textit{denoted} \quad \mathsf{x} \\ I & = & \mathbb{N} \\ R & = & [\![(\mathsf{x} > 0 \wedge \mathsf{x}' = \mathsf{x} + 1) \vee (\mathsf{x} > 0 \wedge \mathsf{x}' = 0) \vee (\mathsf{x} = 0 \wedge \mathsf{x}' = 0)]\!]^R \end{array}$$

In this transition relation, when $\mathsf{x} > 0$ initially, there is nondeterminism in how many times the first transition is chosen before the second transition is chosen. We can determinize this with the predicates $a = (\mathsf{x} > 0)$ and $b = (\mathsf{x} > 0)$, constructing the corresponding M^Ω as follows:

$$\begin{array}{rcl} S^\Omega & = & \mathbb{N} \times \mathbb{N}_\bot \\ I^\Omega & = & \mathbb{N} \times \mathbb{N}_\bot \\ R^\Omega & = & [\![(\mathsf{x} > 0 \wedge \rho = \bot \wedge \mathsf{x}' = \mathsf{x} + 1 \wedge \rho' = \bot) \vee \\ & & (\mathsf{x} > 0 \wedge \rho > 0 \wedge \mathsf{x}' = \mathsf{x} + 1 \wedge \rho' = \rho - 1) \vee \\ & & (\mathsf{x} > 0 \wedge \rho = 0 \wedge \mathsf{x}' = 0 \wedge \rho' \in \mathbb{N}_\bot) \vee \\ & & (\mathsf{x} = 0 \wedge \rho \in \mathbb{N}_\bot \wedge \mathsf{x}' = 0 \wedge \rho' \in \mathbb{N}_\bot)]\!]^R \end{array}$$

In M^Ω the first choice of how many times a transition from $[\![\mathsf{x} > 0 \wedge \mathsf{x}' = \mathsf{x} + 1]\!]^R$ is taken is given by the choice of an initial value for ρ. Any finite number of iterations corresponds to an arbitrarily chosen numeric value of ρ. The case where the transition is taken infinitely many times corresponds to the initial choice of $\bot$ for ρ.

Example 7. *(Running example) For the example given in Section 2, the state space of the original program is $S = \{\ell_1, ..., \ell_9\} \times \{0, 1\}$ denoted pc, x. For $\Omega = \{(\mathsf{pc} = \ell_2, \mathsf{pc} = \ell_5)\}$, we have one prophecy variable denoted ρ, so $S^\Omega = S \times \mathbb{N}_\bot$ and $I^\Omega = \{\ell_1\} \times \{1\} \times \mathbb{N}_\bot$. The transition relation (omitting some uninteresting arcs) is defined as follows:*

$$\begin{array}{rcl} R^\Omega & = & [\![(\mathsf{x} = 1 \wedge \mathsf{pc} = \ell_2 \wedge \rho = \bot \wedge \mathsf{x}' = 1 \wedge \mathsf{pc}' = \ell_2 \wedge \rho' = \bot) \vee \\ & & (\mathsf{x} = 1 \wedge \mathsf{pc} = \ell_2 \wedge \rho > 0 \wedge \mathsf{x}' = 1 \wedge \mathsf{pc}' = \ell_2 \wedge \rho' = \rho - 1) \vee \\ & & (\mathsf{x} = 1 \wedge \mathsf{pc} = \ell_2 \wedge \rho = 0 \wedge \mathsf{x}' = 1 \wedge \mathsf{pc}' = \ell_5 \wedge \rho' \in \mathbb{N}_\bot) \vee \\ & & ...]\!]^R \end{array}$$

4.1 Proving LTL with ∀CTL and determinization

Lemma 3.1 shows that one can prove LTL properties with an ∀CTL verifier and an unmodified transition relation. We now extend this to show that one can prove (*perhaps even more*) LTL properties with ∀CTL and a predicate-determinized machine.

$$\frac{\alpha(s|_{\mathsf{ext}})}{\mathsf{vd_C}\ s\ \alpha\ (\mathsf{CEX}_\alpha\ s)} \qquad \frac{\mathsf{vd_C}\ s\ \Phi\ \chi_1 \quad \mathsf{vd_C}\ s\ \Psi\ \chi_2}{\mathsf{vd_C}\ s\ (\mathsf{CEX}_\vee\ \chi_1, \chi_2)\ \Phi \vee \Psi}$$

$$\frac{\mathsf{vd_C}\ s\ \Phi\ \chi}{\mathsf{vd_C}\ s\ \Phi \wedge \Psi\ (\mathsf{CEX}_\wedge\ \chi)} \qquad \frac{\mathsf{vd_C}\ s\ \Psi\ \chi}{\mathsf{vd_C}\ s\ \Phi \wedge \Psi\ (\mathsf{CEX}_\wedge\ \chi)}$$

$$\frac{\mathsf{vd_C}\ s_n\ \Phi\ \chi \quad \pi = (s, s_1, ..., s_n)}{\mathsf{vd_C}\ s\ \mathsf{AG}\Phi\ (\mathsf{CEX}_{\mathsf{AG}}\ \pi, \chi)}$$

$$\frac{\pi = (s, ..., s_n) \quad \tilde{\pi} = (s_n, s_{n+1}, ...) \quad \forall i \geq n.\ \mathsf{vd_C}\ s_i\ \Phi\ \chi}{\mathsf{vd_C}\ s\ \mathsf{AF}\Phi\ (\mathsf{CEX}_{\mathsf{AF}}\ \pi, \tilde{\pi}, \chi)}$$

$$\frac{\mathsf{vd_C}\ s_n\ \Phi\ \chi_1 \quad \mathsf{vd_C}\ s_n\ \Psi\ \chi_2 \quad \pi = (s, s_1, ..., s_n)}{\mathsf{vd_C}\ s\ \mathsf{A}[\Phi \mathsf{W} \Psi]\ (\mathsf{CEX}_{\mathsf{W}}\ \pi, \chi_1, \chi_2)}$$

Figure 5: Validity $\mathsf{vd_C}$ of an ∀CTL counterexample χ for a property Φ from a state s.

Theorem 4.2. *(∀CTL Approximation with Determinization) For a machine M, LTL property φ and predicates Ω,*

$$M^\Omega \vDash \eta(\varphi) \quad \Rightarrow \quad M \vDash \varphi$$

Proof. Lemma 3.1 says that $M \vDash \eta(\varphi) \Rightarrow M \vDash \varphi$ The process of predicate determinization constructs machine M^Ω from M such that $M^\Omega \cong M$. Since the two machines are trace equivalent and it is known that trace-equivalent machines have the same LTL-behavior, Lemma 3.1 applies to the new machine and hence the theorem holds. □

5 Decision Predicate Refinement

We now describe Refine, our procedure which examines counterexamples from a branching-time verification tool and discovers predicates which characterize the nondeterministic branching within them if any nondeterminism exists.

∀CTL counterexamples. Counterexamples in ∀CTL are trees [13]. The shape of the tree depends on the shape of the property which is violated. While most tools typically do not annotate their counterexamples with subformula, they could be made to do so. We formalize an ∀CTL counterexample tree as follows:

Definition 5.1. *(∀CTL tree counterexample)*

$$\begin{array}{rcl} \chi & ::= & \mathit{CEX}_\alpha\ \mathit{of}\ \pi \\ & | & \mathit{CEX}_\wedge\ \mathit{of}\ \chi \\ & | & \mathit{CEX}_\vee\ \mathit{of}\ \chi \times \chi \\ & | & \mathit{CEX}_{AG}\ \mathit{of}\ \pi \times \chi \\ & | & \mathit{CEX}_{AF}\ \mathit{of}\ \pi \times \pi \times \chi \\ & | & \mathit{CEX}_W\ \mathit{of}\ \pi \times \chi \times \chi \end{array}$$

$$\textsc{PSynth}_D(R, R') = \begin{cases} \{(a,b),(a,\neg b)\} & \text{such that } R \subseteq [\![a \wedge b']\!]^{\mathsf{R}} \text{ and } R' \subseteq [\![a \wedge \neg b']\!]^{\mathsf{R}} \\ \varnothing & \text{if no such } a, b \text{ exist} \end{cases}$$

Figure 6: Specification of $\textsc{PSynth}_D$ which, when given symbolic representations of two relations, returns a predicate pair that distinguishes them. An implementation of this procedure is described in Section 6.

The constructors of an ∀CTL counter example are given in the above definition, and the validity predicate vd_{C} is given in Figure 5. In the above definition there is a constructor for each structural element of an ∀CTL formula. A counterexample to an atomic proposition CEX_{α} is a (single state) trace where the atomic proposition does not hold of the first element. A counterexample to a conjunction $\mathsf{CEX}_{\wedge}$ is a counterexample to one of the conjuncts. A counterexample to a disjunction $\mathsf{CEX}_{\vee}$ is comprised of two counterexamples, one for each disjunct. A counterexample $\mathsf{CEX}_{\mathsf{AG}}$ is a path to a state in which a counterexample exists for the subformula. A counterexample $\mathsf{CEX}_{\mathsf{AF}}$ is a "stem" path to an infinite "lasso" loop where a counterexample exists for the subformula. A counterexample $\mathsf{CEX}_{\mathsf{W}}$ is a path to a state where a counterexample exists for both subformulae. For example, the counterexample to the property $\mathsf{AF}((\mathsf{AG}p) \vee (\mathsf{AG}q))$ consists of a stem and loop (for the AF subformula), and from within the loop a stem for each AG subformula.

Equality $\doteq$ between counterexamples is inductively defined, lifting equality between traces. We denote by $\chi|_{\mathsf{ext}}$ the counterexample which consists of the external projection of paths in all components. Often counterexamples from model checking tools may contain less information than actual concrete traces (*e.g.* SLAM returns abstract traces that include the valuations of pc together with the valuations of the predicates used during the failed proof attempt).

In the AF rule, the counterexample is represented as a "stem" with an infinitely-repeated "lasso" path, along which every subtree is a counterexample to the subformula. In reality, not all counterexamples to termination can be represented this way. There are some rare programs that do not terminate but whose counterexamples cannot be represented as a infinitely-repeated "lasso path."[5] In this case we assume an approximation of the real counterexample has been found and has been encoded using $\mathsf{CEX}_{\mathsf{AF}}$. In some instances this could potentially lead to divergence in our tool.

Counterexample control-flow graphs. From a given counterexample χ, we can construct a corresponding counterexample flow-graph (CEFG) Γ which represents all paths in the counterexample. We use a standard graph-based notation, where nodes $\mathsf{n} \in \mathbb{N}$ correspond to states in the counterexample, and edges are triples $(\mathsf{n}_1, r, \mathsf{n}_2)$ consisting of a starting node, a transition relation r from the counterexample and a destination node.

[5] Here is an example of a non-terminating program without an infinitely-repeated "lasso path":

```
while x > 0
    y:= x;
    x:= x+ 1;
    while y > 0
        y:= y- 1;
```

```
REFINE(χ) :
    S := ∅
    N := {n_0}
    let Γ = cefg(χ|_ext) in
    while true do
         let N' = {n' | n ∈ N ∧ ∃(n, r, n') ∈ Γ} in
         let T = {r | n ∈ N ∧ ∃n'.(n, r, n') ∈ Γ} in
         let Ω = ⋃_{r,r'∈T} PSYNTH_D(r, r') in
           if Ω = ∅ then
              if N' ∪ S = S then
                 return ∅
              else
                 N := N'
                 S := S ∪ N'
           else
               return Ω
    done
```

Figure 7: The REFINE procedure walks down a counterexample flow-graph, at each step simultaneously exploring all possible next steps. If any pair of possible next steps are distinguishable via a predicate from PSYNTH_D then that predicate is immediately returned.

$$
\begin{array}{lll}
\texttt{cefg}(\mathsf{n}_0, \chi) \triangleq & \text{match } \chi \text{ with} & \\
& \mid \mathsf{CEX}_\alpha\ s & \longrightarrow\ (\mathsf{n}_0, Id, \mathsf{n}_0) \\
& \mid \mathsf{CEX}_\wedge\ \chi_1 & \longrightarrow\ \texttt{cefg}(\mathsf{n}_0, \chi_1) \\
& \mid \mathsf{CEX}_\vee\ \chi_1, \chi_2 & \longrightarrow\ \texttt{cefg}(\mathsf{n}_0, \chi_1) \cup \texttt{cefg}(\mathsf{n}_0, \chi_2) \\
& \mid \mathsf{CEX}_{\mathsf{AG}}\ \pi, \chi_1 & \longrightarrow\ \texttt{cefg}_\pi(\mathsf{n}_0, \mathsf{n}_x) \cup \texttt{cefg}(\mathsf{n}_x, \chi_1) \\
& \mid \mathsf{CEX}_{\mathsf{AF}}\ \pi, \tilde{\pi}, \chi_1 & \longrightarrow\ \texttt{cefg}_\pi(\mathsf{n}_0, \mathsf{n}_1) \cup \texttt{cefg}_{\tilde{\pi}}(\mathsf{n}_1, \mathsf{n}_1) \cup\ \texttt{cefg}(\mathsf{n}_1, \chi_1) \\
& \mid \mathsf{CEX}_{\mathsf{W}}\ \pi, \chi_1 & \longrightarrow\ \texttt{cefg}_\pi(\mathsf{n}_0, \mathsf{n}_x) \cup \texttt{cefg}(\mathsf{n}_x, \chi_1)
\end{array}
$$

where $\mathsf{n}_x, \mathsf{n}_1$ are fresh.

$$\texttt{cefg}_\pi(\mathsf{n}_0, \mathsf{n}_x) \triangleq \{(\mathsf{n}_i, r, \mathsf{n}_{i+1}) \mid 0 \le i < |\pi| \wedge (\pi_0^i, \pi_0^{i+1}) \in [\![r]\!]^{\mathsf{R}}\}$$

where each n_i is fresh and $\mathsf{n}_x = \mathsf{n}_{|\pi|}$

Figure 8: The `cefg` procedure consumes a counterexample and constructs a counterexample flow graph, using $\texttt{cefg}_\pi$ to convert a path π to a graph component.

Even when we are working with programs, these CEFGs are different from program CFGs because they represent possible state transitions: there may be multiple CEFG transitions for a single CFG transition (e.g. when the program involves nondeterministic assignment) and there maybe multiple CEFG nodes which have the same CFG node. A counterexample flow graph can be constructed from a counterexample via the translation shown in Figure 8.

Predicate synthesis. The procedure $\text{PSynth}_D(r, r')$ is specified in Figure 6. It consumes two transition relations R, R' and returns two pairs of decision predicates. The implementation of PSynth_D will differ, depending on the context (*i.e.* finite-state systems expressed at the bit-level, infinite-state systems expressed over linear arithmetic, etc). We assume that for a given domain D (a) that D is capable of distinguishing two states and (b) that PSynth_D is capable of discovering sufficient elements in D to do so. If these assumptions do not hold then in some instances our technique may be unable to sufficiently determinize. In our implementation, described in Section 6, we use constraint-solving techniques to find predicates which are monomials over linear inequalities.

Symbolic tree execution. The recursive procedure Refine, given in Figure 7, consumes an ∀CTL counterexample and returns sets of predicates which distinguish nondeterministic branching. This involves first constructing a counterexample flow-graph, and iteratively exploring the frontier. Refine simultaneously steps down each possible branch of the counterexample, ensuring that all of the next states are equivalent using PSynth_D (see the PSynth_D specification in Figure 6) to find distinguishing predicates. When distinct states are found, the corresponding predicates are returned, so that they can be added to Ω and the main algorithm can reiterate.

Progress. We now show that for a given counterexample χ, if Refine discovers predicates, then our algorithm produces a new machine for which $\chi|_{\mathsf{ext}}$ is not a counterexample. We also show that, if no predicates are found by Refine, then a real counterexample to the original LTL property can be constructed from $\chi|_{\mathsf{ext}}$.

Lemma 5.1. *(Counterexample elimination) For a machine M^Ω, property φ,*

$$\begin{array}{c} \textit{if } \chi \textit{ is a counterexample to } M^\Omega \vDash \eta(\varphi) \\ \textit{then } \nexists \textit{ counterexample } \chi' \textit{ to } M^{\Omega'} \vDash \eta(\varphi) \\ \textit{such that } \chi|_{\mathsf{ext}} \doteq \chi'|_{\mathsf{ext}} \end{array}$$

where $\Omega' = \Omega \cup \text{Refine}(\chi)$ and $\text{Refine}(\chi) \neq \varnothing$.

Proof. Let $(a_i, b_i) \in \text{Refine}(\chi)$. By definition of Refine in Figure 7 this predicate pair must have come from a subcomponent of the counterexample χ flow graph of the form $(\mathsf{n}, r, \mathsf{n}'), (\mathsf{n}, r', \mathsf{n}'')$. Moreover $a_i(\Pi^1(r))$, $b_i(\Pi^2(r))$ and $\neg b_i(\Pi^2(r'))$. Now, in the new machine the prophecy vector is augmented with a new element ρ_i. So the set of states denoted $\langle \Pi^1(r), \rho \rangle$ have either $\rho_i = 0$ or $\rho_i \in \{\bot, 1, 2, ...\}$. According to $R^{\Omega'}$, either $(\langle \Pi^1(r), \rho \rangle, \langle \Pi^2(r), \rho \rangle)$ is enabled or else $(\langle \Pi^1(r), \rho \rangle, \langle \Pi^2(r'), \rho \rangle)$ is enabled, but not both. Hence, there is no valid counterexample χ' such that $\chi|_{\mathsf{ext}} \doteq \chi'|_{\mathsf{ext}}$. □

Remark on completeness. There are a few impediments to making a completeness claim. First, for a given ∀CTL counterexample χ, the routine PSYNTH$_D$ must be able to discover predicates to characterize nondeterminism in χ. However, since we use approximation (*e.g.* with linear arithmetic), it will not always be able to discover sufficient predicates when the exist.

Second, even when we have a perfect PSYNTH$_D$ routine, some ∀CTL counterexamples may be spurious, as the underlying ∀CTL also supports only overapproximation in linear arithmetic. Consequently, when REFINE$(\chi) = \varnothing$ we cannot necessarily claim that we have a valid LTL counterexample. Furthermore, as mentioned previously, there are some non-terminating programs that do not have a infinitely-repeated "lasso path." In these instances, the ∀CTL tool itself will either hang or return spurious counterexamples.

Finally, it is unclear whether our refinement loop will discover a finite number of decision predicates. With an infinite predicate vector Ω^∞, all nondeterminism can be represented (given a sufficient predicate domain), but one would hope that for each program/property there is a finite predicate vector.

All of the above issues are the subject of ongoing investigation.

Example 8. *(Running Example) For the example in Section 2, an ∀CTL prover may generate the following counterexample:*

$$(CEX_{AF}\ \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}2\\1\end{smallmatrix}\right], \left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}3\\1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}2\\1\end{smallmatrix}\right], CEX_{\alpha}\ \left[\begin{smallmatrix}5\\0\end{smallmatrix}\right])$$

where a state is represented as $\left[\begin{smallmatrix}pc\\x\end{smallmatrix}\right]$. *From this counterexample, we use* `cefg` *to construct the counterexample flow-graph* Γ *given in Section 2. Each arc represents a possible transition within the counterexample tree. the procedure* REFINE *then walks all possible paths of the control-flow graph simultaneously, starting from the first node as follows:*

Iteration 1:	$N = \{n0\}$,	$S = \varnothing$
Iteration 2:	$N = \{n1\}$,	$S = \{n0, n1\}$
Iteration 3:	$N = \{n2, n3\}$,	$S = \{n0, n1, n2, n3\}$

After the first and second iterations PSYNTH$_D$ *does not discover a predicate to distinguish the two branches, but after the third call to* REFINE*, the predicate pairs* $(pc = \ell_2, pc = \ell_3)$ *and* $(pc = \ell_2, pc \neq \ell_3)$ *are discovered, which distinguish paths that remain in the loop or exit the loop. A new machine is then constructed with prophecy variables corresponding to these decisions, and for this new machine an ∀CTL verifier can prove that the property holds.*

Example 9. *Consider the following program for which we would like to prove* $\varphi = (FG\ y = 1) \vee (F\ x \geq t)$*:*

```
ℓ0:  x = y = 0; t = *;
     while(*)
ℓ1:     x++;
ℓ2:  t = *;
ℓ3:  if (x<t)
ℓ4:     y=1;
     while (true)
ℓ5:     skip;
```

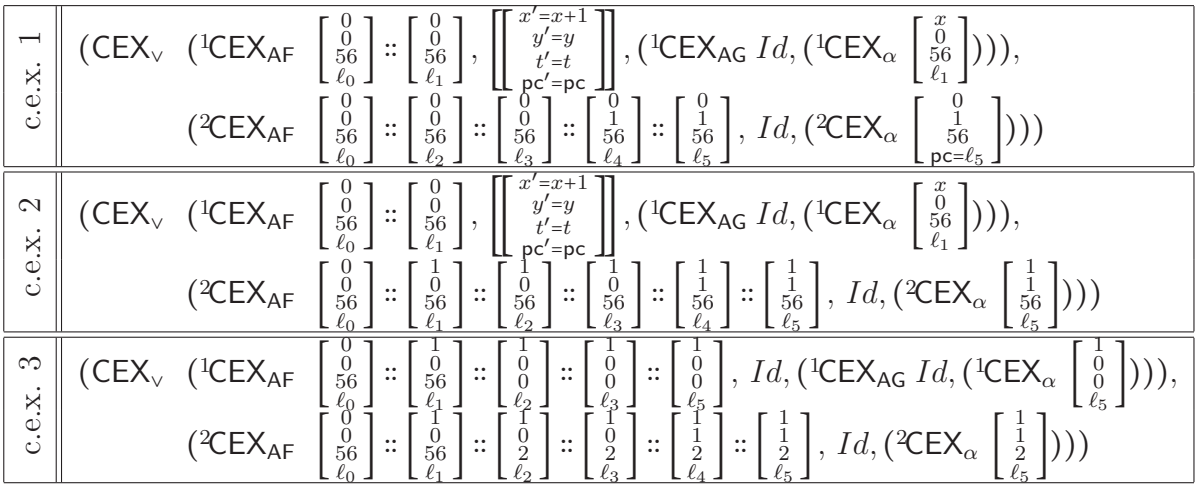


Figure 9: Counterexamples for each of the three iterations of proving Example 9. The notation Id indicates the identity transition (arising from the loop at line ℓ_5).

The machine representing this program can be encoded as follows:

$$
\begin{aligned}
S &= \begin{bmatrix}\mathbb{N}\\\mathbb{N}\\\mathbb{N}\\\mathcal{L}\end{bmatrix} \quad \textit{denoted} \quad \begin{bmatrix}x\\y\\t\\\mathsf{pc}\end{bmatrix} \qquad I = \begin{bmatrix}0\\0\\\mathbb{N}\\\ell_0\end{bmatrix}\\
R &= [\![(\mathsf{pc}=\ell_0 \wedge \mathsf{pc}'=\ell_1 \wedge x'=x \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_0 \wedge \mathsf{pc}'=\ell_2 \wedge x'=x \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_1 \wedge \mathsf{pc}'=\ell_1 \wedge x'=x+1 \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_1 \wedge \mathsf{pc}'=\ell_2 \wedge x'=x+1 \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_2 \wedge \mathsf{pc}'=\ell_3 \wedge x'=x \wedge y'=y \wedge t' \in \mathbb{N}) \vee\\
&\quad (\mathsf{pc}=\ell_3 \wedge \mathsf{pc}'=\ell_4 \wedge x<t \wedge x'=x \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_3 \wedge \mathsf{pc}'=\ell_5 \wedge x\geq t \wedge x'=x \wedge y'=y \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_4 \wedge \mathsf{pc}'=\ell_5 \wedge x'=x \wedge y'=1 \wedge t'=t) \vee\\
&\quad (\mathsf{pc}=\ell_5 \wedge \mathsf{pc}'=\ell_5 \wedge x'=x \wedge y'=y \wedge t'=t)]\!]^R
\end{aligned}
$$

Using an $\forall$CTL prover, we may obtain the first counterexample in Figure 9. From this counterexample, we use `cefg` *to construct the first counterexample flow graph in Figure 10. Each arc represents a possible transition within the counterexample tree. The procedure* Refine *then walks all possible paths of the control-flow graph simultaneously, starting from* `n0` *as follows:*

$$\textit{Iteration 1:} \quad N = \{\mathsf{n0}\}, \quad S = \varnothing$$

In this iteration, Refine *finds that $N' = \{\mathsf{n1},\mathsf{n2}\}$ and that $R = \{(\ell_0,\ell_1),(\ell_0,\ell_2)\}$. Taking the (only) pair of relations from R,* PSynth$_D$ *generates the predicate pairs $(\mathsf{pc}=\ell_0, \mathsf{pc}=\ell_1)$ and $(\mathsf{pc}=\ell_0, \mathsf{pc}\neq\ell_1)$. Corresponding prophecy variables are created, and the $\forall$CTL verifier is used on the newly constructed machine, resulting in the next counterexample in Figure 9. We then get the second counterexample flow graph in Figure 10 and the* Refine *explores it as follows:*

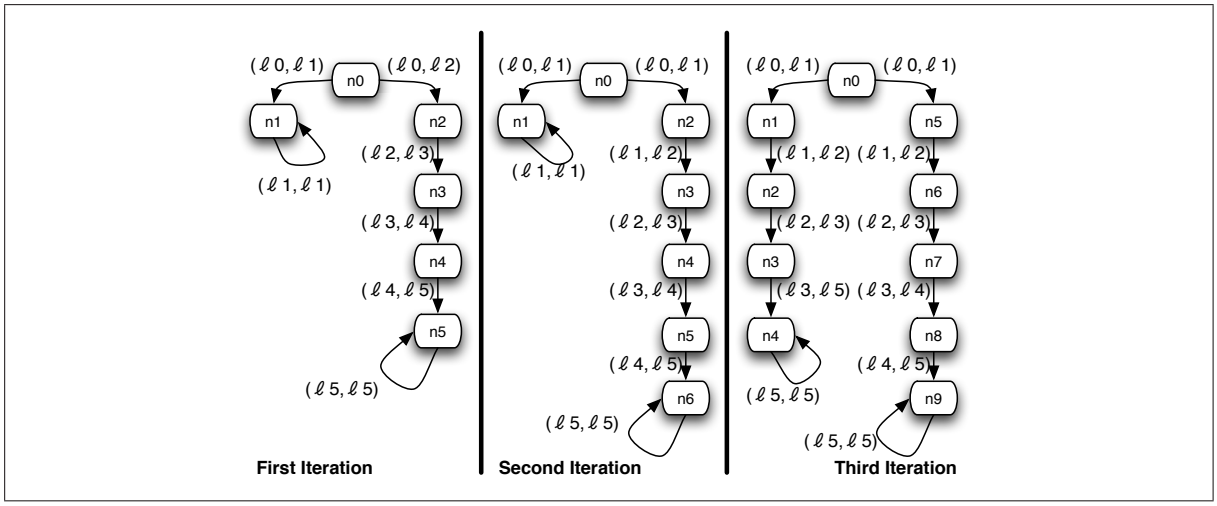


Figure 10: The counterexample flow graphs that are constructed at each iteration of proving Example 9.

$$\begin{array}{lll} \textit{Iteration 1:} & N = \{\mathit{n0}\}, & S = \varnothing \\ \textit{Iteration 2:} & N = \{\mathit{n1},\mathit{n2}\}, & S = \{\mathit{n0},\mathit{n1},\mathit{n2}\} \end{array}$$

After the first iteration, PSYNTH$_D$ *does not discover any predicates to distinguish the two branches, but after the second iteration the predicate pairs* $(\mathsf{pc} = \ell_1, \mathsf{pc} = \ell_1)$ *and* $(\mathsf{pc} = \ell_1, \mathsf{pc} \neq \ell_1)$ *are discovered, which distinguish paths that remain in the loop or exit the loop. The* $\forall$*CTL verifier is executed once again, resulting in the third counterexample in Figure 9. The counterexample flow-graph is given in Figure 10 and* REFINE *explores it as follows:*

$$\begin{array}{lll} \textit{Iteration 1:} & N = \{\mathit{n0}\}, & S = \varnothing \\ \textit{Iteration 2:} & N = \{\mathit{n1},\mathit{n5}\}, & S = \{\mathit{n0},\mathit{n1},\mathit{n5}\} \\ \textit{Iteration 3:} & N = \{\mathit{n2},\mathit{n6}\}, & S = \{\mathit{n0},\mathit{n1},\mathit{n5},\mathit{n2},\mathit{n6}\} \\ \textit{Iteration 4:} & N = \{\mathit{n3},\mathit{n7}\}, & S = \{\mathit{n0},\mathit{n1},\mathit{n5},\mathit{n2},\mathit{n6},\mathit{n3},\mathit{n7}\} \end{array}$$

In the final iteration, PSYNTH$_D$ *discovers the predicate pairs* $(\mathsf{pc} = \ell_2, \mathsf{t} \geq \mathsf{x})$ *and* $(\mathsf{pc} = \ell_2, \mathsf{t} < \mathsf{x})$*. Notice that the second predicate is over a program variable other than* pc *– in the next example we will see that* pc *is not always sufficient to distinguish paths. Running the* $\forall$*CTL verifier one more time yields no counterexamples. Hence the original LTL property holds.*

Example 10. *In the examples above, almost all predicates were over the program counter variable* pc*. In many cases, the program counter serves as a convenient way of distinguishing paths through the program. However, this is not always the case. Consider proving the property* $(\mathsf{G}\,\mathsf{x} = 0) \vee (\mathsf{F}\,\mathsf{x} = 20)$ *for the following program:*

```
ℓ0: x = 0;
    while(x<20)
ℓ1:     x := (x==0)*{0,1} + (x==1)*20;
    while(true)
ℓ2:     skip
```

The notation {0,1} *represents nondeterministic choice between 0 or 1. The LTL property holds because in traces where this nondeterministic choice is always 0, the property* $\mathsf{G}\,\mathsf{x} = 0$ *holds. For any trace in which the nondeterministic choice is 1, the property* $\mathsf{F}\,\mathsf{x} = 20$ *holds.*

We shall represent the state as $\left[\begin{smallmatrix}\mathsf{x}\\ \ell\end{smallmatrix}\right]$ *where* $\mathsf{x} \in \mathbb{N}$*. An* $\forall$*CTL prover will generate the following counterexample to* $(\mathsf{AG}\,\mathsf{x} = 0) \vee (\mathsf{AF}\,\mathsf{x} = 20)$*:*

$$\begin{array}{ll}(\mathsf{CEX}_\vee & (\mathsf{CEX}_{\mathsf{AG}} \left[\begin{smallmatrix}0\\ \ell_0\end{smallmatrix}\right] :: \left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}1\\ \ell_1\end{smallmatrix}\right], (\mathsf{CEX}_\wedge \left[\begin{smallmatrix}1\\ \ell_1\end{smallmatrix}\right])) \\ & (\mathsf{CEX}_{\mathsf{AF}} \left[\begin{smallmatrix}0\\ \ell_0\end{smallmatrix}\right] :: \left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right], Id, (\mathsf{CEX}_\wedge \left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right])))\end{array}$$

For this counterexample Refine *would explore the corresponding CEFG, and discover the decision predicate pairs* $(\mathsf{x} = 0, \mathsf{x} = 1)$ *and* $(\mathsf{x} = 0, \mathsf{x} \neq 1)$ *which distinguish the transition* $(\left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right], \left[\begin{smallmatrix}1\\ \ell_1\end{smallmatrix}\right])$ *from* $(\left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\ \ell_1\end{smallmatrix}\right])$*. Importantly, there is no predicate over the program counter variable alone which distinguishes these two transitions. We can now synthesize a prophecy variable corresponding to this decision predicate and an* $\forall$*CTL prover will discover a proof of the* $\forall$*CTL property, implying that the original LTL property holds.*

6 Implementation

In this section we discuss some details of our implementation of the algorithm in Figure 1, our implementation of an ∀CTL prover, and the results of our tool when applied to example programs.

Predicate synthesis. In Section 5, we have assumed the existence of a predicate synthesis mechanism PSynth_D that met the constraints given in Figure 6:

$$\text{PSynth}_D(R, R') = \begin{cases}\{(a, b), (a, \neg b)\} & \text{such that } R \subseteq [\![a \wedge b']\!]^{\mathsf{R}} \text{ and } R' \subseteq [\![a \wedge \neg b']\!]^{\mathsf{R}} \\ \varnothing & \text{if no such } a, b \text{ exist}\end{cases}$$

Depending on the configuration of the systems considered by the tool, PSynth_D will need to be implemented in different ways. Here we describe a particular method of synthesizing predicates for counterexamples drawn from the style of programs typically accepted by modern model checking tools for infinite-state programs.

As is true in many symbolic model checking tools for software, we will assume that counterexamples are sequences of commands drawn from a path in the program. We will assume that these commands are over a finite set of arithmetic variables, and that the conditional checks and assignment statements only use linear arithmetic. Given this context, an implementation can represent the relations passed to PSynth_D as conjunctions of inequalities using variables. For example, the command sequence

$$\begin{array}{ll}\ell_{41}: & \mathsf{x} := \mathsf{x} - 1; \\ \ell_{21}: & \texttt{assume}(\mathsf{x} > 0); \\ \ell_{10}: & \mathsf{y} := \mathsf{x};\end{array}$$

which might represent a piece of a counterexample can be represented as a relation from valuations on $(\mathsf{x}, \mathsf{y}, \mathsf{pc})$ to valuations on $(\mathsf{x}', \mathsf{y}', \mathsf{pc}')$ where

$\exists \mathsf{x}_1, \mathsf{x}_0, \mathsf{y}_1, \mathsf{y}_0.$

$$\bigwedge \left\{ \begin{array}{c} \mathsf{pc} = \ell_{41} \wedge \mathsf{pc}' = \ell_{10} \wedge \mathsf{x} = \mathsf{x}_0 \wedge \mathsf{x}' = \mathsf{x}_1 \wedge \mathsf{y} = \mathsf{y}_0 \wedge \mathsf{y}' = \mathsf{y}_1 \\ \mathsf{x}_1 = \mathsf{x}_0 - 1 \wedge \mathsf{x}_1 > 0 \wedge \mathsf{y}_1 = \mathsf{x}_1 \end{array} \right\}$$

We can reduce the search for predicates in this setting to the search for functions satisfying a set of constraints. In this instance we hope to find families of affine functions f and g such that the following conditions are true

1. $(\exists V'.R_1 \wedge \exists V'.R_2) \Rightarrow \bigwedge_{i \in \mathsf{dom}(f)} f_i(V) > 0$
2. $R_1 \Rightarrow \bigwedge_{i \in \mathsf{dom}(g)} g_i(V') > 0$
3. $R_2 \Rightarrow \neg(\bigwedge_{i \in \mathsf{dom}(g)} .g_i(V') > 0)$

The set of pre-states common to both relations R_1 and R_2 are given $S \equiv \exists V'.R_1 \wedge \exists V'.R_2$, *i.e.* we are existentially quantifying out the post-states by quantifying out the variables that are used to represent them. We then find an over-approximation of S that is expressible as the conjunction of inequalities using f. The second and third constraints force the function g—which is expressed only over the primed variables—to distinguish between two transitions.

As done elsewhere [34], we can apply Farkas' lemma [22] and an SMT solver (*e.g.* Z3 [2] or Yices [19]) to find linear functions f_i and g_i that satisfy the above constraints. Thus, to implement $\text{PSynth}_D(R_1, R_2)$ we find families f and g satisfying the above constraints. We then return the predicates a and b where

$$a \equiv \bigwedge_{i \in \mathsf{dom}(f)} f_i(V) > 0 \quad \text{and} \quad b \equiv \bigwedge_{i \in \mathsf{dom}(g)} g_i(V) > 0$$

A witness to $\exists V'.R_1 \wedge \exists V'.R_2$ can be computed using a quantifier elimination procedure, or alternatively, an additional application of Farkas' lemma. In practice, however, a good guess is simply to take the valuation of pc from both R_1 and R_2, *i.e.* $S \equiv \mathsf{pc} = \ell$, where $R_1 \Rightarrow \mathsf{pc} = \ell$ and $R_2 \Rightarrow \mathsf{pc} = \ell$.

Proving ∀CTL for infinite-state systems. We use ∀CTL verification tool for infinite-state programs, described elsewhere [16]. Our ∀CTL prover works by reducing the task of ∀CTL verification, via a program transformation, to an interprocedural program analysis problem. Thus, we can use known safety analysis tools [5, 11, 18, 25] combined with techniques for refining termination arguments [7, 8, 17, 21, 36] to obtain an ∀CTL verification tool whose power is limited only by the power of these underlying tools. The transformation uses recursion and nondeterminism in such a way that when these tools are applied to the transformed program, they effectively perform the necessary reasoning (*e.g* backtracking, eventuality checking, tree counterexamples, abstraction, abstraction-refinement, etc.) to prove branching-time behaviors of the original program. Formally, our transformation $\mathcal{T}$ works as follows: For a program P and an ∀CTL property Φ,

$$\exists \mathcal{M}.\, \mathcal{T}(P, \mathcal{M}, \Phi) \text{ cannot return } \mathsf{false} \;\Rightarrow\; P \vDash \Phi$$

where $\mathcal{M}$ is assumed to be a finite set of disjunctively well-founded relations [35]. The new program $\mathcal{T}(P, \mathcal{M}, \Phi)$ is constructed by recursively walking the structure of Φ. Instances of $\mathsf{AF}(p)$ is syntactically decomposed into proving termination to a set of states in which p holds; $\mathsf{AG}(p)$ can be decomposed into checking that p holds at each line of the program, etc. $\mathcal{M}$ can be thought of as the argument of progress when proving Φ. Once a suitable set $\mathcal{M}$ has been found, proving that $\mathcal{T}(P, \mathcal{M}, \Phi)$ cannot return false can be accomplished

with existing interprocedural analysis tools. Satisfying instances of $\mathcal{M}$ can be found using the same technique as is used in TERMINATOR [17]. In our implementation we use SLAM [5] as the underlying safety prover, and RANKFINDER [34] as the method of finding new ranking functions f from spurious counterexamples χ.

Experiments. We have drawn out a set of LTL challenge problems from industrial code bases. Examples were taken from code models of the I/O subsystem of the Windows kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. We also include a few toy examples, as well as the example from Figure 8 in [15]. Sources of these examples can be found at

`http://www.cl.cam.ac.uk/~ejk39/ltl/`

In many cases, heap-commands from the original sources have been abstracted away using the approach due to Magill *et al.* [29]. This abstraction introduces new arithmetic variables that track the sizes of recursive predicate found as a byproduct of a successful memory safety analysis using an abstract domain based on separation logic. This abstraction also may introduce extra nondeterminism into the transition relation which, in more complex cases, may force our method to synthesize decision predicates.

The only previously known tool for automatically proving LTL-like properties of infinite-state programs is described in [15], which is a TERMINATOR-like [17] procedure with an extension for fairness. LTL2BA [23] is used to convert LTL formulae to Büchi automata. As we have done in our implementation of Figure 1, the implementation of [15] uses SLAM as the underlying safety model checker, and RANKFINDER [34] as the rank function synthesis tool.

Table 1 reports the results of our experiments. The first column describes the code artifact. We added bugs into several of the examples. The second column "LOC" reports the number of lines of code for each example. We studied the results for properties of differing shapes (e.g. $\mathsf{G}(p \Rightarrow Fq)$, $\mathsf{FG}p$, $\mathsf{GF}p$, etc.). Experiments were run using Windows Vista and an Intel 2.66GHz processor.

For both tools we report the total time, the number of ranking functions required (denoted $|\mathbf{M}|$), and the result for each of the benchmarks. A ✓ indicates that the tool proved the property, and χ is used to denote cases where bugs were found. In the case that the tool exceeded the timeout threshold of 4 hours, "**T.O.**" is used to represent the time, the result is listed as "**???**", and we simply report the current size of $|\mathbf{M}|$ at the time that the tool was killed together with a "+" symbol.

For our approach we report the number of decision predicates required $|\Omega|$. For these examples relatively few prophecy variables are usually required. This confirms our assumption that faster CTL-based techniques *usually* work, so long as we have a fast method for evaluating the potential spuriousness of CTL counterexamples, and an effective strategy of refinement when CTL methods fail. We also observe that $|\mathbf{M}|$ is typically smaller when using the decision predicates based tool.

We implemented support for fairness in our decision predicate based approach tool in order to support Figure 8 of [15]. This is due to the fact that one of the fairness constraints actually comes from an environment assumption and thus must still be modeled. Our support for fairness uses essentially the same recipe as given in [15], combined with the source-to-source transformation.

Program	LOC	Property	Fair termination tool [15]			Decision predicates tool (Figure 1)			
			Time (s)	$\|\mathbf{M}\|$	**Result**	**Time** (s)	$\|\mathbf{M}\|$	$\|\Omega\|$	**Result**
Example from Section 2	5	$\mathsf{FG}p$	2.32	1	✓	1.98	1	1	✓
Example from Fig. 8 of [15]	34	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	209.64	1	✓	27.94	0	0	✓
Toy acquire/release example	14	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	103.48	3	✓	14.18	1	0	✓
Toy linear arith. 1	13	$p \Rightarrow \mathsf{F}q$	126.86	1	✓	34.51	1	0	✓
Toy linear arith. 2	13	$p \Rightarrow \mathsf{F}q$	**T.O.**	1+	**???**	6.74	1	0	✓
PostgreSQL strmsrv	259	$\mathsf{G}(p \Rightarrow \mathsf{FG}q)$	**T.O.**	5+	**???**	9.56	0	0	✓
PostgreSQL strmsrv+bug	259	$\mathsf{G}(p \Rightarrow \mathsf{FG}q)$	87.31	0	χ	47.16	1	0	χ
PostgreSQL pgarch	61	$\mathsf{FG}p$	31.50	2	✓	15.20	0	0	✓
PostgreSQL dropbuf	152	$\mathsf{G}p$	**T.O.**	2+	**???**	1.14	0	0	✓
PostgreSQL dropbuf	152	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	53.99	1	✓	27.54	2	0	✓
Apache `accept()` liveness	314	$\mathsf{G}p \Rightarrow \mathsf{GF}q$	**T.O.**	1+	**???**	197.41	1	2	✓
Apache progress	314	$\mathsf{G}(p \Rightarrow (\mathsf{F}q_1 \vee \mathsf{F}q_2))$	685.34	0	✓	684.24	0	0	✓
Windows OS fragment 1	180	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	901.81	2	✓	539.00	2	0	✓
Windows OS fragment 2	158	$\mathsf{FG}p$	16.47	0	✓	52.10	3	3	✓
Windows OS fragment 2+bug	158	$\mathsf{FG}p$	26.15	0	χ	30.37	0	0	χ
Windows OS fragment 3	14	$\mathsf{FG}p$	4.21	0	✓	15.75	1	1	✓
Windows OS fragment 4	327	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	**T.O.**	7+	**???**	1,114.18	1	0	✓
Windows OS fragment 4	327	$(\mathsf{F}a) \vee (\mathsf{F}b)$	1,223.96	5	✓	100.68	1	0	✓
Windows OS fragment 5	648	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	**T.O.**	1+	**???**	**T.O.**	0	0	**???**
Windows OS fragment 6	13	$\mathsf{FG}p$	149.41	2	✓	59.56	1	0	✓
Windows OS fragment 6+bug	13	$\mathsf{FG}p$	6.06	0	χ	22.12	0	0	χ
Windows OS fragment 7	13	$\mathsf{GF}p$	**T.O.**	1+	**???**	55.77	1	0	✓
Windows OS fragment 8	181	$\mathsf{FG}p$	**T.O.**	1+	**???**	5.24	1	0	✓

Table 1: Comparison of fair termination based LTL prover [15] to decision predicate based algorithm from Figure 1 . Examples drawn from PostgreSQL database server, Apache web server, as well as the I/O subsystem of the Windows OS. The property column indicates the shape of the temporal properties, where p and q are atomic propositions specific to the program. A ✓ indicates that the tool has proved the property, whereas a χ indicates that a valid LTL counterexample has been found. $|\Omega|$ indicates the number of decision predicates needed, and $|\mathbf{M}|$ the number of progress measures required. **T.O.** indicates that the experiment timed out after 4 hours, and in such cases we specify at least how many termination arguments were needed (denoted +).

As mentioned in Section 1, a limitation to our approach is that there are cases when we see a minor performance penalty for our strategy of only tracking correlations on demand (*e.g.* in "Windows OS fragment 2.") We also see some minor overhead when computing real counterexamples (*e.g.* in "Windows OS fragment 2+bug").

The most dramatic aspect of Table 1 is the overall result: our decision predicate based LTL prover was able to prove/disprove all but 1 example in usually a fraction of a minute, whereas the fair termination based tool fails on nearly a quarter of the benchmarks. This is due to our strategy of first trying to use ∀CTL proof strategies, and only tracking subtle relationships between families of traces on demand using decision predicates. Without our approach we could not reliably use an ∀CTL-based proof strategy with precision equal to native LTL-based approaches. In each of these **T.O.** cases but one our decision predicate based tool proves all of the examples with reasonable runtimes (resulting in a ✓). Furthermore, our tool reported no spurious counterexamples: in the cases where a purely ∀CTL-based approach would have been incomplete for LTL (resulting in a spurious counterexample), our refinement procedure quickly found and then symbolically shifted the problematic nondeterminism into the state-space of the system.

7 Conclusion

We have described a new algorithm for proving LTL properties of infinite-state systems. Our algorithm searches for instances of nondeterminism that preclude the use of CTL-based proof methods. We characterize these instances of nondeterminism using decision predicates, and then symbolically shift them into the state-space using a partial-determinization procedure. The advantage to this approach is that CTL proof methods can be used where they would have previously failed. We find in practice that most instances of nondeterminism is harmless to CTL proof methods. Thus, in many cases, we see performance improvements when using this strategy.

Acknowledgments. We thank Josh Berdine, Matko Botinčan, Axel Legay, Peter O'Hearn, Matthew Parkinson, Nir Piterman, Moshe Vardi and Hongseok Yang for their comments and thoughtful discussions. Stephen Magill provided several of the examples from Table 1. We also thank the Gates Cambridge Scholarship program for funding Eric Koskinen's Ph.D.

References

[1] Cadence SMV. `http://www.kenmcmil.com/smv.html`.

[2] The Z3 Theorem Prover. research.microsoft.com/projects/Z3.

[3] Abadi, M., and Lamport, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (1991), 253–284.

[4] Abdulla, P. A., Jonsson, B., Nilsson, M., d'Orso, J., and Saksena, M. Regular model checking for LTL(MSO). In *CAV* (2004).

[5] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S., and Ustuner, A. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review 40*, 4 (2006), 85.

[6] Bouajjani, A., Legay, A., and Wolper, P. Handling liveness properties in (ω-) regular model checking. *Electronic Notes in Theoretical Computer Science 138*, 3 (2005), 101–115.

[7] Bradley, A., Manna, Z., and Sipma, H. Termination of polynomial programs. In *VMCAI* (2005).

[8] Bradley, A. R., Manna, Z., and Sipma, H. B. Linear ranking with reachability. In *CAV* (2005).

[9] Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. Symbolic model checking: 10 to the 20 states and beyond. *Information and Computation 98*, 2 (1992).

[10] Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS 8*, 2 (1986), 263.

[11] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *JACM 50*, 5 (2003), 794.

[12] Clarke, E., Grumberg, O., and Peled, D. *Model checking*. Springer, 1999.

[13] Clarke, E., Jha, S., Lu, Y., and Veith, H. Tree-like counterexamples in model checking. In *LICS* (2002).

[14] Clarke, E. M., Grumberg, O., and Hamaguchi, K. Another look at LTL model checking. *Form. Methods Syst. Des. 10*, 1 (1997), 47–71.

[15] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y. Proving that programs eventually do something good. In *POPL* (2007).

[16] Cook, B., Koskinen, E., and Vardi, M. Branching-time reasoning for programs. Tech. Rep. UCAM-CL-TR-788, University of Cambridge, Computer Laboratory, Jan. 2011.

[17] Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI* (2006).

[18] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTREE analyzer. In *ESOP* (2005).

[19] Dutertre, B., and de Moura, L. M. A fast linear-arithmetic solver for dpll(t). In *CAV* (2006), T. Ball and R. B. Jones, Eds., vol. 4144 of *LNCS*, Springer, pp. 81–94.

[20] Esparza, J., Kucera, A., and Schwoon, S. Model-checking LTL with regular valuations for pushdown systems. In *TACS* (2001).

[21] Fang, Y., Piterman, N., Pnueli, A., and Zuck, L. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer (STTT) 8*, 3 (2006), 261–279.

[22] Farkas, J. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik 124* (1902), 1–27.

[23] Gastin, P., and Oddoux, D. Fast LTL to Büchi automata translation. In *CAV* (July 2001).

[24] Havelund, K., and Pressburger, T. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT) 2*, 4 (2000), 366–381.

[25] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV* (2002).

[26] Hobor, A., Appel, A. W., and Nardelli, F. Z. Oracle semantics for concurrent separation logic. In *ESOP* (2008).

[27] Holzmann, G. J. The model checker SPIN. *IEEE Trans. Software Eng. 23*, 5 (1997), 279–295.

[28] Kwiatkowska, M., Norman, G., and Parker, D. PRISM: Probabilistic symbolic model checker. *LNCS 2324* (2002), 200–204.

[29] Magill, S., Berdine, J., Clarke, E., and Cook, B. Arithmetic strengthening for shape analysis. *LNCS 4634* (2007), 419.

[30] Maidl, M. The common fragment of CTL and LTL. In *FOCS* (2000).

[31] Nain, S., and Vardi, M. Branching vs. linear time: Semantical perspective. In *ATVA* (2007).

[32] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science* (1977), IEEE, pp. 46–57.

[33] Pnueli, A., and Zaks, A. PSL model checking and run-time verification via testers. In *FM* (2006), J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085 of *LNCS*, Springer, pp. 573–586.

[34] Podelski, A., and Rybalchenko, A. A Complete Method for the Synthesis of Linear Ranking Functions. *LNCS* (2003), 239–251.

[35] Podelski, A., and Rybalchenko, A. Transition invariants. In *LICS* (2004), pp. 32–41.

[36] Podelski, A., and Rybalchenko, A. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL* (2007).

[37] Qadeer, S., Sezgin, A., and Tasiran, S. Back and forth: Prophecy variables for static verification of concurrent programs. Tech. Rep. MSR-TR-2009-142, Microsoft, 2009.

[38] Safra, S. On the complexity of omega -automata. In *SFCS* (1988).

[39] Sankaranarayanan, S., Sipma, H., and Manna, Z. Constraint-based linear-relations analysis. In *SAS* (2004).

[40] Schneider, K. Model checking on product structures. *FMCAD* (1998).

[41] Schuppan, V., and Biere, A. Liveness checking as safety checking for infinite state spaces. In *Workshop on Verification of Infinite-State Systems (INFINITY)* (2005).

[42] Vardhan, A., Sen, K., Viswanathan, M., and Agha, G. Using language inference to verify Omega-regular properties. In *TACAS* (2005).

[43] Vardi, M. Branching time vs. linear time: Final showdown. In *TACAS* (2001).

[44] Vardi, M. Y., and Wolper, P. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS* (1986).