Temporal Property Verification as a Program Analysis Task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2,5,8,24,32]) together with techniques for discovering termination arguments (*e.g.* [3,6,17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL ($\forall$CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* $\forall$CTL rather than CTL). Existential reasoning would also be possible, but care is required to ensure that

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 333–348, 2011.

the underlying program analysis tools appropriately use universal abstractions ("may" transitions) as well as existential abstractions ("must" transitions). Finally, our method works best when properties do not involve deep and complex nesting of temporal operators. In order to better support these more complex properties our implementation would need to mix the construction of the program analysis problem with the analysis itself in the spirit of Impact [27], as invariants proved during a lazy unrolling could be used to prune away much of the work. As presented here, our approach instead creates a single encoding up front before performing program analysis.

2 From Temporal Logic to Program Analysis

In this section we introduce a reduction which, when given a transition system M and an $\forall$CTL temporal logic property φ, generates a program that encodes the search for the proof that φ holds of M. Existing program analysis tools can then be used to reason about the validity of the property. We begin with some definitions and terminology.

2.1 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 $I \subseteq S$. A *trace* of a transition system is a sequence of states $(s_0, s_1, ...)$ such that $s_0 \in I$ and $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$. For convenience, we do not allow finite traces. The transition relation must be such that every state has at least one successor state: $\forall s \in S.\ \exists s'.\ R(s, s')$. This is without a loss of generality, as final states can be encoded as states that loop back to themselves.

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 [30]. 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. We are concerned with verifying temporal properties that may be written either as trace-based properties in LTL or as state-based properties in the universal fragment of computation tree logic ($\forall$CTL). The encoding we describe in this section is state-based in nature and, as such, is readily suitable to $\forall$CTL properties. To prove LTL properties we use a recently described iterative symbolic determinization technique [15] with the $\forall$CTL proving technique described here.

The syntax of a $\forall$CTL formula is $\varphi ::= \alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AG}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi]$. The standard semantics of $\forall$CTL are given in Fig. 1. α is an atomic proposition. $\forall$CTL's temporal operators are state-based in structure. The operator $\mathsf{AG}\varphi$

specifies that φ globally holds in all reachable future states. The operator $\mathsf{AF}\varphi$ specifies that, across all computation sequences from the current state, a state in which φ holds must be reached. Finally, the $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ operator specifies that φ_1 holds in every state where φ_2 does not yet hold.

We use $\mathsf{AF}, \mathsf{AG}, \mathsf{AW}$ as our base operators (as opposed to the more standard U and R), as each corresponds to a distinct form of proof: AF to termination, AG to safety, and AW to sequencing. We omit the next state operator AX. Formulae with U and R can be expressed in $\forall$CTL. We assume that formulae are written in negation normal form, in which negation only occurs next to atomic propositions (we also assume that the domain of atomic propositions is closed under negation). A formula that is not in negation normal form can be easily normalized. $\mathsf{sub}(\varphi)$ is defined to be the set of all subformulae of φ.

$$\frac{\alpha(s)}{R, s \vDash \alpha} \qquad \frac{R, s \vDash \varphi_1 \quad R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \wedge \varphi_2} \qquad \frac{R, s \vDash \varphi_1 \ \vee \ R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \vee \varphi_2}$$

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

$$\frac{\forall(s_0, s_1, ...).\ s_0 = s \Rightarrow \forall i \geq 0.\ R, s_i \vDash \varphi}{R, s \vDash \mathsf{AG}\varphi}$$

$$\frac{\forall(s_0, s_1, ...).\ s = s_0 \Rightarrow (\forall i \geq 0.\ R, s_i \vDash \varphi_1) \vee (\exists j \geq 0.\ R, s_j \vDash \varphi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \varphi_1)}{R, s \vDash \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]}$$

Fig. 1. Semantics of $\forall$CTL: $\vDash$

```
let rec E(⟨s, ψ⟩, M, R) : bool =
  match(ψ) with
  | α → return α(s)
  | ψ′∧ψ″ →
      if (*) return E(⟨s, ψ′⟩, M, R)
      else return E(⟨s, ψ″⟩, M, R);
  | ψ′∨ψ″ →
      if (E(⟨s, ψ′⟩, M, R)) return true;
      else return E(⟨s, ψ″⟩, M, R);
  | AGψ′ →
      while (true) {
        if (¬ E(⟨s, ψ′⟩, M, R))
          return false;
        if (*) return true;
        s := choose({s′ | R(s, s′)});
      }
  | AFψ′ → local dup := false; local ′s ;
      while (true) {
        if (E(⟨s, ψ′⟩, M, R)) return true;
        if (dup ∧ ¬(∃f ∈ M. f(s) ≺ f(′s)))
          return false;
        if (¬ dup ∧ *) { dup := true; ′s := s; }
        if (*) return true;
        s := choose({s′ | R(s, s′)});
      }
  | A[ψ′Wψ″] →
      while(true) {
        if (¬E(⟨s, ψ′⟩, M, R))
          return E(⟨s, ψ″⟩, M, R);
        if (*) return true;
        s := choose({s′ | R(s, s′)});
      }
```

Fig. 2. The encoding $\mathcal{E}$ which takes a state s, a property ψ, a finite set of ranking functions $\mathcal{M}$, and a transition relation R, and constructs a recursive program which can be used to prove ψ if $\mathcal{M}$ is sufficient. choose() nondeterministically selects an element from the set given by its argument. $* \equiv \mathsf{choose}(\{\mathsf{true}, \mathsf{false}\})$

2.2 Encoding

We now show that the problem of ∀CTL verification can be reduced to a program analysis task. Our encoding $\mathcal{E}$ is given in Fig. 2. When given a transition relation system $M = (S, R, I)$ and an ∀CTL property φ, the program $\mathcal{E}$ encodes the search for the proof that φ holds of M. The arguments $(\langle s, \psi\rangle, \mathcal{M}, R)$ passed to $\mathcal{E}$ are a pair consisting of the state s, a φ-subformula ψ of interest, a finite set of ranking functions $\mathcal{M}$ and the transition relation R. Executions of the procedure $\mathcal{E}$ explore the $S \times \mathsf{sub}(\varphi)$ 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 (we discuss the termination of $\mathcal{E}$ below). 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 $\models$ is: for a transition system $M = (S, R, I)$ and ∀CTL property φ,

$$[\exists \text{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ cannot return } \mathsf{false}] \ \Rightarrow\ \forall s \in I.\ R, s \models \varphi$$

where $\mathcal{M}$ is, as described earlier, a finite set of ranking functions. We formally define "cannot return false" by giving $\mathcal{E}$ as a guarded transition system in our technical report [16], but informally it means there is no execution of $\mathcal{E}$ where false is returned.

Completeness (*i.e.* $\Leftarrow$) holds when equality over S can be determined in finite time and the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs). These results can be found in Section 3.

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 AG subformula, then $\mathcal{E}$ ensures that all possibilities are considered by establishing feasible paths to all of them. When a particular ψ is a $\vee$ or AF 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. This is the intuition behind the first invariant maintained by $\mathcal{E}$:

$$INV_1 : \forall s, \psi, \mathcal{M}, R.\ \text{ if } R, s \not\models \varphi \text{ then } \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ can return } \mathsf{false}$$

Consider this case from the definition of $\mathcal{E}$:

$$\begin{array}{rl} \mid & \psi \vee \psi' \rightarrow \mathsf{if}\ (\mathcal{E}(\langle s, \psi\rangle, \mathcal{M}, R))\ \mathsf{return\ true};\\ & \qquad\qquad \mathsf{return}\ \mathcal{E}(\langle s, \psi'\rangle, \mathcal{M}, R); \end{array}$$

Imagine that $\psi \equiv \mathsf{x} \neq 1$, and $\psi' \equiv \mathsf{AG}(\mathsf{x} = 0)$. In this case we want to know that one of the subformulae (*i.e.* $\mathsf{x} \neq 1$ or $\mathsf{AG}(\mathsf{x} = 0)$) holds. A recursive call is made with the current state s to explore whether $\mathsf{x} \neq 1$ as well as a separate recursive

call with the same current state s to explore $\mathsf{AG}(\mathsf{x} = 0)$. During a symbolic execution of this program, all executions will be considered in a search for a way to cause the program to fail. If it is possible for both recursive calls to return false (*i.e.* they abide INV_1), then there will be an execution in which the current call can return false (also abiding INV_1). A standard program analysis tool (*e.g.* SLAM [2]) will find this case. By maintaining this invariant recursively, a proof that the outermost level of $\mathcal{E}$ cannot return false implies that the outermost property holds of the original system.

Because we want to consider every state that is reachable from a finite prefix of an infinite path, it must be possible for the recursive calls to return from every state. If it were possible for the checking of a subformula like $\mathsf{AG}(\mathsf{x} = 0)$ to diverge (thus never returning false) then the above code fragment would never return false, and thus the top-level call to $\mathcal{E}$ would never return false. To this end, $\mathcal{E}$ maintains a second invariant:

$$INV_2 : \forall s, \psi, \mathcal{M}, R.\ \ \mathcal{E}(\langle s, \psi \rangle, \mathcal{M}, R) \text{ can return } \mathsf{true}$$

It is this requirement that necessitates the additional nondeterministic "if (*) return true" commands found within each loop in $\mathcal{E}$. One can think of "if (*) return true" as a form of backtracking. In our encoding, a nondeterministic return of true is not declaring that the property holds (we must *always* return true to do that). Instead, a nondeterministic return of true in the encoding means that a program analysis can freely backtrack and switch to other possible scenarios during its search for a proof.

In the AF case, our encoding must allow a program analysis to demonstrate that all paths must eventually reach a state where the subformula holds. While exploring the reachable states in R the encoding may, at any point, nondeterministically decide to capture the current state (setting dup to true and saving s as $'s$). When each subsequent state s is considered, a check is performed that there is some rank function that witnesses the well-foundedness of this particular subset of the transitive closure of the transition system (we will precisely say which subset in Section 3). This is an adaptation of a known technique [17]. However, rather than using assert to check that one of the ranking functions in $\mathcal{M}$ holds, our encoding instead returns false, allowing other possibilities to be considered (if any exist) in outer disjunctive or AF formulae.

Partial evaluation. In practice, if the input transition system is implemented as a program, then we can perform a number of static optimizations from abstract interpretation and partial evaluation that facilitates the application of current program analysis tools. Our procedure PEval implements this mixture. For lack of space, we only briefly describe these transformations. Some additional details about PEval's optimizations are provided in our technical report [16].

PEval uses the following facts: (a) the input ∀CTL formula φ is always finite (b) the structure of $\mathcal{E}$ is unchanged and (c) the program and initial state are fixed. Thus we can partially evaluate $\mathcal{E}$ on φ and the input program and obtain a first-order program for which modern program analysis techniques can be effective. For example, consider the naïve implementation of AG given in Fig. 2 which,

in essence, is interpreting the cross product of R together with the following program:

```
while true do
    if (¬ E((s, ψ'), M, R)) return false;
    if (*) return true;
done
```

Since we are considering programs as our input systems, we can build an encoding where the following fragment is instrumented in each line of a procedure based on the original input program:

```
if (¬ E((s, ψ'), M, R)) return false;
if (*) return true;
```

We will see an example of this in Section 4.

Because the program state is passed on the stack, recursive calls to $\mathcal{E}$ will not modify variables in the outer scope, and thus can be treated as `skip` statements when analyzing the iterations of R. Invariants within a given subprocedure can be vital to the pruning, simplification, and partial evaluation required to prepare the output of $\mathcal{E}$ for program analysis.

2.3 Looking for $\mathcal{M}$

Finally, recall that we must ultimately find a finite set of ranking functions $\mathcal{M}$ such that a program analysis can prove for every $s \in I$ that $\mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R)$ does not return false. Our top-level procedure adapts a known method [17] in order to iteratively find a sufficient $\mathcal{M}$. See Fig. 3. This procedure first constructs an $\mathcal{E}_\varphi$, which is a version of $\mathcal{E}$ that has been specialized on P and φ. Then, in our implementation, new ranking functions are automatically synthesized by examining counterexamples. A counterexample in ∀CTL is tree-like as follows:

```
let prove(P, φ) =
  let E_φ = PEval(E, φ, P) in
  M := ∅
  while (E_φ(M) can return false) do
      let χ be a counterexample in
      if ∃ lasso path fragment χ' from χ then
        if ∃ witness f showing χ' w.f. then
          M := M ∪ {f}
        else
          return χ
      else
          return χ
  done
  return Success
```

Fig. 3. Rank function refinement procedure where the input transition system P is assumed to be a program

$$\chi ::= \mathsf{CEX}_\alpha \text{ of } s \mid \mathsf{CEX}_\wedge \text{ of } \chi \mid \mathsf{CEX}_\vee \text{ of } \chi \times \chi$$
$$\mid \mathsf{CEX}_{\mathsf{AG}} \text{ of } \pi \times \chi \mid \mathsf{CEX}_{\mathsf{AF}} \text{ of } \pi \times \pi \times \chi \mid \mathsf{CEX}_{\mathsf{W}} \text{ of } \pi \times \chi \times \chi$$

where π is a trace through the transformed program $\mathcal{E}$. Note that often tools will not report a concrete trace but rather a *path*, *i.e.* a sequence of program counter values corresponding to a class of traces (in rare instances paths may be reported that are spurious). The counterexample structure for an atomic proposition CEX_α is simply a state in which α does not hold. Counterexamples for conjunction and disjunction are as expected. A counterexample to an AG property is a path to a place where there is a counterexample to the sub-property. A counterexample to an AF property is a "lasso"—a stem path to a particular program location, then a cycle which returns to the same program location, and a sub-counterexample along that cycle in which the sub-property does not hold. Finally, an AW counterexample is a path to a place where there is a sub-counterexample to the first property as well as a sub-counterexample to the second property.

In our encoding we obtain these tree-shaped counterexamples effectively for free with program analysis tools like SLAM that report stack-based traces for assertion failures. Information about the stack depth available in the counterexamples allows us to re-construct the tree counterexamples. That is, by walking backward over the stack trace, we can determine the tree-shape of the counterexample. Consider, for example, the case of AF. The counterexample found by the underlying tool will visit commands through the encoding of $\mathcal{E}$, including points where dup is set to true. The commands from the input program can be used to populate an instance of χ.

When a counterexample is reported that contains an instance of $\mathsf{CEX}_{\mathsf{AF}}$ (*i.e.* a "lasso fragment") it is possible that the property still holds, but that we have simply not found a sufficient ranking function to witness the termination of the lasso. In this case our procedure finds the lasso fragments and attempts to enlarge the set of ranking functions $\mathcal{M}$. One source of incompleteness of our implementation comes from our reliance on lassos: some non-terminating programs have only well-founded lassos, meaning that in these cases our refinement algorithm will fail to find useful refinements. The same problem occurs in [17]. In industrial examples these programs rarely occur.

3 Correctness

Theorem 1 (Soundness and completeness). *For a transition system* $M = (S, R, I)$ *and* $\forall CTL$ *property* φ*,*

$$[\exists \textit{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \textit{ cannot return } \mathsf{false}] \ \Rightarrow\ \forall s \in I.\ R, s \vDash \varphi$$

where $\mathcal{M}$ *is a finite set of ranking functions. Completeness (*i.e. $\Leftarrow$*) holds when equality over* S *can be determined in finite time and the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs).*

Using the Coq theorem prover we have proved the above theorem. Details can be found in the Coq proof script listed in our technical report [16]. In this section we discuss the structure of the proof and state some of the key lemmas.

$$\frac{I \subseteq \{s \mid \alpha(s)\}}{\langle R, I\rangle \vdash \alpha} \qquad \frac{\langle R, I\rangle \vdash \varphi_1 \quad \langle R, I\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \wedge \varphi_2}$$

$$\frac{\exists I_1, I_2.\ I = I_1 \cup I_2 \wedge \langle R, I_1\rangle \vdash \varphi_1 \wedge \langle R, I_2\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \vee \varphi_2}$$

$$\frac{\langle R, \{s' \mid \exists s \in I.\mathsf{reach}\ (s, s')\}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{AG}\varphi}$$

$$\frac{\exists \mathcal{F}.\mathsf{walk}_I^{\mathcal{F}} \text{ is w.f. } \wedge \langle R, \mathcal{F}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{AF}\varphi}$$

$$\frac{\exists \mathcal{F}.\forall (s, s') \in \mathsf{walk}_I^{\mathcal{F}}.\langle R, \{s\}\rangle \vdash \psi \wedge \langle R, \mathcal{F}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{A}[\psi \mathsf{W} \varphi]}$$

$$\frac{}{\mathsf{reach}_0\ (s, s)} \qquad \frac{\mathsf{reach}_n\ (s, s') \wedge R(s', s'')}{\mathsf{reach}_{n+1}\ (s, s'')}$$

$$\frac{R(s, s') \wedge s \notin \mathcal{F} \wedge s \in I}{\mathsf{walk}_I^{\mathcal{F}}\ (s, s')}$$

$$\frac{R(s', s'') \wedge s' \notin \mathcal{F} \wedge \mathsf{walk}_I^{\mathcal{F}}\ (s, s')}{\mathsf{walk}_I^{\mathcal{F}}\ (s', s'')}$$

We write $\mathsf{reach}\ (s, s')$
to mean $\exists n \geq 0.\mathsf{reach}_n\ (s, s')$.

Fig. 4. Relational formulation of ∀CTL: $\vdash$

For convenience, in the proof we introduce an alternative relational formulation of ∀CTL, $\vdash$. This formulation more closely matches our definition of $\mathcal{E}$ in that it is given over sets of states, AG is defined in terms of reachability, and AF is defined in terms of well-foundedness. In effect the encoding $\mathcal{E}$ is characterizing these sets with nondeterminism and by returning true or false. Our proof starts by showing that $\vdash$ is equivalent to $\models$ and then showing that

$$[\exists \text{ finite } \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \text{ cannot return } \mathsf{false}] \quad \Rightarrow \quad \langle R, I\rangle \vdash \varphi$$

from which point soundness directly follows.

Relational formulation of ∀CTL semantics. Our relational formulation of ∀CTL is displayed in Fig. 4. Unlike the standard formulation, ours is more amenable to reasoning about infinite-state systems because proof trees are based on *partitioning* the state space rather than *enumerating* the state space. We use the notation $\langle R, I\rangle \vdash \varphi$ to denote that a property φ is valid for a transition system. This entailment relation is then defined inductively.

An atomic proposition α involves a simple check to see if I is contained within the set of states in which α holds. The conjunction rule requires that both φ_1 and φ_2 hold of all states in I and the disjunction rule splits the states into two sets, one in which φ_1 holds and one in which φ_2 holds. The semantics of the property $\mathsf{AG}\varphi$ says that for every reachable state s', that s' entails φ.

Frontiers. The property $\mathsf{AF}\varphi$ depends on the existence of a set of states which we will call a *frontier* $\mathcal{F}$. Intuitively, the frontier $\mathcal{F}$ of a set of initial states I, is a set of states through which every trace originating at a state in I must pass.

We use frontiers in our formulation of $\mathsf{AF}\varphi$ to characterize the places where φ holds, requiring that all paths from I eventually reach a frontier. We formalize this idea by defining the inductive relation $\mathsf{walk}_I^{\mathcal{F}}$ given on the right in Fig. 4.

$\mathsf{walk}_I^{\mathcal{F}}$ is a subset of R that includes every possible transition along every trace from I up to $\mathcal{F}$. In our characterization of AF we require that $\mathsf{walk}_I^{\mathcal{F}}$ be well-founded. In this way, we recast the ∀CTL semantics of AF in terms of the well-foundedness of a relation, rather than the existence of an i-th state in every trace. This formulation allows us to more efficiently prove AF properties because we can discover well-founded relations that are over-approximations of $\mathsf{walk}_I^{\mathcal{F}}$ rather than searching for per-trace ranking functions. The final rule in the left of Fig. 4 is for the AW operator, which also uses a frontier and the relation $\mathsf{walk}_I^{\mathcal{F}}$ representing the arcs along the way to the frontier $\mathcal{F}$. To prove $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, all states along the path to the frontier must satisfy φ_1 and states at the frontier—*should one ever get there*—all must satisfy φ_2.

The following lemma shows that if a property holds in our relational semantics, then it also holds in the standard semantics of ∀CTL.

Lemma 1. *For every* φ, I, R, $\quad \langle R, I\rangle \vdash \varphi \quad \Longleftrightarrow \quad \forall s \in I.\ R, s \vDash \varphi.$

In our technical report [16] we formalize $\mathcal{E}$ as a guarded transition system. Since φ is finite, we can partially evaluate $\mathcal{E}$ with respect to φ, and represent $\mathcal{E}$ as a finite graph. The stack and return values are encoded in the configurations of the graph. Executions and the notion "cannot return false" are then defined in the natural way.

Lemma 2. *For a transition system* $M = (S, R, I)$ *and ∀CTL property* φ,

$$[\exists \mathcal{M}.\ \forall s \in I.\ \mathcal{E}(\langle s, \varphi\rangle, \mathcal{M}, R) \textit{ cannot return } \mathsf{false}] \quad \Rightarrow \quad \langle R, I\rangle \vdash \varphi.$$

*Completeness (*i.e. $\Leftarrow$*) holds when equality over* S *can be determined in finite time and each of the ranking functions are enumerable (e.g. represented as a possibly infinite list of state/rank pairs).*

Proof. By induction on φ.

From these lemmas we can prove Theorem 1.

4 Example

Consider the example in Fig. 5. After applying $\mathcal{E}$ and PEval we obtain the program given in Fig. 6. The intermediate output without partial evaluation is given in the technical report [16]. The encoding has been partially evaluated with respect to φ, and with respect to the program counter. For every $\psi \in \mathsf{sub}(\varphi)$ and $\mathtt{pc}$ valuation, there is a corresponding method $\mathtt{E}_{\text{“}\psi\text{”}}\mathtt{\_pc}$. Since we are working with a linear arithmetic program where ranking functions can be given as linear inequalities, integer $<$ is a sufficient ordering for $\prec$. The `main` procedure in the encoding initializes the program state (*i.e.* `x,n`) and then asserts that $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}\mathtt{\_0}$ cannot return false.

An execution of this program consists of a cascade of calls down the hierarchy of sub-procedures. Each procedure for a subformula maintains invariants INV_1 and INV_2. This encoding allows us to ask questions of the form "starting now (*i.e.* from this state) does there exist an execution that violates my property," and answer them using standard analysis tools.

For example, procedure $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$ corresponds to the property $\mathsf{AG}((x \neq 1) \vee \mathsf{AF}(x = 0))$ and returns false if there is a reachable state where $((x \neq 1) \vee \mathsf{AF}(x = 0))$ does not hold. It accomplishes this by calling $\mathtt{E}_{\text{“}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$ on each line and passing the current state.

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

Fig. 5. Example where $\varphi = \mathsf{AG}[(\mathtt{x} = 1) \Rightarrow \mathsf{AF}(\mathtt{x} = 0)]$ and initially $\mathtt{x} = 0$

If $((x \neq 1) \vee \mathsf{AF}(x = 0))$ does not hold from the current state, then there will be a way for $\mathtt{E}_{\text{“}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$ to return false, in which case $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$ immediately returns false (leading to an assertion failure in `main`). The procedures for disjunction ($\mathtt{E}_{\text{“}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$) and atomic propositions ($\mathtt{E}_{\text{“}x\neq 1\text{”}}$ and $\mathtt{E}_{\text{“}x=0\text{”}}$) are straight-forward following Fig. 2, and also maintain INV_1.

The procedure $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$ is, in some sense, the complement of AG. It is designed to return true whenever there is a path to a state where $x = 0$ holds, and will return false if there is an infinite execution that never reaches such a state. This is accomplished by checking at each state (*i.e.* on each line of the program) whether $\mathtt{E}_{\text{“}x=0\text{”}}$ returns true, and returning false if a location is reached multiple times and there is no ranking function in $\mathcal{M}$ that is decreasing.

With the transformation in hand, we can now apply the algorithm in Fig. 3. What remains is the task of finding a finite $\mathcal{M}$ such that in $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$ the check that $\exists f \in \mathcal{M}.\ f(\mathtt{x}_5, \mathtt{n}_5) > f(\mathtt{x}, \mathtt{n})$ always holds. Initially we let $\mathcal{M} \equiv \emptyset$. Running a refinement-based safety prover will yield a counterexample pertaining to line `lab_5` of $\mathtt{E}_{\text{“}\mathsf{AF}(x=0)\text{”}}$, where we denote a state as $\left[\begin{smallmatrix}\mathtt{x}\\ \mathtt{n}\\ \mathtt{pc}\end{smallmatrix}\right]$ and we denote transition relations as $\left[\!\!\left[\begin{smallmatrix}\mathtt{'x=x}\\ \mathtt{'n=n}\\ \mathtt{'pc=pc.}\end{smallmatrix}\right]\!\!\right]$:

$$\begin{array}{l}(\mathsf{CEX}_{\mathsf{AG}}\ \ \left(\left[\begin{smallmatrix}0\\ \mathtt{n}\\ 1\end{smallmatrix}\right] :: \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 2\end{smallmatrix}\right] :: \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 3\end{smallmatrix}\right] :: \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 4\end{smallmatrix}\right] :: \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 5\end{smallmatrix}\right]\right),\\ \quad (\mathsf{CEX}_{\vee}\ \ (\mathsf{CEX}_{\alpha}\ \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 5\end{smallmatrix}\right])\\ \qquad\qquad (\mathsf{CEX}_{\mathsf{AF}}\ \ \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 5\end{smallmatrix}\right],\ \ \left[\!\!\left[\begin{smallmatrix}\mathtt{x_5=x}\\ \mathtt{n_5=n+1}\\ \mathtt{pc_5=pc}\end{smallmatrix}\right]\!\!\right],\ \ (\mathsf{CEX}_{\alpha}\ \left[\begin{smallmatrix}1\\ \mathtt{n}\\ 5\end{smallmatrix}\right]))))\end{array}$$

In our implementation we then use a rank function synthesis tool on this counterexample (as described by Cook *et al.* [17]), find that ranking can be done on $\mathtt{n}$, and obtain a new $\mathcal{M} \equiv \{\lambda s.\ s(\mathtt{n})\}$. With this new $\mathcal{M}$ in place, $\mathtt{E}_{\text{“}\mathsf{AG}((x\neq 1)\vee\mathsf{AF}(x=0))\text{”}}$ *always* returns true, and consequently, by Theorem 1, φ holds of the original program.

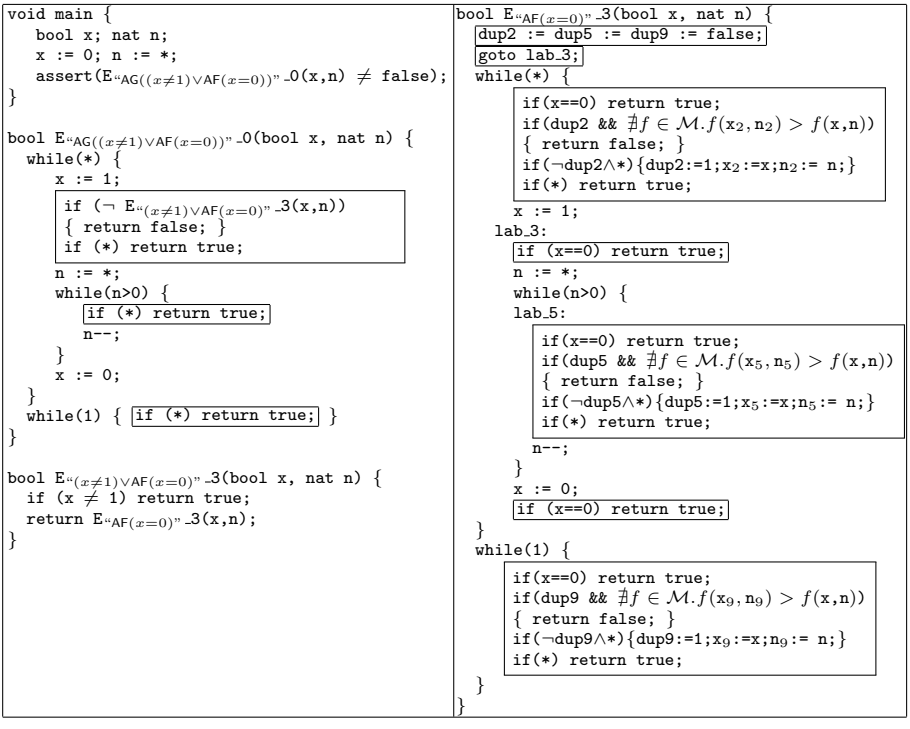

Fig. 6. The encoding $\mathcal{E}$ of property $\mathsf{AG}[(x = 1) \Rightarrow \mathsf{AF}(x = 0)]$ and the program given in Fig. 5 after PEVAL has been applied

5 Related work

There is a relationship between temporal logic verification and the problem of finding winning strategies in games or game-like structures such as alternating automata [4,25,34]. These previous results do not directly apply because they are geared toward finite state spaces. However, the technique presented in this paper can be viewed as a generalization of prior work to games over infinite state spaces. We explore this generalization in our technical report [16]. Specifically, we first show that the existence of solutions to infinite-state games (such as those used to represent the ∀CTL model checking problem) is equivalent to the existence of a solution to a mix of safety and liveness games, when those games have a certain structure. We then show that our encoding described here can be generalized to games that meet this constraint.

Other previous tools and techniques are known for proving temporal properties of finite-state systems (*e.g.* [7,11,25]) or classes of infinite-state systems with specific structure (*e.g.* pushdown systems [36,37] or parameterized systems [19]). Our proposal works for arbitrary transition systems, including programs.

A previous tool proves only trace-based (*i.e.* linear-time) properties of programs [14] using an adaptation of the traditional automata-theoretic

approach [35]. In contrast, our reduction to program analysis promotes a state-based (*e.g.* branching-time) approach. Trace-based properties can be proved with our tool using a recently described iterative symbolic determinization technique [15]. In most cases our new approach is faster for LTL verification than [14] by several orders of magnitude.

When applying traditional bottom-up based methods for state-based logics (*e.g.* [12,18,20]) to infinite-state transition systems, one important challenge is to track reachability when considering relevant subformulae from the property. In contrast to the standard method of directly tracking the valuations of subformulae in the property with additional variables, we instead use recursion to encode the checking of subformulae as a program analysis problem. As an interprocedural analysis computes procedure summaries it is in effect symbolically tracking the valuations of these subformulae depending on the context of the encoded system's state. Thus, in contrast to bottom-up techniques, ours only considers reachable states (via the underlying program analysis).

Chaki *et al.* [9] attempt to address the same problem of subformulae and reachability for infinite-state transition systems by first computing a finite abstraction of the system *a priori* that is never refined again. Then standard finite-state techniques are applied. In our approach we reverse the order: rather than applying abstraction first, we let the underlying program analysis tools perform abstraction after we have encoded the search for a proof as a new program. The approach due to Schmidt and Steffen [33] is similar.

The tool YASM [23] takes an alternative approach: it implements a refinement mechanism that examines paths which represent abstractions of tree counterexamples (using multi-valued logic). This abstraction loses information that limits the properties that YASM can prove (*e.g.* the tool will usually fail to prove $\mathsf{AFAG}p$). With our encoding the underlying tools are performing abstraction-refinement over tree counterexamples. Moreover, YASM is primarily designed to work for unnested existential properties [22] (*e.g.* $\mathsf{EF}p$ or $\mathsf{EG}p$), whereas our focus is on precise support for arbitrary (possibly nested) universal properties.

Our encoding shares some similarities with the finite-state model checking procedure CEX from Figure 6 in Clarke *et al.* [13]. The difference is that a symbolic model checking tool is used as a sub-procedure within CEX, making CEX a recursively defined model checking procedure. The finiteness of the state-space is crucial to CEX, as in the infinite-state case it would be difficult to find a finite partitioning *a priori* from which to make a finite number of model checking calls when treating temporal operators such as AG and AF.

6 Experiments

In this section we report on experiments with a prototype tool that implements $\mathcal{E}$ from Fig. 2 as well as the refinement procedure from Fig. 3. In our tool we have implemented $\mathcal{E}$ as a source-to-source translation using the CIL compiler infrastructure. We use SLAM [2] as our implementation of the safety prover, and RANKFINDER [29] as the rank function synthesis tool.

We have drawn out a set of both ∀CTL and LTL liveness property challenge problems from industrial code bases. Examples were taken from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. In order to make these examples self-contained we have, by hand, abstracted away the unnecessary functions and struct definitions. We also include a few toy examples, as well as the example from Fig. 8 in [14]. Sources of examples can be found in our technical report [16]. Heap commands from the original sources have been abstracted away using the approach due to Magill *et al.* [26]. This abstraction introduces new arithmetic variables that track the sizes of recursive predicates found as a byproduct of a successful memory safety analysis using an abstract domain based on separation logic [28]. Support for variables that range over the natural numbers is crucial for this abstraction.

As previous mentioned in Section 5, there are several available tools for verifying state-based properties of general purpose (infinite-state) programs. Neither the authors of this paper, nor the developer of YASM [23] were able to apply YASM to the challenge problems in a meaningful way, due to bugs in the tool. Note that we expect YASM would have failed in many cases [22], as it is primarily designed to work for unnested existential properties (*e.g.* $\mathsf{EG}p$ or $\mathsf{EF}p$). We have also implemented the approach due to Chaki *et al.* [9]. The difficulty with applying this approach to the challenge problems is that the programs must first be abstracted to finite-state before branching-time proof methods are applied. Because the challenge problems focus on liveness, we have used transition predicate abstraction [31] as the abstraction method. However, because abstraction must happen first, predicates must be chosen ahead of time either by hand or using heuristics. In practice we found that our heuristics for choosing an abstraction *a priori* could not be easily tuned to lead to useful results.

Because the examples are infinite-state systems, popular CTL-proving tools such as Cadence SMV [1] or NUSMV [10] are not directly applicable. When applied to finite instantiations of the programs these tools run out of memory.

The tool described in Cook *et al.* [14] can be used to prove LTL properties if used in combination with an LTL to Büchi automata conversion tool (*e.g.* [21]). To compare our approach to this tool we have used two sets of experiments: Fig. 7 displays the results on challenge problems in ∀CTL verification; Fig. 8 contains results on LTL verification. Experiments were run using Windows Vista and an Intel 2.66GHz processor.

In both figures, the code example is given in the first column, and a note as to whether it contains a bug. We also give a count of the lines of code and the shape of the temporal property where p and q are atomic propositions specific to the program. For both the tools we report the total time (in seconds) and the result for each of the benchmarks. A ✓ indicates that a tool proved the property, and χ is used to denote cases where bugs were found (and a counterexample returned). In the case that a tool exceeded the timeout threshold of 4 hours, ">14400.00" is used to represent the time, and the result is listed as "**???**".

Program	LOC	Property	Prev. tool [14] Time	Result	Our tool (Sec. 2) Time	Result
Acq/rel	14	$\mathsf{AG}(a \Rightarrow \mathsf{AF}b)$	103.48	✓	14.18	✓
Ex from Fig. 8 of [14]	34	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	209.64	✓	27.94	✓
Toy linear arith. 1	13	$p \Rightarrow \mathsf{AF}q$	126.86	✓	34.51	✓
Toy linear arith. 2	13	$p \Rightarrow \mathsf{AF}q$	>14400.00	**???**	6.74	✓
PostgreSQL smsrv	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	>14400.00	**???**	9.56	✓
PostgreSQL smsrv+bug	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	87.31	χ	47.16	χ
PostgreSQL pgarch	61	$\mathsf{AFAG}p$	31.50	✓	15.20	✓
Apache progress	314	$\mathsf{AG}(p \Rightarrow (\mathsf{AF} \vee \mathsf{AF}))$	685.34	✓	684.24	✓
Windows OS 1	180	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	901.81	✓	539.00	✓
Windows OS 4	327	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	**???**	1,114.18	✓
Windows OS 4	327	$(\mathsf{AF}a) \vee (\mathsf{AF}b)$	1,223.96	✓	100.68	✓
Windows OS 5	648	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	**???**	>14400.00	**???**
Windows OS 7	13	$\mathsf{AGAF}p$	>14400.00	**???**	55.77	✓

Fig. 7. Comparison between our tool and Cook *et al.* [14] on ∀CTL verification benchmarks. All of the above ∀CTL properties have equivalent corresponding LTL properties so they are suitable for direct comparison with the LTL tool [14].

Program	LOC	Property	Prev. tool [14] Time	Result	Our tool (Sec. 2) Time	#	Result
Ex. from [15]	5	$\mathsf{FG}p$	2.32	✓	1.98	2	✓
PostgreSQL dropbuf	152	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	53.99	✓	27.54	3	✓
Apache `accept` liveness	314	$\mathsf{G}p \Rightarrow \mathsf{GF}q$	>14400.00	**???**	197.41	3	✓
Windows OS 2	158	$\mathsf{FG}p$	16.47	✓	52.10	4	✓
Windows OS 2+bug	158	$\mathsf{FG}p$	26.15	χ	30.37	1	χ
Windows OS 3	14	$\mathsf{FG}p$	4.21	✓	15.75	2	✓
Windows OS 6	13	$\mathsf{FG}p$	149.41	✓	59.56	1	✓
Windows OS 6+bug	13	$\mathsf{FG}p$	6.06	χ	22.12	1	χ
Windows OS 8	181	$\mathsf{FG}p$	>14400.00	**???**	5.24	1	✓

Fig. 8. Comparison between our tool and Cook *et al.* [14] on LTL benchmarks. For our tool, we use a recently described iterative symbolic determinization strategy [15] to prove LTL properties by using Fig. 3 as the underlying ∀CTL proof technique. The number of iterations is reported in the # column.

When comparing approaches on ∀CTL properties (Fig. 7) we have chosen properties that are equivalent in ∀CTL and LTL and then directly compared our procedure from Fig. 3 to the tool in Cook *et al.* [14]. When comparing approaches on LTL verification problems (Fig. 8) we have used an iterative symbolic determinization strategy [15] which calls our procedure in Fig. 3 on successively refined ∀CTL verification problems. The number of such iterations is given as column "#." in Fig. 8. For example, in the case of benchmark Windows OS 3, our procedure was called twice while attempting to prove a property of the form $\mathsf{FG}p$.

Our technique was able to prove or disprove all but one example, usually in a fraction of a minute. The competing tool fails on over 25% of the benchmarks.

7 Conclusions

We have introduced a novel temporal reasoning technique for (potentially infinite-state) transition systems, with an implementation designed for systems described as programs. Our approach shifts the task of temporal reasoning to a program analysis problem. When an analysis is performed on the output of our encoding, it is effectively reasoning about the temporal and possibly branching behaviors of the original system. Consequently, we can use the wide variety of efficient program analysis tools to prove properties of programs. We have demonstrated the practical viability of the approach using industrial code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Acknowledgments. We would like to thank Josh Berdine, Michael Greenberg, Daniel Kroening, Axel Legay, Rupak Majumdar, Peter O'Hearn, Joel Ouaknine, Nir Piterman, Andreas Podelski, Noam Rinetzky, and Hongseok Yang for valuable discussions regarding this work. We also thank the Gates Cambridge Trust for funding Eric Koskinen's Ph.D. degree program.

References

1. Cadence SMV, `http://www.kenmcmil.com/smv.html`
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys, pp. 73–85 (2006)
3. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: POPL, pp. 211–224 (2007)
4. Bernholtz, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking (extended abstract). In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 142–155. Springer, Heidelberg (1994)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207 (2003)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. Automata, Languages and Programming, 1349–1361 (2005)
7. Burch, J., Clarke, E., et al.: Symbolic model checking: 10^{20} states and beyond. Information and computation 98(2), 142–170 (1992)
8. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
9. 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* (2005), pp. 53–69.
10. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An openSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 359. Springer, Heidelberg (2002)
11. Clarke, E., Emerson, E.,, S.: Automatic verification of finite-state concurrent systems using temporal logic specifications. TOPLAS 8(2), 263 (1986)
12. Clarke, E., Grumberg, O., Peled, D.: Model checking (1999)
13. Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29 (2002)

14. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL, pp. 265–276 (2007)
15. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: POPL, pp. 399–410 (2011)
16. Cook, B., Koskinen, E., Vardi, M.: Branching-time reasoning for programs. Tech. Rep. UCAM-CL-TR-788, University of Cambridge, Computer Laboratory (January 2011), http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-788.html
17. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
18. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
19. Emerson, E., Namjoshi, K.: Automatic verification of parameterized synchronous systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, Springer, Heidelberg (1996)
20. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Program specialization for verifying infinite state systems: An experimental evaluation. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 164–183. Springer, Heidelberg (2011)
21. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 53. Springer, Heidelberg (2001)
22. Gurfinkel, A.: Personal communication (2010)
23. Gurfinkel, A., Wei, O., Chechik, M.: YASM: A software model-checker for verification and refutation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006)
24. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 526. Springer, Heidelberg (2002)
25. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM 47(2), 312–360 (2000)
26. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
27. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
28. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
29. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
30. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
31. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL (2005)
32. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61 (1995)
33. Schmidt, D.A., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
34. Stirling, C.: Games and modal mu-calculus. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 298–312. Springer, Heidelberg (1996)
35. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop, pp. 238–266 (1995)
36. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
37. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: FST TCS, pp. 127–138 (2000)