Reasoning about Nondeterminism in Programs

Byron Cook

Microsoft Research Cambridge
& University College London

Eric Koskinen [*]

New York University

Abstract

Branching-time temporal logics (*e.g.* CTL, CTL*, modal μ-calculus) allow us to ask sophisticated questions about the nondeterminism that appears in systems. Applications of this type of reasoning include planning, games, security analysis, disproving, precondition synthesis, environment synthesis, etc. Unfortunately, existing automatic branching-time verification tools have limitations that have traditionally restricted their applicability (*e.g.* push-down systems only, universal path quantifiers only, etc).

In this paper we introduce an automation strategy that lifts many of these previous restrictions. Our method works reliably for properties with non-trivial mixtures of universal and existential modal operators. Furthermore, our approach is designed to support (possibly infinite-state) programs.

The basis of our approach is the observation that existential reasoning can be reduced to universal reasoning if the system's state-space is appropriately restricted. This restriction on the state-space must meet a constraint derived from recent work on proving non-termination. The observation leads to a new route for implementation based on existing tools. To demonstrate the practical viability of our approach, we report on the results applying our preliminary implementation to a set of benchmarks drawn from the Windows operating system, the PostgreSQL database server, SoftUpdates patching system, as well as other hand-crafted examples.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; Correctness proofs; Reliability; D.4.5 [*Operating Systems*]: Reliability—Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Verification, Theory, Reliability

Keywords CTL, branching-time temporal logic, formal verification, termination, program analysis, model checking

1. Introduction

Branching-time temporal logics facilitate reasoning about the nondeterminism that exists within the transition relations of systems. For example we might wish to prove that a program P could (but is not forced to) terminate from every reachable state, and furthermore if and when termination does occur, p will hold. In the branching-time logic CTL, we can express this as[1]

$$P \vDash \mathsf{A}[\mathsf{EF}p \ \mathsf{W} \ p]$$

[1] We abuse notation slightly; we mean that $s \vDash \mathsf{A}[\mathsf{EF}p \ \mathsf{W} \ p]$ for every initial state s in program P.

Here we are using two temporal operators:

- $\mathsf{A}[a \ \mathsf{W} \ b]$ specifies that a and b are temporally sequenced in all executions through the system: either a might happen forever, or b must happen if a ever ceases to hold. $\mathsf{A}[a \ \mathsf{W} \ b]$ is a *universal* temporal operator: it must hold over *all* executions of the system, no matter which nondeterministic choices are made.

- $\mathsf{EF}a$ specifies that there exist nondeterministic choices that can be made such that a will eventually hold.

Applications of this type of reasoning include planning, games, security analysis, disproving, precondition synthesis, environment synthesis, and many others. In the area of planning, for example, nondeterministic choices in a system often represent choices that can be manipulated by a plan. With a proof that $P \vDash \mathsf{A}[\mathsf{EF}p \ \mathsf{W} \ p]$, we could devise a plan that would cause the system P to terminate in state p whenever desired [32]. As another example, consider environment synthesis: Imagine that we would like to find a condition that—if maintained—would guarantee whenever p holds, q will eventually hold: $\mathsf{AG}(p \Rightarrow \mathsf{AF} \ q)$. Towards this goal we can prove $\mathsf{EG}(p \Rightarrow \mathsf{AF} \ q)$, which states that it's *possible* but not guaranteed that whenever p then eventually q. With a proof in hand we can then work out the conditions required to guarantee such a desired condition, thus giving us a restriction on the state-space such that $\mathsf{AG}(p \Rightarrow \mathsf{AF} \ q)$ would be true.

At first glance the above applications of CTL reasoning look appealing. Unfortunately, there is a problem: The question of how to reliably automate proofs in CTL and similar branching-time logics (*e.g.* CTL*, modal μ-calculus) for infinite-state programs remains an open problem.

In this paper we report on the first known robust automatic proof method to support both universal and existential branching-time modal operators for programs. Our approach is based on the observation that existential reasoning can be reduced to universal reasoning when an appropriate restriction is placed on the the state-space of the system. For soundness we require that the restriction used meets a condition adapted from recent approaches to non-termination proving [22]. For example, to prove that there *exists* a path in a program P such that p holds (*i.e.* $P \vDash \mathsf{EF}p$), we search for a restriction $\mathcal{C}$ on P's state-space such that in the program restricted to $\mathcal{C}$-states, *for all* paths p eventually holds (*i.e.* $P|_{\mathcal{C}} \vDash \mathsf{AF}p$). To ensure that there still exists at least one path to p in $P|_{\mathcal{C}}$ we must check a condition that the restriction $\mathcal{C}$ be a *recurrent* set [22].

The advantage of our approach is that there are known solutions to the problem of universal branching-time, as well proving non-termination. Thus, to construct a full-fledged CTL prover for infinite-state programs, we need only develop a method that synthesizes appropriate restrictions $\mathcal{C}$. Towards this goal we give a procedure that refines candidate restrictions on demand using failed proof attempts with insufficient restrictions.

To demonstrate the practical viability of our approach we have built a preliminary unoptimized prover and applied it to toy bench-

marks with a variety of combinations of temporal operators, as well as examples drawn the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

Limitations. The formal treatment given in this paper is defined over general transitions systems. Thus, we make no assumption about the structure of programs (recursive, higher-order, etc). In practice, however, our implementation uses constraint-based methods as well as safety and termination proving techniques that limit the applicability of the tool to non-recursive programs with variables that range over arithmetic domains. Heap-based programs can be handled by first constructing a numerical abstraction (*e.g.* [26], [29]).

For the purpose of brevity we have limited ourselves here to the logic CTL, as CTL is a simple logic that supports mixtures of existential and universal reasoning. Applying these techniques to more expressive logics (*e.g.* CTL* or modal μ-calculus) is left as future work.

While the CTL proof rules we develop are sound and complete, our practical implementation is not complete for several reasons:

- The underlying technique for proving the universal subset of CTL is not complete.
- Our heuristics for synthesizing restrictions on the state-space is not complete.
- The refinement procedure for state-space restrictions may make incorrect choices early during the iterative proof search that limit the choices available later in the search. In these circumstances we a form of backtracking could potentially be used to consider alternative decisions.

2. Example

In this section we informally discuss a simple example. In the following sections we develop the approach more rigorously.

```
1 x = 0;
2 while (true) {
3    ρ¹ = *;
4    y = ρ¹;
5    x = 1;
6    ρ² = *;
7    n = ρ²;
8    while (n>0) {
9       n = n - y;
10   }
11   x = 0;
12 }
```

Here we have made all of the nondeterminism explicit through the use of variables ρ^1 and ρ^2. The symbol $*$ represents nondeterministic choice. Imagine that we would like to prove that it is *possible* to maintain the invariant that, whenever $x = 1$, then eventually $x = 0$. In CTL we would express this property as:

$$\mathsf{EG}(x = 1 \Rightarrow \mathsf{AF}(x = 0))$$

Here $\mathsf{EG}p$ specifies that there exist nondeterministic choices such that p holds forever. $\mathsf{AF}p$ says that in all nondeterministic choices p eventually becomes true.

Our method is based on the search for a restriction on the nondeterministic choices made (*i.e.* the ρ-variables) such that we can use tools that support only universal reasoning to do the hard work. In this case, if we restrict the nondeterministic choice $\rho^1 = *$ at line 3 to be positive then the *universal* variation of original property, $\mathsf{AG}(x = 1 \Rightarrow \mathsf{AF}(x = 0))$, holds of a similar program:

```
1 x = 0;
2 while (true) {
3    ρ¹ = *; assume(ρ¹ ≥ 1);
4    y = ρ¹;
5    x = 1;
6    ρ² = *;
7    n = ρ²;
8    while (n>0) {
9       n = n - y;
10   }
11   x = 0;
12 }
```

Here we have used a restriction on the state space:

$$\mathcal{C} \;\equiv\; (\text{program is at line 3} \;\Rightarrow\; \rho^1 \geq 1)$$

which we have implemented using an `assume` instruction [31].

$$\rho^1 = *; \underline{\text{assume}(\rho^1 \geq 1);}$$

We have to be careful when choosing restrictions, as we have done above. For example, the restriction

$$\mathcal{C} \;\equiv\; (\text{program is at line 3} \Rightarrow \mathsf{false})$$

would cause the universal reasoning to succeed. However, the `assume(false)` command that would appear on Line 3 would cause there to be no executions of this restricted program. Consequently, EG does not hold. In this paper we will describe a requirement on our choice of chute so that we guarantee non-emptiness.

Finding restrictions. To automate the discovery of restrictions we begin with $\mathcal{C} \equiv \mathsf{true}$. We then use failed proof attempts in the universal subset of CTL to guide where further restrictions need to be added. In our example we would try proving $\mathsf{AG}(x = 1 \Rightarrow \mathsf{AF}(x = 0))$ using $\mathcal{C} \equiv \mathsf{true}$, which fails. As a result we will get the following counterexample:

A path to $(x = 1)$:	`x = 0;`
	`ρ¹ = *;`
	`y = ρ¹;`
	`x = 1;`
	`ρ² = *;`
	`n = ρ²;`

A cyclic path where $(x = 0)$ *never occurs*:	`assume(n>0);`
	`n = n - y;`

Because the cyclic path is executed forever we can infer that $y \leq 0$ is invariant in the cyclic path. That is, we can strengthen the cyclic path to:

```
assume(y<=0);
assume(n>0);
n = n - y;
```

As is standard in tools based on counterexample-guided abstraction refinement (*e.g.* [30], [2]) we can represent this command sequence as a logical formula expressed using static single assignment variables:

$$\bigwedge \left\{ \begin{array}{c} x_1 = 0 \\ y_1 = \rho_1^1 \\ x_2 = 1 \\ n_1 = \rho_1^2 \\ y_1 \leq 0 \\ n_1 > 0 \\ n_2 = n_1 - y_1 \end{array} \right\}$$

We would like to compute a condition using one of the ρ variables that would make this path spurious. Take ρ_1^1 as our candidate. We

can compute a condition by performing quantifier elimination on the variables $(x_1, y_1, x_2, n_1, n_2, \rho_1^2)$ that are not in scope just after the command $\rho^1 := *$. The elimination is thus:

$$\exists y_1, x_1, x_2, n_1, n_2, \rho_1^2, \rho_1^0. \bigwedge \left\{ \begin{array}{l} x_1 = 0 \\ y_1 = \rho_1^1 \\ x_2 = 1 \\ n_1 = \rho_1^2 \\ y_1 \leq 0 \\ n_1 > 0 \\ n_2 = n_1 - y_1 \end{array} \right\} \equiv \rho_1^1 \leq 0$$

Finally, we can restrict the original program such that this path is not possible. To this end, we instrument the negation of this condition $\neg(\rho^1 \leq 0)$, after the assignment $\rho^1 := *$. Hence, we strengthen $\mathcal{C}$ with

$$\mathcal{C} := \mathcal{C} \wedge (\text{ program is at line } 3 \Rightarrow \rho^1 > 0)$$

In more complex cases with nested temporal existential operators we must find families of restrictions, maintaining a set of $\mathcal{C}$ restrictions, indexed by their use in a CTL proof search. In the next section we describe our procedure more formally and discuss this complexity in some detail.

3. Treating the Existential Fragment of CTL

We now turn to a formal description of our technique. We begin with new proof rules for CTL. These generalize our previous rules for $\forall$CTL [10, 11]. Our rules are given in terms of sets of states, allowing us to partition the state space rather than enumerate the state space.

In this paper we introduce a new proof rule for existential operators that characterizes the existence of traces as a subset of the transition relation called a *chute*. This allows us to apply *universal* reasoning to the subset, in order to prove *existential* properties. We characterize the side condition of non-emptiness with *recurrent sets*. In later sections, we will describe how proof derivations (including discovery of chutes) can be obtained automatically.

3.1 Preliminaries

Transition systems. A transition system $M = (S, R, I)$ is a set of states S, a transition relation $R \subseteq S \times S$, and a set of initial states $I \subseteq S$. We use the notation $r|_1$ to mean the first projection of relation r. A *trace* π of a transition system is an infinite sequence of states $(s_0, s_1, ...)$ such that $s_0 \in I$, $\forall i \geq 0$. $s_i \in S$ and $\forall i \geq 0$. $(s_i, s_{i+1}) \in R$. We denote by $\Pi(S, R, I)$ the set of all such traces. For convenience, we do not allow finite traces. The transition relation must be such that every state has at least one successor state: $\forall s \in S. \exists s'. (s, s') \in R$. This is without a loss of generality, as final states can be encoded as states that loop back to themselves. 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. $\pi_0^i = (\pi^i)_0$. We will use similar notation for a *finite path* ϖ. Additionally, we will say that ϖ^{end} denotes the final element of ϖ.

Ranking functions. For a state space S, a ranking function f is a total map from S to a well-ordered set with ordering relation $\prec$. A relation $R \subseteq S \times S$ is *well-founded* if and only if there exists a ranking function f such that $\forall (s, s') \in R$. $f(s') \prec f(s)$. We denote a finite set of ranking functions (or *measures*) as $\mathcal{M}$. Note that the existence of a finite set of ranking functions for a relation R is equivalent to containment of R^+ within a finite union of well-founded relations [34]. That is to say that a set of ranking functions $\{f_1, ..., f_n\}$ can denote the disjunctively well-founded relation $\{(s, s') \mid f_1(s') \prec f_1(s) \vee ... \vee f_n(s') \prec f_n(s)\}$.

Temporal logic. The syntax of a CTL formula is

$$\Phi ::= p \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \mathsf{AF}\Phi \mid \mathsf{EF}\Phi \mid \mathsf{A}[\Phi\mathsf{W}\Phi] \mid \mathsf{E}[\Phi\mathsf{W}\Phi]$$

Note that $\mathsf{AG}p$ can be defined as $\mathsf{A}[p \mathsf{\ W\ false}]$, and $\mathsf{EG}p$ can be defined as $\mathsf{E}[p \mathsf{\ W\ false}]$. The standard semantics of CTL is given in Figure 1. p is an atomic proposition. There are two classes of temporal constructors: constructors that quantify universally over paths $(\mathsf{AF}, \mathsf{AW})$ and constructors that quantify existentially over paths $(\mathsf{EF}, \mathsf{EW})$. The AF and EF constructors specify that a state in which Φ holds must be reached. The $[\Phi_1 \mathsf{W} \Phi_2]$ operator specifies that Φ_1 holds in every state where Φ_2 does not yet hold.

We use F and W as our base temporal operators (as opposed to the more standard U and R), as each corresponds to a distinct form of proof: F to termination, and W to safety. We omit the next state operator X as it is not particularly useful in an imperative programming language: the next step simply is a transition across a command and can be supported via F. We assume that formulae are written in negation normal form, in which negation only occurs next to atomic propositions, and assume that the domain of atomic propositions is closed under negation. A formula that is not in negation normal form can be easily normalized.

Definition 3.1 (CTL machine entailment). *For every* $M = (S, R, I)$ *and CTL property* Φ, $M \vDash \Phi \equiv \forall s \in I. R, s \vDash \Phi$.

Our rules are composed structurally over a CTL formula. In order to track components of a proof we need to uniquely identify subformulae. To this end, our definition of subformulae maintains a context path: $\kappa \equiv \epsilon \mid \mathsf{L}\,\kappa \mid \mathsf{R}\,\kappa$ that indicates the path from the root ϵ (the outermost property Φ), to the particular subproperty of interest, at each step taking either the left or right subformula ($\mathsf{L}\kappa$ or $\mathsf{R}\kappa$). For a CTL property Φ, the set of subformulae is a set of (κ, Φ) pairs as follows:

$$\mathsf{sub}(\Phi) \equiv \mathsf{sub}(\epsilon, \Phi)$$
$$\mathsf{sub}(\kappa, p) \equiv \{(\kappa, p)\}$$
$$\mathsf{sub}(\kappa, \Phi \vee \Phi') \equiv \{(\kappa, \Phi \vee \Phi')\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi')$$
$$\mathsf{sub}(\kappa, \Phi \wedge \Phi') \equiv \{(\kappa, \Phi \wedge \Phi')\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi')$$
$$\mathsf{sub}(\kappa, \mathsf{AF}\Phi) \equiv \{(\kappa, \mathsf{AF}\Phi)\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi)$$
$$\mathsf{sub}(\kappa, \mathsf{EF}\Phi) \equiv \{(\kappa, \mathsf{EF}\Phi)\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi)$$
$$\mathsf{sub}(\kappa, \mathsf{A}[\Phi\mathsf{W}\Phi']) \equiv \{(\kappa, \mathsf{A}[\Phi\mathsf{W}\Phi'])\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi')$$
$$\mathsf{sub}(\kappa, \mathsf{E}[\Phi\mathsf{W}\Phi']) \equiv \{(\kappa, \mathsf{E}[\Phi\mathsf{W}\Phi'])\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi')$$

3.2 A Proof System for CTL

Our proof system given in Figure 2 is a relation between a set of states X and a CTL context/formula κ, Φ. Atomic proposition $\mathrm{R_{AP}}$ and conjunction $\mathrm{R_{AND}}$ are as expected (note that $[\![p]\!]$ means the set of states such that p holds). Disjunction $\mathrm{R_{OR}}$ partitions the set of states into two sets X_1 and X_2 where Φ_1 holds of X_1 and Φ_2 holds of X_2.

Quantification operators. In this paper, we generalize the proof system to encompass the universal and existential temporal operators. To this end, we decompose the temporal operators based on quantification, so we have:

$$\gamma ::= \mathsf{F}\Phi \mid [\Phi \mathsf{\ W\ } \Phi]$$
$$\Phi ::= p \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \mathsf{A}\gamma \mid \mathsf{E}\gamma$$

This decomposition allows us to treat universal and existential temporal operators similarly. When an $\mathsf{A}\gamma$ or $\mathsf{E}\gamma$ quantification operator is reached, a corresponding rule ($\mathrm{R_A}$ or $\mathrm{R_E}$) is used to identify three key sets of states that will be relevant to γ:

1. The initial states X
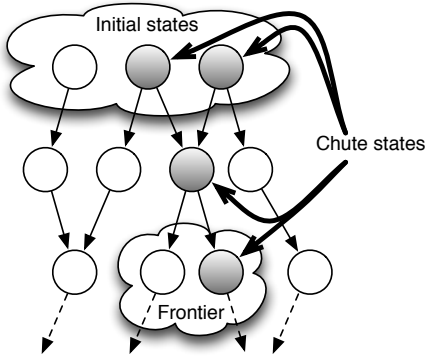2. The *chute* states $\mathcal{C}$
3. The *frontier* [10, 11] states $\mathcal{F}$

$$
\begin{aligned}
R, s \vDash p &\iff s \in \llbracket p \rrbracket \\
R, s \vDash \Phi_1 \wedge \Phi_2 &\iff R, s \vDash \Phi_1 \ \text{ and } \ R, s \vDash \Phi_2 \\
R, s \vDash \Phi_1 \vee \Phi_2 &\iff R, s \vDash \Phi_1 \ \text{ or } \ R, s \vDash \Phi_2 \\
R, s \vDash \mathsf{AF}\Phi &\iff \forall (s_0, s_1, \ldots) \in \Pi(S, R, \{s\}).\ \exists i \geq 0.\ R, s_i \vDash \Phi \\
R, s \vDash \mathsf{EF}\Phi &\iff \exists (s_0, s_1, \ldots) \in \Pi(S, R, \{s\}).\ \exists i \geq 0.\ R, s_i \vDash \Phi \\
R, s \vDash \mathsf{A}[\Phi_1 \ \mathsf{W} \ \Phi_2] &\iff \forall (s_0, s_1, \ldots) \in \Pi(S, R, \{s\}).\ (\forall i \geq 0.\ R, s_i \vDash \Phi_1)\ \vee\ (\exists j \geq 0.\ R, s_j \vDash \Phi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \Phi_1) \\
R, s \vDash \mathsf{E}[\Phi_1 \ \mathsf{W} \ \Phi_2] &\iff \exists (s_0, s_1, \ldots) \in \Pi(S, R, \{s\}).\ (\forall i \geq 0.\ R, s_i \vDash \Phi_1)\ \vee\ (\exists j \geq 0.\ R, s_j \vDash \Phi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \Phi_1)
\end{aligned}
$$

Figure 1: Standard CTL semantics, a relation from a state to a formula.

$$
\frac{X \subseteq \llbracket p \rrbracket}{X \vdash \kappa, p} \ \textsc{Rap} \qquad \frac{X \vdash \mathsf{L}\kappa, \Phi_1 \quad X \vdash \mathsf{R}\kappa, \Phi_2}{X \vdash \kappa, \Phi_1 \wedge \Phi_2} \ \textsc{Rand}
$$

$$
\frac{X = X_1 \cup X_2 \quad X_1 \vdash \mathsf{L}\kappa, \Phi_1 \quad X_2 \vdash \mathsf{R}\kappa, \Phi_2}{X \vdash \kappa, \Phi_1 \vee \Phi_2} \ \textsc{Ror}
$$

$$
\frac{X, S, \mathcal{F}^\kappa \Vdash \kappa, \gamma}{X \vdash \kappa, \mathsf{A}\gamma} \ \textsc{Ra} \qquad \frac{(X, \mathcal{C}^\kappa, \mathcal{F}^\kappa) \text{ is rcr} \quad X, \mathcal{C}^\kappa, \mathcal{F}^\kappa \Vdash \kappa, \gamma}{X \vdash \kappa, \mathsf{E}\gamma} \ \textsc{Re}
$$

$$
\frac{\mathcal{R}^{\mathcal{F}^\kappa}_{\underset{X}{\mathcal{C}^\kappa}} \text{ is w.f.} \quad \mathcal{F}^\kappa \vdash \mathsf{L}\kappa, \Phi}{X, \mathcal{C}^\kappa, \mathcal{F}^\kappa \Vdash \kappa, \mathsf{F}\Phi} \ \textsc{Rf} \qquad \frac{\mathcal{R}^{\mathcal{F}^\kappa}_{\underset{X}{\mathcal{C}^\kappa}} |_1 \vdash \mathsf{L}\kappa, \Phi_1 \quad \mathcal{F}^\kappa \vdash \mathsf{R}\kappa, \Phi_2}{X, \mathcal{C}^\kappa, \mathcal{F}^\kappa \Vdash \kappa, [\Phi_1 \ \mathsf{W} \ \Phi_2]} \ \textsc{Rw}
$$

$$
\frac{(s, t) \in R \quad s \in X \quad s \notin \mathcal{F} \quad s, t \in \mathcal{C}}{\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}(s, t)}
$$

$$
\frac{\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}(s, t) \quad (t, u) \in R \quad t \notin \mathcal{F} \quad u \in \mathcal{C}}{\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}(t, u)}
$$

Figure 2: On the left is a proof system for CTL that unifies the temporal treatment of universal A and existential E. There is a side condition on the existential rule that the relevant states form a recurrent set. The definition of $\mathcal{R}$ is given on the right.

We can think about these as: the start states X, the relevant region of the transition relation $\mathcal{C}$ through which execution is allowed to pass, and a stopping point at the frontier $\mathcal{F}$. Visually:

The chute $\mathcal{C}$ effectively carves out the portion of the state space that will be relevant to the subproperty. This generalization with chutes provides a means of treating each quantification fragment of CTL:

1. For a given subformula, *existential* properties can be treated by restricting the state space to a chute and applying universal reasoning on that chute (Re). The non-emptiness side condition is that this chute must be recurrent, discussed below.
2. For a given subformula, *universal* properties can be treated by letting the chute be the entire state space (Ra).

The quantification proof rules (Figure 2) each involve a proof obligation for the temporal operator γ, in terms of this triple:

$X, \mathcal{C}, \mathcal{F} \Vdash \kappa, \gamma$. In the case of Ra, we use the entire state space S as the chute. In the case of Re, we must identify a chute $\mathcal{C}$ and ensure a special side condition that the triple $(X, \mathcal{C}, \mathcal{F})$ be a recurrent set, defined as follows:

Definition 3.2 (Recurrent set). *For sets of states $X, \mathcal{C}, \mathcal{F}$ and transition relation R, we say that $\mathcal{C}$ is a* recurrent set *w.r.t. X and $\mathcal{F}$ (denoted $(X, \mathcal{C}, \mathcal{F})$ is rcr) provided that $X \cap \mathcal{C} \neq \varnothing$ and:*

1. *$X \cap \mathcal{C} \subseteq \mathcal{F}$, or*
2. *For every $x \in \mathcal{C}$, there exists x' such that $(x, x') \in R$ and $x' \in \mathcal{F} \vee x' \in \mathcal{C}$.*

Intuitively, a recurrent set $\mathcal{C}$ is such that either: (i) there is a path from X that either immediately intersects with $\mathcal{F}$, (ii) there is a path that reaches $\mathcal{F}$ by following $\mathcal{C}$ at each transition or (iii) there is a path that remains in $\mathcal{C}$ forever, never reaching $\mathcal{F}$. Note that this is a generalization of a previously described notion of a recurrent set [22]. For proving non-termination, the simpler notion of recurrent set is sufficient because there is no frontier. With nested temporal operators, we must account for a set being recurrent modulo a frontier, wherein the subproperty is satisfied.

Temporal operators. The quantification operators discussed above are designed to be compatible with either temporal operator. First we identify the region (*i.e.* subset) of the transition relation that is relevant to a given triple $X, \mathcal{C}, \mathcal{F}$ via the relation $\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}$, given on the right-hand side of Figure 2. Notice that $\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}} \subseteq R$. In the base case, a transition (s, t) is in $\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}$ if $s \in X, s \in \mathcal{C}$, and $s \notin \mathcal{F}$. In the inductive case, a transition (t, u) is included in $\mathcal{R}^{\mathcal{F}}_{\underset{X}{\mathcal{C}}}$ whenever

some $(s,t) \in \mathcal{R}_{\substack{\mathcal{F} \\ \mathcal{C} \\ X}}$, $t \notin \mathcal{F}$ and $u \in \mathcal{C}$. Intuitively, these are all of the double-line edges in the following diagram:
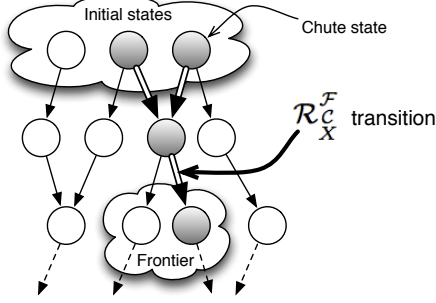
The proof that $X, \mathcal{C}, \mathcal{F} \Vdash \mathsf{F}\Phi$ involves showing that $\mathcal{R}_{\substack{\mathcal{F} \\ \mathcal{C} \\ X}}$ is well-founded. This well-foundedness condition ensures that all traces through $X, \mathcal{C}, \mathcal{F}$ reach the frontier $\mathcal{F}$ after finitely many steps. Moreover, the RF rule requires that $\mathcal{F} \vdash \Phi$. The proof rule RW requires that along every path from X through the chute $\mathcal{C}$, Φ_1 holds (by requiring that the first projection of $\mathcal{R}_{\substack{\mathcal{F} \\ \mathcal{C}}}$ satisfies Φ_1) unless the frontier $\mathcal{F}$ has been reached at which point Φ_2 holds.

Remark: CTL*. Note that our decomposition of quantification and temporal operators bares some resemblance to CTL* [16]. CTL* is a generalization of CTL (and of LTL) that includes state propositions (E, A) as well as path propositions (F, W). The difference is that a path proposition F or W is already fixed on a particular path. By contrast, our treatment relates F or W to the triple $(X, \mathcal{C}, \mathcal{F})$, giving us more structure to work with in the temporal operators. For example, using chutes and frontiers allows us to characterize eventuality as the well-foundedness of a subset of the transition relation $\mathcal{R}_{\substack{\mathcal{F} \\ \mathcal{C} \\ X}}$.

Example 1. *Consider the following program for which we would like to prove that the nested property* $\Phi = \mathsf{EF}(\mathsf{EG}(p > 0))$ *holds.*

```
   assume(p == 0 ∧ x > 0); // Initial state
1 while(x>0) {
2    ρ¹ = *;
3    if (ρ¹ > 0) {
4       x = x + 1;
5    } else {
6       x = x - 1;
7    }
8 }
9 while(1) {
10   ρ² = *;
11   if (ρ² > 0)
12      p = 1;
13   else
14      p = 0;
15 }
```

Intuitively, the property holds because there is an execution that exits the first loop, enters the second loop, sets `p = 1` *and then forever iterates the second loop, each time avoiding* `p = 0`. *If we restrict the nondeterministic choices, then we can treat* EFEGp *like* AFAGp. *This can be accomplished by constraints on the* ρ *variables.*

A derivation for this example is given in Figure 3 where $I = \llbracket p = 0 \land x > 0 \rrbracket$. *On the left-hand side is the proof tree, built syntactically from* Φ. *Note that we are using* EGp *which is a special case:* EGp = E[p W false]. *On the right-hand side are values*

for the chutes and frontiers. (Note that `pc` *is a special variable containing the value of the program counter, i.e., line number.) Chute* $\mathcal{C}^\epsilon$ *ensures that in the first loop, executions always take the second branch, and decrement* `x`. *Chute* $\mathcal{C}^\epsilon$ *also ensures that once an execution exits the first loop and enters the second loop, it immediately takes the* `p=1` *branch, satisfying that* $p > 0$. *This is also the point at which frontier* $\mathcal{F}^\epsilon$ *is reached.*

From this state, we can ensure that $\mathsf{EG}(p > 0)$ *holds using chute* $\mathcal{C}^{L\epsilon}$, *which specifies that executions of the second loop only take the* `p=1` *branch. This must hold in every subsequently reachable state (i.e. the semantics of* EG), *so we choose* $\mathcal{F}^{L\epsilon} = \mathsf{false}$. *What remains are the proof obligations:*

1. $(I, \mathcal{C}^\epsilon, \mathcal{F}^\epsilon)$ *is rcr. This triple is recurrent because there is at least one state in* I *(in fact, we can choose any initial state), and whenever we are at the first loop there is at least* some *successor in the chute (iterating* `x=x-1` *or proceeding along the path* `8,9,10,11,12`) *until the frontier* $\mathcal{C}^\epsilon$ *is reached at location* `12`.
2. $\mathcal{R}_{\substack{\mathcal{F}^\epsilon \\ \mathcal{C}^\epsilon \\ I}}$ *is well-founded. This holds because, when executions are restricted to the second branch in the first loop, they are well-founded:* `x` *is decremented and bounded from below by 0.*
3. $(\mathcal{F}^\epsilon, \mathcal{C}^{L\epsilon}, \mathcal{F}^{L\epsilon})$ *is rcr. This set is recurrent because there is always a successor state in* $\mathcal{C}^{L\epsilon}$: *taking the first branch of the second loop ad infinitum.*

This program appears in the EFEGp *benchmark in Figure 6.*

Equivalence to CTL semantics. The following theorem shows that our treatment in Figure 2 is equivalent to CTL. The proof involves representing traces as infinite sequences (*i.e* streams), and numerous support lemmas based on coinduction. We also incorporate recurrent sets and several corresponding lemmas.

Theorem 3.1 (Equivalence to CTL). *For all* $\Phi, M = (S, R, I)$,

$$I \vdash \epsilon, \Phi \iff \forall s \in I.\, R, s \models \Phi.$$

Proof. (sketch) We first expand our proof rules for $X \vdash \kappa, \mathsf{A}\gamma$ and $X \vdash \kappa, \mathsf{E}\gamma$ for each γ case, obtaining four direct $\vdash$ rules: $X \vdash \kappa, \mathsf{AF}\Phi$, $X \vdash \kappa, \mathsf{EF}\Phi$, $X \vdash \kappa, \mathsf{A}[\Phi_1 \ \mathsf{W} \ \Phi_2]$, and $X \vdash \kappa, \mathsf{E}[\Phi_1 \ \mathsf{W} \ \Phi_2]$ (and no $\Vdash$ rules). The proof then proceeds by induction on the (negation normal form) CTL property Φ. $\square$

4. Automation

In this section we present an algorithm for automatically discovering the chutes necessary for proving a CTL property. Our method assumes an underlying proof technique for applying *universal* reasoning to CTL verification, such as the one we previously described [10, 11]. We extend this underlying prover slightly, by adding cases for the *existential* subformulae that are identical to their universal counterparts, except that the transition relation is constrained by the per-subformula chute. Our algorithm begins by assuming that no chute restriction is necessary for proving existential subformulae. This typically leads to counterexamples from the underlying prover. From counterexamples, we synthesize chutes by attempting to eliminate the behaviors witnessed by the counterexamples.

We begin with a symbolic treatment of chutes. Chutes, as seen in Section 3, are (potentially infinite) sets of states. In our refinement procedure we will work with predicates instead, written in first-order logic:

Definition 4.1 (Chute predicate cp). *A chute predicate cp is a first-order logic formula over the states of a transition system.*
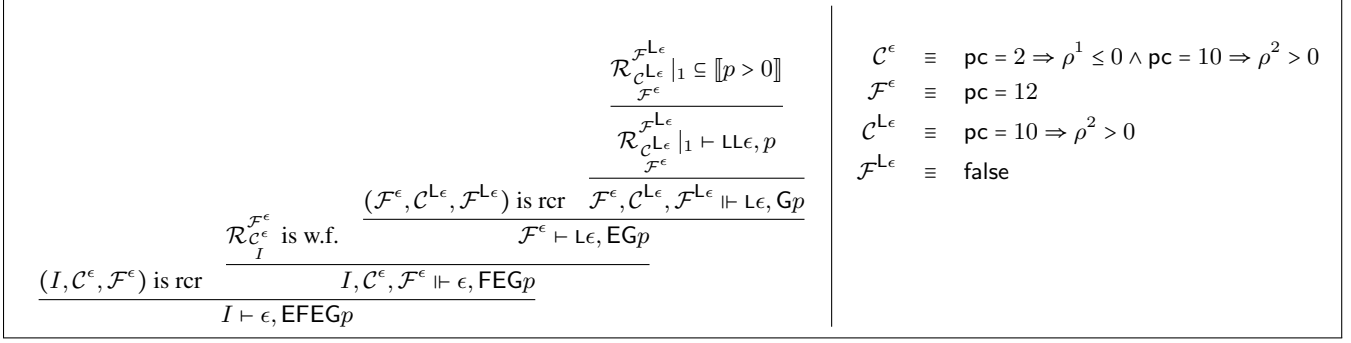
$$
\begin{array}{c}
\cfrac{(I,\mathcal{C}^\epsilon,\mathcal{F}^\epsilon)\text{ is rcr} \quad \cfrac{\mathcal{R}^{\mathcal{F}^\epsilon}_{\mathcal{C}^\epsilon}\,{}_I\text{ is w.f.} \quad \cfrac{(\mathcal{F}^\epsilon,\mathcal{C}^{\mathsf{L}\epsilon},\mathcal{F}^{\mathsf{L}\epsilon})\text{ is rcr} \quad \cfrac{\cfrac{\cfrac{\mathcal{R}^{\mathcal{F}^{\mathsf{L}\epsilon}}_{\mathcal{C}^{\mathsf{L}\epsilon}}\,|_1 \subseteq [\![\,p > 0\,]\!]}{\mathcal{F}^\epsilon}}{\cfrac{\mathcal{R}^{\mathcal{F}^{\mathsf{L}\epsilon}}_{\mathcal{C}^{\mathsf{L}\epsilon}}\,|_1 \vdash \mathsf{LL}\epsilon,\,p}{\mathcal{F}^\epsilon}} \quad \mathcal{F}^\epsilon,\mathcal{C}^{\mathsf{L}\epsilon},\mathcal{F}^{\mathsf{L}\epsilon} \Vdash \mathsf{L}\epsilon,\,\mathsf{G}p}{\mathcal{F}^\epsilon \vdash \mathsf{L}\epsilon,\,\mathsf{EG}p}}{I,\mathcal{C}^\epsilon,\mathcal{F}^\epsilon \Vdash \epsilon,\,\mathsf{FEG}p}}{I \vdash \epsilon,\,\mathsf{EFEG}p}
\qquad
\begin{array}{lll}
\mathcal{C}^\epsilon & \equiv & \mathsf{pc} = 2 \Rightarrow \rho^1 \le 0 \land \mathsf{pc} = 10 \Rightarrow \rho^2 > 0 \\
\mathcal{F}^\epsilon & \equiv & \mathsf{pc} = 12 \\
\mathcal{C}^{\mathsf{L}\epsilon} & \equiv & \mathsf{pc} = 10 \Rightarrow \rho^2 > 0 \\
\mathcal{F}^{\mathsf{L}\epsilon} & \equiv & \mathsf{false}
\end{array}
\end{array}
$$

Figure 3: Derivation for CTL property $\mathsf{EF}(\mathsf{EG}(p > 0))$. On the left is the proof tree and on the right are the values for the chutes and frontiers. Note that we use EG which is a simple case of EW where the second subproperty is simply false.

For a given CTL property Φ, we use the notation $\bar{\mathcal{C}}$ to mean an indexed set of chutes, consisting of a conjunctive chute predicate $\mathcal{C}^\kappa$ for every $(\kappa, \_) \in \mathsf{sub}(\Phi)$. We will abuse notation and sometimes use $\mathcal{C}^\kappa$ to refer to a chute predicate. When this is then used in the context of a set of states, we mean the set of all states such that $\mathcal{C}^\kappa$ holds: $[\![\mathcal{C}^\kappa]\!]$.

Algorithm. Given a CTL property Φ and a program P, our refinement algorithm discovers a proof that $P \vDash \Phi$ by iteratively refining an indexed set of chute predicates $\bar{\mathcal{C}}$. Our algorithm is given in Figure 4. It begins by initializing each $\mathcal{C}^\kappa$ to true. We then use an underlying method for attempting to prove that Φ holds, given the set $\bar{\mathcal{C}}$:

$$\text{attempt } M \vdash \epsilon, \Phi \text{ using } \bar{\mathcal{C}}$$

We discuss our implementation of attempt in Section 5. When attempt succeeds, a candidate proof is returned. This candidate includes an indexed set of frontiers $\bar{\mathcal{F}}$. Together, $\bar{\mathcal{C}}$ and $\bar{\mathcal{F}}$ specify the chutes and frontiers in a $\vdash$ proof tree. Moreover, attempt returns an indexed set of rank functions $\bar{\mathcal{M}}$, which are the proofs of each well-foundedness obligation in F subformulae. What remains are the recurrent set proof obligations for $\mathsf{E}\gamma$ subformulae, accomplished in RCRCHECK (Section 5.3).

When attempt fails, it has discovered a counterexample to Φ in the form of a path $\varpi : \mathsf{list}\ (S \times \mathsf{sub}(\Phi))$.[2] The path ϖ is part of a counterexample to Φ and traverses the $S \times \mathsf{sub}(\Phi)$ state space. (A discussion of counterexamples is given in [9].) Although attempt has failed, Φ may still hold of P. Choices may have been made in $\mathsf{E}\gamma$ subformulae which exercised a region of the state space where γ does not hold. For example, consider the property $\mathsf{EG}(\mathsf{x} = 1)$ and the program:

```
1 x = 1; ρ = *; // init
2 if(ρ > 0)
3   while(1) { x = 0 }
4 else
5   while(1) { x = 1 }
```

Here, attempt may return the path

$$(\mathsf{pc} = 1, \mathsf{x} = 1, \rho = 1),\ (\mathsf{pc} = 3, \mathsf{x} = 0, \rho = 1),\ (\mathsf{pc} = 3, \mathsf{x} = 0, \rho = 1), \ldots$$

While this path proves that $\mathsf{AG}(\mathsf{x} = 1)$ does not hold, it is still the case that $\mathsf{EG}(\mathsf{x} = 1)$ does hold. If we restrict the transition relation's state space to exclude such paths, alternative decisions may be possible, leading to an alternative proof that Φ holds of P. In the case above, if we exclude all of the states where $\mathsf{pc} = 3$ (by

---

[2] In our implementation (see Section 5) this path is represented as a sequence of program commands rather than concrete states.

```
let SYNTHcp(ϖ) ≡
⎰ (cp, κ)      ∃(κ, Eγ) ∈ sub(Φ). ∃i.
⎱                 1. ϖ₀ⁱ ∈ [[Cᴷ]] × {(κ, Eγ)}
                   2. ϖ₀ⁱ⁺¹ ∈ [[Cᴷ]] × {(κ, Eγ)}
                   3. ϖ₀ⁱ⁺¹ ∉ [[Cᴷ ∧ cp]] × {(κ, Eγ)}
  Not_found    otherwise


let prove(M, Φ) =
  C̄ := {κ ↦ true | (κ, Eγ) ∈ sub(Φ)};
  loop
      match (attempt M ⊢ ε, Φ using C̄) with
      | Counterexample(ϖ, F̄) ⟶
          let (cp, κ) = SYNTHcp(ϖ) in
            Cᴷ := Cᴷ ∧ cp
          with Not_found ⟶ return Fail(F̄, C̄)

      | MaybeProof(F̄, M̄) ⟶
          if (∀κ, Eγ ∈ sub(Φ). RCRCHECK(F, Cᴷ, Fᴷ))
            return Succeed
  done
```
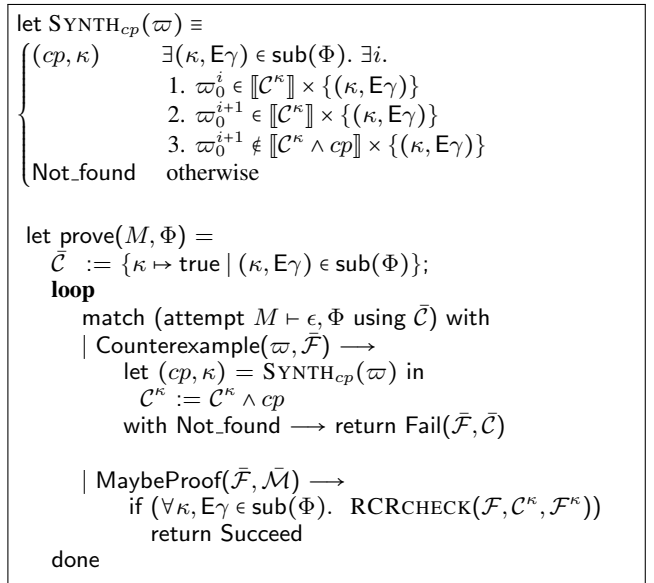
Figure 4: Chute-refinement procedure. Counterexamples from failed proofs using $\forall$CTL techniques with chutes $\mathcal{C}$ are used to help refine $\mathcal{C}$.

ensuring that ρ is never more than zero) then we are left with a transition system for which $\mathsf{AG}(\mathsf{x} = 1)$ holds.

Synthesizing chute predicates. The procedure SYNTH_{cp} is used to automatically discover predicates such as $(\rho \le 0)$, which are necessary to restrict the transition relation to the states in which the existential subproperty holds. The path ϖ returned from attempt is part of a counterexample to Φ and traverses the $S \times \mathsf{sub}(\Phi)$ state space. SYNTH_{cp} must discover a predicate cp such that path ϖ is not possible. To this end, it examines ϖ to find regions of ϖ that are existential temporal operators. Let $(\kappa, \mathsf{E}\gamma)$ be such a region. SYNTH_{cp} must then look for nondeterministic choices that were made in that scope within ϖ where some other alternative choice was possible. In our implementation we can easily find such cases because we have automatically lifted all nondeterministic choice into assignments to special variables denoted ρ_i.

Theorem 4.1 (Soundness of refinement algorithm). *For every transition system M and CTL property Φ,*

$$\mathit{prove}(M, \Phi) \quad \Rightarrow \quad M \vDash \Phi$$

Proof. (Sketch) We first argue that $\mathsf{prove}(M, \Phi) \Rightarrow M \vdash \epsilon, \Phi$ and then use Theorem 3.1. Our algorithm iteratively prunes nondeterministic choices by restricting the transition relation, within a particular subformula, via chute predicates. When proving that a set of states $X \vdash \kappa, \mathsf{E}\gamma$, it is always sound to remove behaviors and apply universal reasoning (in attempt), provided that before we conclude that $\mathsf{E}\gamma$ holds, we ensure that there is at least *one* behavior (accomplished in RCRCHECK). □

In some cases, our refinement algorithm will give an answer that is not the expected one due to an incorrect refinement choice. In cases where an incorrect chute is synthesized we can easily backtrack.

5. Implementation

In this section we describe some implementation-level details that proved important when developing our prototype tool.

5.1 CTL Proof Attempts

Assume for now that we have a set of chute predicates $\bar{\mathcal{C}}$. In Section 4 we assumed the following method:

$$\mathsf{attempt}\ (M \vdash \epsilon, \Phi)\ \mathsf{using}\ \bar{\mathcal{C}}$$

When this procedure fails to prove the property, it returns a path ϖ : $(S \times \mathsf{sub}(\Phi))$ list that is part of a counterexample that demonstrates how Φ can be violated given the current state-space restriction to chutes $\bar{\mathcal{C}}$.

In our implementation we developed this attempt procedure as an extension to our previous method [10, 11], which handles the universal subset of CTL. In our previous work, we introduced a reduction which, when given a transition system P and an $\forall$CTL temporal logic property Φ, generates a procedural program $\mathcal{E}$ that encodes the search for the proof that Φ holds of P [10, 11]. Existing program analysis tools can then be used to reason about the validity of the property.

Our encoding $\mathcal{E}$ is given in Figure 5 and has been extended with the two new rules in large shadowed boxes so that we can now handle full CTL properties. The notation $P[c/c']$ indicates that the program P is modified such that each command c is replaced with a new command c'. When given a program P and a CTL property Φ, the new program $\mathcal{E}$ encodes the search for the proof that Φ holds of P. The arguments $(\langle s, \psi \rangle, \mathcal{M}, R)$ passed to $\mathcal{E}$ are a pair consisting of the state s and a Φ-subformula ψ of interest, a finite set of ranking functions $\mathcal{M}$, an indexed set of chute predicates $\bar{\mathcal{C}}$ and the program source P. Executions of the procedure $\mathcal{E}$ explore the $S \times \mathsf{sub}(\Phi)$ state space from an initial state $s_0 \in I$ in a depth-first manner. At each recursive call, $\mathcal{E}$ is attempting to determine whether ψ holds of s. Rather than explicitly tracking this information, however, $\mathcal{E}$ returns false (recursively) whenever ψ does not hold of s. Consequently, if $\mathcal{E}$ can be proved to never return false, it must be the case that the overall property Φ holds of the initial state s. When a program analysis is applied to $\mathcal{E}$ it is implementing what is needed to prove branching-time behaviors of the original transition system (*e.g.* backtracking, eventuality checking, tree counterexamples, abstraction, abstraction-refinement, etc). Formally the relationship between $\mathcal{E}$ and $\vdash$ is: for a program P (with initial states I) and CTL property Φ,

$$\exists \bar{\mathcal{C}}.\ \exists \bar{\mathcal{M}}.\ \forall s \in I.\ \mathcal{E}(s, \Phi, \bar{\mathcal{M}}, \bar{\mathcal{C}}, R)\ \text{cannot return false}$$
$$\Rightarrow$$
$$\forall s \in I.\ R, s \vDash \Phi$$

What remains is to understand how $\mathcal{E}$ determines whether a subformula ψ holds of a state s. By passing the state on the stack, we can consider multiple branching scenarios. When a particular ψ is a $\wedge$ or AW/EW subformula, then $\mathcal{E}$ ensures that all possibilities are considered by establishing feasible paths to all of them. When

Figure 5: Encoding CTL verification as a finite set of procedures. Universal rules discussed in [10, 11]. The two new existential rules, which account for (κ, ψ)-subformula indexed sets of chute predicates are highlighted with shadow-boxes.

a particular ψ is a $\vee$ or AF/EF subformula, $\mathcal{E}$ enables executions to consider all of the possible cases that might cause ψ to hold of s. As soon as one is found, true is returned. Otherwise, false will be returned if none are found.

New existential cases. The encoding has two new cases, in large shadowed boxes, for the existential operators: the new subformula encoding $\mathsf{enc}^\kappa_{\mathsf{EF}\psi}$ which is similar to $\mathsf{enc}^\kappa_{\mathsf{AF}\psi}$, as well as encoding $\mathsf{enc}^\kappa_{\mathsf{E}[\psi_1\ \mathsf{W}\ \psi_2]}$ which is similar to $\mathsf{enc}^\kappa_{\mathsf{A}[\psi_1\ \mathsf{W}\ \psi_2]}$. They are designed such that we can again apply the universal reasoning and try to prove that $\mathsf{enc}^\kappa_{\mathsf{EF}\psi}$ (or $\mathsf{enc}^\kappa_{\mathsf{E}[\psi_1\ \mathsf{W}\ \psi_2]}$) cannot return false. What is new is that $\mathsf{enc}^\kappa_{\mathsf{EF}\psi}$ and $\mathsf{enc}^\kappa_{\mathsf{E}[\psi\mathsf{W}\psi']}$ must take care to restrict possible proofs/counterexamples as specified by the chute predicates we have discovered. This is where the indexed vector of chute predicates $\bar{\mathcal{C}}$ is used. The encodings for $\mathsf{enc}^\kappa_{\mathsf{EF}\psi}$ and $\mathsf{enc}^\kappa_{\mathsf{E}[\psi\mathsf{W}\psi']}$ in-

clude wavy underlined `assume`($\mathcal{C}^\kappa$) statements in Figure 5. These statements restrict the behavior of the transition system to chute $\mathcal{C}^\kappa$.

5.2 Synthesizing Chute Predicates

Our refinement algorithm, given in Figure 4, then assumes a method for synthesizing new chute predicates, given counterexamples to proof attempts. In our implementation, when the proof attempt fails a path $\varpi : (c, \kappa, \psi)$ list is returned, which is a sequence of commands in the original program P, annotated with the relevant subformula. Our implementation of $\textsc{Synth}_{cp}$ must return a chute predicate cp and subformula index κ that meets the interface given in Figure 4. When our implementation of `attempt` generates the encoding $\mathcal{E}$, it introduces a variable ρ_i at each nondeterministic choice. For example, we would transform nondeterministic assignments as follows:

$$\texttt{x := read();} \quad \rightsquigarrow \quad \rho_x \texttt{ := *; x := } \rho_x;$$

We similarly transform nondeterministic branching:

$$\texttt{if (read()) then } C_1 \texttt{ else } C_2$$
$$\rightsquigarrow \quad \rho_i \texttt{ := *; if (}\rho_i\texttt{>0) then } C_1 \texttt{ else } C_2$$

These transformations simplify chute predicate synthesis by standardizing the sources of nondeterminism as values of ρ-variables. When we are given a path ϖ, we look for a ρ-variable:

$$..., (c_{i-1}, \kappa_{i-1}, \psi_{i-1}), (\underset{\sim}{\rho_i \texttt{:=*}}, \kappa_i, \mathsf{E}\gamma_i), (c_{i+1}, \kappa_{i+1}, \psi_{i+1}), ...$$

Given such a path, we restrict our attention to the commands that are annotated with $(\kappa_i, \mathsf{E}\gamma_i)$, as the other commands are only relevant to other subformulae. We then build a formula T that is the conjunction of inequalities in static single-assignment form that represents the remaining commands. Assume that V is the variables *not* in the $(\kappa_i, \mathsf{E}\gamma_i)$ scope. Using quantifier elimination we compute $T' = \exists V.T$. Our procedure then ensures that T' is in CNF form: $T' = \bigwedge_i T'_i$ where each T'_i is an inequality. We prune T' ensuring that each conjunct mentions the variable ρ:

$$T_\rho = \neg \bigwedge \{T'_i \mid \rho \in \textsc{FreeVars}(T'_i)\}$$

The new candidate for chute refinement is thus $\texttt{pc} = \ell \Rightarrow T_\rho$ where ℓ is assumed to be the location of the command $\rho := *$.

Example. Imagine that we have the following path:

(...;	,	($\mathsf{R}\epsilon, \mathsf{E}\gamma \vee \mathsf{AF}p$))
(x:=a; y:=b; v:=c; y:=k;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(rho := *;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(assume(rho>3);	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(x := w;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(y := y-1;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(assume(x<z);	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(rho := *;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(assume(rho>x);	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(x := x - 1;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(v := y;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(rho := *;	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))
(assume(rho>3);	,	($\mathsf{LR}\epsilon, \mathsf{E}\gamma$))

The SSA formula representing this would be:

$$
\begin{aligned}
x_1 &= a_0 \\
y_1 &= b_0 \\
v_1 &= c_0 \\
y_1 &= k_0 \\
rho_1 &> 3 \\
x_2 &= w_0 \\
y_2 &= y_1 - 1 \\
x_2 &< z_0
\end{aligned}
\qquad
\begin{aligned}
rho_2 &> x_2 \\
x_3 &= x_2 - 1 \\
v_2 &= y_1 \\
rho_3 &> 3 \\
a_1 &= x_3 \\
b_1 &= y_2 \\
c_1 &= v_2
\end{aligned}
$$

Imagine that we'd like to find a condition from the second `rho := *`. In this case we quantify out all variables except x_2, y_2, v_1, k_0, z_0, and rho_2. That leaves us with the condition: $rho_2 > x_2$

Thus we might use $\neg(rho_2 > x_2)$ as the refinement for $\mathcal{C}^{\mathsf{LR}\epsilon}$. There are usually many choices available, some of which may lead to failed proofs due to a violation in RCRCHECK. To filter out potentially bad chute predicates, our implementation performs a simple non-termination check of the scope in question. If multiple candidates still exist we choose the one that appears as the last assignment in the inner most scope. We have found that these heuristics for choosing chute predicates was effective at proving a wide variety of CTL properties, as seen in our experimental results (Section 6).

5.3 Checking Recurrent Sets

Finally, when a proof `attempt` succeeds, recall that we must check that our chutes have not restricted the state space to empty traces. To this end, we must check that chutes for existential subformulae are *recurrent*. This does not necessarily mean that we must check all subformula. For example, the property $(\mathsf{EG}p) \vee (\mathsf{EG}q)$ holds if $\mathsf{EG}p$ holds. In this case, we only need to ensure that the chute $\mathcal{C}^{\mathsf{LL}\epsilon}$ for the left operator is recurrent. A proof that $M \vdash \epsilon, \Phi$ involves indexed sets $\bar{\mathcal{C}}$ and $\bar{\mathcal{F}}$. For each recurrent set obligation, we can use a simple reduction to satisfiability.

Inside our implementation of `attempt`, for every subformula $(\kappa, \mathsf{E}\gamma)$, there is an encoding of the relevant subset of the transition relation, restricted by chute predicate $\mathcal{C}^\kappa$. Our implementation uses known non-termination proving techniques to prove, in the case of EG, that there is at least one non-terminating execution and, in the case of EF that there at least one execution to a place where the subproperty may hold.

6. Evaluation

Using our prototype implementation we performed two sets of experiments. First, we ran the tool on a series of small benchmarks crafted to explore the various combinations of temporal operators. We then used the tool on a set of examples drawn from industrial code bases. In both cases, we generated the encoding and ran our algorithm on an Intel x64-based 2.8 GHz single-core processor. The sources of our experiments are available—please contact the authors. As discussed in Section 1, our tool is the first to handle mixtures of universal and existential quantifiers. Thus at this time there are no competing tools to compare against. Note that, as our tool is not especially optimized, the purpose of these experiments is only to demonstrate the promise of the approach.

For our small benchmarks, we wrote example programs in C. For each example, we wrote down corresponding CTL properties. The results of these experiments are given in Figure 6. Some of the CTL properties are meant to hold (denoted $\checkmark$) and others are meant not to hold (denoted χ). We report the time it took to prove or discover a counterexample in the **Time** column. In each case, we constructed another benchmark for the corresponding negated property (using the same program). For example, in the benchmark 22 where $\mathsf{EGAG}p$ holds, we ensure that $\mathsf{AFEF}\neg p$ does not hold of benchmark 49. The $\mathsf{EFEG}p$ benchmark is displayed in Example 1.

In all but two cases we were able to prove that the property held or discover a counterexample in less than a minute, often in seconds or less. Benchmark #20 ran out of memory during abstraction refinement in the underlying safety proof. In benchmark #24 of Figure 6, our tool failed to report the correct answer for the property $\mathsf{EG}(q \Rightarrow \mathsf{EF}p)$. This is because our tool made the wrong choice about synthesizing chute, but restricting execution to a particular set of paths (via a predicate on ρ). However, a more mature version of our tool can simply backtrack and make other

#	Property	Result		Time
		Exp.	Act.	(s)
1.	AFp	✓	✓	1.2
2.	AFp	χ	χ	0.8
3.	AGp	✓	✓	0.3
4.	AGp	χ	χ	0.5
5.	EFp	✓	✓	
6.	EFp	χ	χ	1.7
7.	EGp	✓	✓	0.9
8.	EGp	χ	χ	0.9
9.	AGAFp	✓	✓	10.8
10.	AGAFp	χ	χ	1.9
11.	AGEFp	✓	✓	29.0
12.	AGEGp	✓	✓	1.2
13.	AFEGp	✓	✓	55.8
14.	AFEFp	✓	✓	3.7s
15.	AFAGp	✓	✓	1.3
16.	AFAGp	χ	χ	11.0
17.	EFEGp	✓	✓	44.3
18.	EFEGp	χ	χ	54.7
19.	EFAGp	✓	✓	0.6
20.	EFAFp	✓	?	*mem*
21.	EGEFp	χ	χ	10.4
22.	EGAGp	✓	✓	0.8
23.	EGAFp	✓	✓	12.5
24.	EG$(q \Rightarrow$ EF$p)$	✓	χ	33.9
25.	EG$(q \Rightarrow$ AF$p)$	χ	χ	150.2
26.	AG$(q \Rightarrow$ EG$p)$	✓	✓	2.2
27.	AG$(q \Rightarrow$ EF$p)$	✓	✓	29.5
28.	EG $\neg p$	χ	χ	0.8
29.	EG $\neg p$	✓	✓	0.9
30.	EF $\neg p$	χ	χ	0.5
31.	EF $\neg p$	✓	✓	0.6
32.	AG $\neg p$	χ	χ	0.8
33.	AG $\neg p$	✓	✓	2.0
34.	AF $\neg p$	χ	χ	0.5
35.	AF $\neg p$	✓	✓	0.4
36.	EFEG$\neg p$	χ	χ	1.9
37.	EFEG$\neg p$	✓	✓	3.6
38.	EFAG$\neg p$	χ	χ	3.9
39.	EFAF$\neg p$	χ	χ	6.3
40.	EGAF$\neg p$	χ	χ	10.9
41.	EGAG$\neg p$	χ	χ	37.7
42.	EGEF$\neg p$	χ	χ	2.7
43.	EGEF$\neg p$	✓	✓	5.8
44.	AGAF$\neg p$	χ	χ	3.6
45.	AGAF$\neg p$	✓	✓	10.2
46.	AGEF$\neg p$	χ	χ	23.8
47.	AGEG$\neg p$	χ	χ	7.5
48.	AFAG$\neg p$	✓	✓	40.3
49.	AFEF$\neg p$	χ	χ	0.9
50.	AFEG$\neg p$	χ	χ	13.8
51.	AF$(q \wedge$ AG$\neg p)$	χ	χ	2.0
52.	AF$(q \wedge$ EG$\neg p)$	χ	χ	13.8
53.	EF$(q \wedge$ AF$\neg p)$	χ	χ	6.3
54.	EF$(q \wedge$ AG$\neg p)$	χ	χ	3.9

Figure 6: The performance of our tool when applied to small benchmarks. We report the property proved/disproved, the expected/actual result, and the time. For each property (1–27) we have also attempted to prove/disprove the negated property (28–54). Benchmarks 28–54 are the same as 1–27 but with the property negated.

#	Example	LOC	Property shape	Result		Time
				Exp.	Act.	(s)
1	OS frag. 1	29	AG$(p \Rightarrow$ AF$q)$	✓	✓	4.6
2	OS frag. 1	29	AG$(p \Rightarrow$ AF$q)$	χ	χ	9.1
3	OS frag. 1	29	AG$(p \Rightarrow$ EF$q)$	✓	✓	9.5
4	OS frag. 1	29	AG$(p \Rightarrow$ EF$q)$	χ	χ	1.5
5	OS frag. 2 [8]	58	AG$(p \Rightarrow$ AF$q)$	✓	✓	2.1
6	OS frag. 2 [8]	58	AG$(p \Rightarrow$ AF$q)$	χ	χ	1.8
7	OS frag. 2 [8]	58	AG$(p \Rightarrow$ EF$q)$	✓	✓	3.7
8	OS frag. 2 [8]	58	AG$(p \Rightarrow$ EF$q)$	χ	χ	1.5
9	OS frag. 3	370	AG$(p \Rightarrow$ AF$q)$	✓	✓	38.9
10	OS frag. 3	370	AG$(p \Rightarrow$ AF$q)$	χ	χ	18.0
11	OS frag. 3	370	AG$(p \Rightarrow$ EF$q)$	✓	✓	90.0
12	OS frag. 3	370	AG$(p \Rightarrow$ EF$q)$	χ	χ	107.8
13	OS frag. 4	370	AF$q \vee$ AFp	✓	✓	34.3
14	OS frag. 4	370	AF$q \vee$ AFp	χ	χ	18.8
15	OS frag. 4	370	EF$q \wedge$ EFp	✓	✓	1261
16	OS frag. 4	370	EF$q \wedge$ EFp	χ	?	*mem*
17	OS frag. 5	43	AGAFp	✓	✓	569.7
18	OS frag. 5	43	AGAFp	χ	χ	65.1
19	OS frag. 5	43	AGEFp	✓	?	*time*
20	OS frag. 5	43	AGEFp	χ	?	*mem*
21	PgSQL arch	90	AGAFp	✓	?	*mem*
22	PgSQL arch	90	AGAFp	χ	χ	38.1
23	PgSQL arch	90	AGEFp	✓	?	*mem*
24	PgSQL arch	90	AGEFp	χ	χ	42.7
25	S/W Updates	36	$p \Rightarrow$ AFq	✓	✓	70.2
26	S/W Updates	36	$p \Rightarrow$ AFq	χ	χ	32.4
27	S/W Updates	36	$p \Rightarrow$ EFq	✓	✓	18.5
28	S/W Updates	36	$p \Rightarrow$ EFq	χ	χ	1.3
29	OS frag. 1	29	EF$(p \wedge$ EG$\neg q)$	χ	χ	12.5
30	OS frag. 1	29	EF$(p \wedge$ EG$\neg q)$	✓	✓	3.5
31	OS frag. 1	29	EF$(p \wedge$ AG$\neg q)$	χ	χ	18.1
32	OS frag. 1	29	EF$(p \wedge$ AG$\neg q)$	✓	✓	105.7
33	OS frag. 2 [8]	58	EF$(p \wedge$ EG$\neg q)$	χ	χ	6.5
34	OS frag. 2 [8]	58	EF$(p \wedge$ EG$\neg q)$	✓	✓	1.2
35	OS frag. 2 [8]	58	EF$(p \wedge$ AG$\neg q)$	χ	χ	8.7
36	OS frag. 2 [8]	58	EF$(p \wedge$ AG$\neg q)$	✓	✓	5.6
37	OS frag. 3	370	EF$(p \wedge$ EG$\neg q)$	χ	χ	1930.9
38	OS frag. 3	370	EF$(p \wedge$ EG$\neg q)$	✓	✓	1680.7
39	OS frag. 3	370	EF$(p \wedge$ AG$\neg q)$	χ	?	*mem*
40	OS frag. 3	370	EF$(p \wedge$ AG$\neg q)$	✓	?	*mem*
41	OS frag. 4	370	EG$\neg p \wedge$ EG$\neg q$	χ	χ	61.3
42	OS frag. 4	370	EG$\neg p \wedge$ EG$\neg q$	✓	✓	7.6
43	OS frag. 4	370	AF$\neg q \vee$ AF$\neg p$	χ	χ	0.9
44	OS frag. 4	370	AF$\neg q \vee$ AF$\neg p$	✓	✓	0.6
45	OS frag. 5	43	EFEG$\neg p$	χ	χ	1471.7
46	OS frag. 5	43	EFEG$\neg p$	✓	✓	351.1
47	OS frag. 5	43	EFAG$\neg p$	χ	χ	85.5
48	OS frag. 5	43	EFAG$\neg p$	✓	✓	255.8
49	PgSQL arch	90	EFEG$\neg p$	χ	χ	45.3
50	PgSQL arch	90	EFEG$\neg p$	✓	✓	35.2
51	PgSQL arch	90	EFAG$\neg p$	χ	?	*mem*
52	PgSQL arch	90	EFAG$\neg p$	✓	✓	30.2
53	S/W Updates	36	$p \wedge$ EG$\neg q$	χ	χ	0.4
54	S/W Updates	36	$p \wedge$ EG$\neg q$	✓	✓	4.5
55	S/W Updates	36	$p \wedge$ AG$\neg q$	χ	χ	0.5
56	S/W Updates	36	$p \wedge$ AG$\neg q$	✓	✓	0.3

Figure 7: The results of applying our refinement algorithm on a variety of examples from industrial code. In each case, we report the shape of the property, the expected/actual result returned by our tool, and the time it took to prove the property or discover a counterexample. Benchmarks 29–56 are the same as 1–28 but with the property negated.

choices. Understanding these tradeoffs is an important direction for future work.

In Figure 7 we report the results of applying our tool to CTL challenge problems drawn from industrial code bases. The 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 SoftUpdates patch system [25]. Line counts are given in the third column. In many of these 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. Note that this abstraction often allows us to reason about nondeterminism related to data-structures (e.g. a subroutine that increases the length of a list by some nondeterministic amount).

For each benchmark, we considered a meaningful property. Many are standard acquire/release or malloc/free-style properties ($AG[p \Rightarrow AFq]$) and, for example, the OS frag. 4 property (Figure 7, #13) says that either an I/O completion occurs successfully or else a failure code is returned. Figure 7 reports the shape of these properties (eliding the details of the atomic propositions p, q, etc.).

The final columns of Figure 7 report the expected/actual result returned by our tool, and the time it took to prove the property or discover a counterexample. In the vast majority of the cases, we were able to prove/disprove the property, often very quickly. In some cases, our implementation ran out of memory (*mem*) while performing $SYNTH_{cp}$. We believe that these examples could be proved if we applied program slicing on the encoding $\mathcal{E}$. In one case our implementation timed out (*time*) after 24 hours.

In summary, we can prove CTL properties of C functions that have hundreds of lines of code. For example, OS frag. 3 is 370 LOC. The majority of the examples (both the small benchmarks in Figure 6 and industrial code in Figure 7) were examples that previously could not be proved. The exceptions are the single-temporal-operator small benchmarks (which reduce to simple safety/liveness) and the strictly-universal industrial examples such as #21, which can be treated by previous work [11].

It is reasonable to ask about a comparison between our tool on the property $AFfalse$ (which encodes termination) and a termination prover, or $EGtrue$ and a non-termination prover. In fact, in our system the encoding of $AFfalse$ is isomorphic to the reduction used in tools such as TERMINATOR [12], similarly $EGtrue$ in our system reduces immediately to the identical techniques used in current non-termination provers. Thus, the experimental results are essentially identical, with the variability in the performance only due to the level of optimizations implemented in the underlying tools.

7. Related Work

Temporal property verification (and related problems such as finding winning strategies in games) have been extensively studied for finite-state systems (*e.g.* [1, 4, 6, 7, 28, 37]). Temporal property verification for some limited types of infinite-state systems have also been studied (*e.g.* pushdown systems [36, 39, 40], parameterized systems [17], etc.). In limited cases we can abstract programs to finite-state systems and apply these known methods, but existing abstraction methods usually do not allow us to reliably prove properties that mix universal and existential modal operators in nontrivial ways. Methods for proving *linear-time* properties of infinite-state programs have been studied in recent work (*e.g.* [8, 9, 13, 33]). These techniques do not, however, facilitate reasoning about the nondeterminism in systems (see [9] for a discussion on this).

Methods of proving branching-time properties with only universal path quantifiers are known (*e.g.* [5, 10, 11, 13]). Techniques are also known for proving AF false (*a.k.a.* termination [3, 12, 18]) as

well as EG true (*a.k.a.* non-termination [22]). In this work we build on these known techniques.

Previous tools based on fixed *a priori* abstractions are known for CTL. The abstraction used by YASM [23] is aimed primarily at the unnested existential subset of CTL. The work of Song and Touili [36] abstracts the program to pushdown systems. The difference here is that we reduce the problem of CTL (in a way that information is not lost) to different problems, which allows abstractions to be computed by refinement later using known techniques.

Although we solve a problem more general than termination, our iterative algorithm for finding restrictions on the state-space shares some similarity with the idea of alternation used in Harris *et al.* [24]. Related approaches are also seen in Godefroid *et al.* [19] and Gulavani *et al.* [20] for proving safety properties.

Our work characterizes CTL verification in a way that is complete upto a certain parameter (*chutes*) and gives a method and implementation for discovering these parameters automatically. In this sense it may be applicable to discovering focus sets [14] or the necessary ingredients in a games-based abstraction framework [15]. Our paper also might contribute towards novel approaches to the known problem of supporting mixtures of may/must [19]. Specifically we contribute a reduction that facilitates existing successful universal techniques (*e.g.* predicate abstraction, interpolation, etc) and gives a more general underapproximation for existential properties than previously known approaches to may/must abstraction.

Kesten and Pnueli [27] describe a proof system for CTL^*. Due to our desire for full automation, ours differs in significant ways. For example, while a proof of eventuality (AFq or EFq) in our case involves reasoning along paths until q, we do not decompose into path assertions in the proof system. Instead, we identify the appropriate fragment of the transition relation (*i.e.* $\mathcal{R}_{C}^{\mathcal{F}}{}_{X}$) and a general property that must hold of this transition relation (wellfoundedness). In this way, we leave the details of eventuality proving to underlying abstraction techniques such as interpolation or abstraction refinement. This simplifies the proof system and lends itself many different forms of reasoning over $\mathcal{R}_{C}^{\mathcal{F}}{}_{X}$.

There are a few other connections to existing work. Pruning paths (as we do in chute refinement) appears in many static analysis techniques [21, 41]. Second, in our work we synthesize program variables that correspond to properties of the transition system (or of the desired system). The success of similar techniques in other contexts [35] reinforces the practical potential and experimental results of our paper. Finally, Vardhan and Viswanathan [38] describe tool called Lever which uses language inference to discover fixed points. These fixed points could potentially be used in conjunction with our method once we have discovered the appropriate state restrictions.

8. Concluding Remarks

Until now, the question of how to reliably automate proofs in branching-time logics (such as CTL, CTL^* and the modal μ-calculus) for infinite-state programs remained an open problem. In this paper we have reported on the first known robust automatic proof method to support both universal and existential branching-time temporal operators (in CTL) for programs. In our approach existential reasoning is reduced to universal reasoning together with the search for a restrictions on the program's state-space. The advantage of the approach is that there are known solutions for universal reasoning. To demonstrate the practical viability of our approach we discussed the outcome of an experimental evaluation with a prototype implementation using examples drawn the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

There are a few open problems left as future work. It would be helpful to have a more extensive evaluation considering, for example, performance of properties that have high levels of nesting in the temporal operators. Further, one might bring these techniques to the interprocedural level, investigating the use of temporal lemmas to optimize the performance of the proof search. Finally, these techniques could be extended to more expressive logics such as CTL* and the modal μ-calculus.

Acknowledgments

References

[1] BERNHOLTZ, O., VARDI, M. Y., AND WOLPER, P. An automata-theoretic approach to branching-time model checking (extended abstract). In *CAV'94* (1994), D. L. Dill, Ed., vol. 818, Springer, pp. 142–155.

[2] BEYER, D., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. The software model checker blast. *STTT 9*, 5-6 (2007), 505–525.

[3] BRADLEY, A., MANNA, Z., AND SIPMA, H. The polyranking principle. *Automata, Languages and Programming* (2005), 1349–1361.

[4] BURCH, J., CLARKE, E., ET AL. Symbolic model checking: 10^{20} states and beyond. *Information and computation 98*, 2 (1992), 142–170.

[5] CHAKI, S., CLARKE, E. M., GRUMBERG, O., OUAKNINE, J., SHARYGINA, N., TOUILI, T., AND VEITH, H. State/event software verification for branching-time specifications. In *IFM'05* (2005), J. Romijn, G. Smith, and J. van de Pol, Eds., vol. 3771, pp. 53–69.

[6] CLARKE, E., JHA, S., LU, Y., AND VEITH, H. Tree-like counterexamples in model checking. In *LICS* (2002), pp. 19–29.

[7] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS 8* (April 1986), 244–263.

[8] COOK, B., GOTSMAN, A., PODELSKI, A., RYBALCHENKO, A., AND VARDI, M. Y. Proving that programs eventually do something good. In *POPL'07* (2007), pp. 265–276.

[9] COOK, B., AND KOSKINEN, E. Making prophecies with decision predicates. In *POPL'11* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 399–410.

[10] COOK, B., KOSKINEN, E., AND VARDI, M. Temporal verification as a program analysis task [extended version]. *FMSD* (2012).

[11] COOK, B., KOSKINEN, E., AND VARDI, M. Y. Temporal property verification as a program analysis task. In *CAV'11* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806, Springer, pp. 333–348.

[12] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *PLDI'06* (2006), M. I. Schwartzbach and T. Ball, Eds., pp. 415–426.

[13] COUSOT, P., AND COUSOT, R. An abstract interpretation framework for termination. In *POPL'12* (2012), ACM, pp. 245–258.

[14] DAMS, D., AND NAMJOSHI, K. S. The existence of finite abstractions for branching time model checking. In *LICS* (2004), pp. 335–344.

[15] DE ALFARO, L., GODEFROID, P., AND JAGADEESAN, R. Three-valued abstractions of games: Uncertainty, but with precision. In *LICS* (2004), pp. 170–179.

[16] EMERSON, E. A., AND HALPERN, J. Y. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM 33*, 1 (1986), 151–178.

[17] EMERSON, E. A., AND NAMJOSHI, K. S. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV'96* (1996), vol. 1102, pp. 87–98.

[18] GIESL, J., SCHNEIDER-KAMP, P., AND THIEMANN, R. Aprove 1.2: Automatic termination proofs in the dependency pair framework. *Automated Reasoning* (2006), 281–286.

[19] GODEFROID, P., NORI, A. V., RAJAMANI, S. K., AND TETALI, S. Compositional may-must program analysis: unleashing the power of alternation. In *POPL'10* (2010), ACM, pp. 43–56.

[20] GULAVANI, B. S., HENZINGER, T. A., KANNAN, Y., NORI, A. V., AND RAJAMANI, S. K. SYNERGY: a new algorithm for property checking. In *FSE'06* (2006), ACM, pp. 117–127.

[21] GULWANI, S., JAIN, S., AND KOSKINEN, E. Control-flow refinement and progress invariants for bound analysis. In *PLDI'09* (2009), pp. 375–385.

[22] GUPTA, A., HENZINGER, T. A., MAJUMDAR, R., RYBALCHENKO, A., AND XU, R.-G. Proving non-termination. *SIGPLAN Not. 43* (January 2008), 147–158.

[23] GURFINKEL, A., WEI, O., AND CHECHIK, M. Yasm: A software model-checker for verification and refutation. In *CAV'06* (2006), vol. 4144, pp. 170–174.

[24] HARRIS, W. R., LAL, A., NORI, A. V., AND RAJAMANI, S. K. Alternation for termination. In *SAS* (2010).

[25] HAYDEN, C. M., MAGILL, S., HICKS, M., FOSTER, N., AND FOSTER, J. S. Specifying and verifying the correctness of dynamic software updates. In *VSTTE'12* (2012), vol. 7152, pp. 278–293.

[26] IOSIF, R., BOZGA, M., BOUAJJANI, A., HABERMEHL, P., MORO, P., , AND VOJNAR, T. Programs with lists are counter automata. In *CAV* (2006).

[27] KESTEN, Y., AND PNUELI, A. A compositional approach to ctl* verification. *Theor. Comput. Sci. 331*, 2-3 (2005), 397–428.

[28] KUPFERMAN, O., VARDI, M., AND WOLPER, P. An automata-theoretic approach to branching-time model checking. *Journal of the ACM 47*, 2 (2000), 312–360.

[29] MAGILL, S., TSAI, M.-H., LEE, P., AND TSAY, Y.-K. Automatic numeric abstractions for heap-manipulating programs. In *POPL'10* (2010), ACM, pp. 211–222.

[30] MCMILLAN, K. L. Lazy abstraction with interpolants. In *CAV'06* (2006), T. Ball and R. B. Jones, Eds., vol. 4144, pp. 123–136.

[31] NELSON, G. A generalization of Dijkstra's calculus. *TOPLAS 11*, 4 (1989), 517–561.

[32] PISTORE, M., AND TRAVERSO, P. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI'01* (2001), Springer.

[33] PNUELI, A., AND ZAKS, A. Psl model checking and run-time verification via testers. In *FM* (2006), pp. 573–586.

[34] PODELSKI, A., AND RYBALCHENKO, A. Transition invariants. In *LICS* (2004), pp. 32–41.

[35] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *PLDI* (2006), ACM, pp. 404–415.

[36] SONG, F., AND TOUILI, T. Pushdown model checking for malware detection. In *TACAS* (2012).

[37] STIRLING, C. Games and modal mu-calculus. In *TACAS* (1996), vol. 1055, pp. 298–312.

[38] VARDHAN, A., AND VISWANATHAN, M. Learning to verify branching time properties. *FMSD 31*, 1 (2007), 35–61.

[39] WALUKIEWICZ, I. Pushdown processes: Games and model checking. In *CAV* (1996), vol. 1102, pp. 62–74.

[40] WALUKIEWICZ, I. Model checking ctl properties of pushdown systems. In *FSTTCS* (2000), S. Kapoor and S. Prasad, Eds., vol. 1974, pp. 127–138.

[41] YANG, Z., AL-RAWI, B., SAKALLAH, K. A., HUANG, X., SMOLKA, S. A., AND GROSU, R. Dynamic path reduction for software model checking. In *IFM* (2009), vol. 5423, pp. 322–336.