Form Methods Syst Des (2012) 41:66–82
DOI 10.1007/s10703-012-0153-5

Temporal property verification as a program analysis task

Extended Version

Byron Cook · Eric Koskinen · Moshe Vardi

Published online: 27 April 2012

Abstract We describe a reduction from temporal property verification to a program analysis problem. First we present a proof system that, unlike the standard formulation, is more amenable to reasoning about infinite-state systems: disjunction is treated by partitioning, rather than enumerating, the state space and temporal operators are characterized with special sets of states called frontiers. We then describe a transformation that, with the use of procedures 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.

Keywords Automatic software verification · Program analysis · Temporal logic · Model checking · Termination · Formal verification

1 Introduction

In this paper, we describe a method of proving temporal properties of (possibly infinite-state) transition systems. Our method is designed to incorporate modern analysis techniques (abstraction refinement, interpolation, termination argument refinement, and so forth). Consequently, we obtain a tool that is more effective than previous efforts at reasoning about the validity of temporal properties of infinite-state systems.

B. Cook
Microsoft Research and University College London, London, UK
e-mail: bycook@microsoft.com

E. Koskinen (✉)
New York University, New York, NY, USA
e-mail: ejk@cims.nyu.edu

M. Vardi
Rice University, Houston, TX, USA
e-mail: vardi@cs.rice.edu

We begin with a novel proof system for temporal verification (Sect. 3). Disjunction is based on *partitioning* the state space rather than *enumerating* the state space. Temporal operators use sets of states called *frontiers* to characterize those states in which subformulae hold. Thus, since individual states are not mentioned, we can often find finite derivations in our proof system, despite infinite state spaces. Algorithms and tools can often find finitely representable over-approximations for the sets of states in a derivation.

Second, we observe that, with subtle use of procedures and nondeterminism, temporal reasoning can be encoded as a program analysis problem (Sect. 4). Specifically, we decompose verification into a *safety* problem in tandem with a search for sufficient termination arguments to solve a *liveness* 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, 6, 9, 32]) together with techniques for discovering termination arguments (*e.g.* [3, 7, 18]), we can implement temporal logic provers whose power is effectively only limited by the power of the underlying tools. An output solution to the tandem problems gives us (a symbolic representation of) the states, frontiers, and termination arguments that comprise a derivation in our proof system.

Based on the above 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 (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [16].

This paper is an elaboration of the work we presented last year [17]. Full formal detail (including scripts for the Coq proof assistant) can be found in Koskinen's dissertation [25].

Limitations We have shown that our technique applies to infinite-state transition systems. In practice, however, our approach has only been applied to sequential non-recursive programs. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL) as it is sufficient for verifying LTL properties when used with our iterated symbolic determinization [16]. Existential reasoning is left for future work.

2 Example

Consider the following program P where we are interested in proving the property $\Phi = \mathsf{AG}[\mathtt{x} \Rightarrow \mathsf{AF}\ \neg\mathtt{x}]$.

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

This is a standard acquire/release style property, letting us prove properties such as "whenever a lock is acquired, it is eventually released." Note that * represents nondeterministic choice.

In this paper we describe a reduction from temporal property verification to a program analysis task. Specifically, we perform a transformation from an input program P and property Φ to a new program P' that is parameterized by a finite set of ranking functions $\mathcal{M}$. This leads us to two tandem tasks:

1. A search for a sufficient finite set $\mathcal{M}$ of ranking functions, such that
2. $P'(\mathcal{M})$ can be proved to be safe.

If we can, indeed, find a finite set of ranking functions $\mathcal{M}$ such that P' can be proved to be safe then the property Φ holds of P (*i.e.* $P \models \Phi$). In this way, we can leverage sophisticated abstraction techniques for termination and safety from modern program analysis tools in order to achieve temporal verification.

Our technique is state-based in nature, which we observe to be typically more efficient than trace-based techniques. As such, the properties are expressed in the universal fragment of Computation Tree Logic (∀CTL). This, however, does not preclude trace-based logics. We can combine the work here with our previously described iterated symbolic determinization [16] in order to prove trace-based properties, such as those expressed in Linear Temporal Logic (LTL).

For the above example, our technique will produce the output program P' given in Fig. 1, explained later. Notice that this program P' has a *safety* assertion (in `main`) and is parameterized by *termination* arguments $\mathcal{M}$ (in $\texttt{enc}^{\mathsf{RL}_\epsilon}_{\mathsf{AF}_{\neg \mathtt{x}}}$). In this case, if we let

$$\mathcal{M} \equiv \left\{ \left(\lambda \begin{bmatrix} \mathtt{x} \\ \mathtt{n} \\ \mathsf{pc} \end{bmatrix} . \mathtt{n} \right) \right\}$$

then a standard program analysis tool can prove P' to be safe, and so the property must hold of P.

The rest of this paper is organized as follows. In Sect. 3 we review temporal logic definitions and describe a novel proof system for ∀CTL that serves as the intuition and correctness argument for our technique. In Sect. 4 we describe our core verification technique. We discuss some related work in Sect. 5 and report experimental results in Sect. 6.

3 Frontiers for ∀CTL verification

We now describe a novel proof system for ∀CTL that is geared toward infinite state spaces. As we will see, disjunction is treated by partitioning (rather than enumerating) the state space, and temporal operators are based on the existence of sets of states called *frontiers*. Thus, since individual states are not mentioned, algorithms and tools can often find (finitely representable) over-approximations of sets of states. The proof system presented in this section serves as the intuition and correctness argument for our verification technique.

States, transition systems We assume nothing about the set of states S, except that state equality is decidable: for every $s, s' \in S$, either $s = s'$ or $s \neq s'$ and that this can be determined in finite time. 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 an

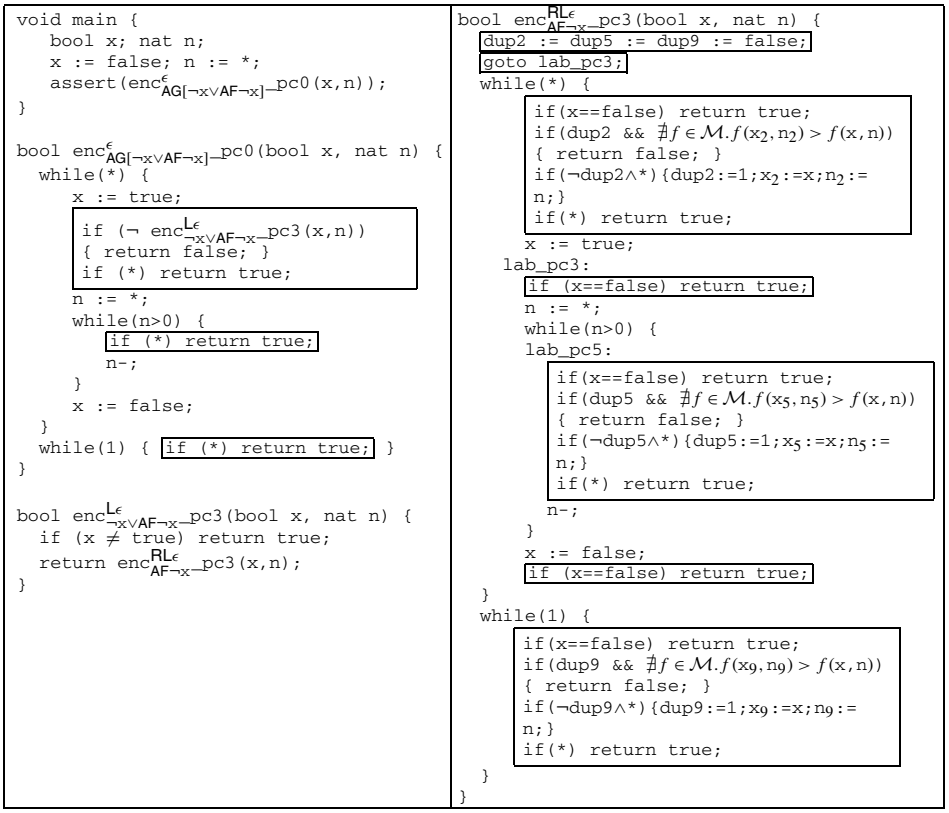

Fig. 1 The encoding $\mathcal{E}$ for property $\mathsf{AG}[x \Rightarrow \mathsf{AF}\ \neg x]$ and the program from Sect. 2. This simplified version of the encoding is derived from the full encoding given in Fig. 3 via partial evaluation. Some procedures are inlined, some are specialized w.r.t. the program counter, and unreachable/inconsequential code is eliminated

infinite sequence of states $(s_0, s_1, \ldots)$ 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 loss of generality, as final states can be encoded as states that loop back to themselves. We use $[\![\alpha]\!]^{\mathsf{S}}$ to denote the set of states for which atomic proposition α holds.

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 $\{f_1, \ldots, f_n\}$ can denote the disjunctively well-founded relation:

$$\{(s, s') \mid f_1(s') \prec f_1(s) \vee \cdots \vee f_n(s') \prec f_n(s)\}.$$

Temporal logic The technique describe in this paper is state-based in nature and, as such, is readily suitable to ∀CTL properties. To prove LTL properties we use our recently described iterative symbolic determinization technique [16] with the ∀CTL proving technique

described here. The syntax of ∀CTL is:

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

and the semantics are as follows:

$$\frac{s \in [\![\alpha]\!]^{\mathsf{S}}}{R, s \vDash \alpha} \quad \frac{R, s \vDash \Phi_1 R, s \vDash \Phi_2}{R, s \vDash \Phi_1 \wedge \Phi_2} \quad \frac{R, s \vDash \Phi_1 \vee R, s \vDash \Phi_2}{R, s \vDash \Phi_1 \vee \Phi_2}$$

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

$$\frac{\begin{array}{c}\forall (s_0, s_1, \ldots).s = s_0 \Rightarrow (\forall i \geq 0.\ R, s_i \vDash \Phi_1) \vee \\ (\exists j \geq 0.\ R, s_j \vDash \Phi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \Phi_1)\end{array}}{R, s \vDash \mathsf{A}[\Phi_1\,\mathsf{W}\,\Phi_2]}$$

∀CTL's temporal operators are state-based in structure. The operator $\mathsf{AF}\Phi$ specifies that, across all computation sequences from the current state, a state in which Φ holds must be reached. The $\mathsf{A}[\Phi_1\,\mathsf{W}\,\Phi_2]$ operator specifies that Φ_1 holds in every state where Φ_2 does not yet hold.

We use AF and 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 and AW to safety. The AG operator can be expressed with AW, and we omit the next state operator AX as it is not usually useful in the context of programs. 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 normalized.

We will need to enumerate ∀CTL subformulae, taking care to uniquely identify each one. 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 Φ of interest, at each step taking either the left or right subformula ($\mathsf{L}\kappa$ or $\mathsf{R}\kappa$). For an ∀CTL property Φ, the set of subformulae is a set of (κ, Φ) pairs as follows:

$$\begin{aligned}
\mathsf{sub}(\Phi) &\equiv \mathsf{sub}(\epsilon, \Phi) \\
\mathsf{sub}(\kappa, \alpha) &\equiv \{(\kappa, \alpha)\} \\
\mathsf{sub}(\kappa, \Phi \vee \Phi') &\equiv \{(\kappa, \Phi \vee \Phi')\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi') \\
\mathsf{sub}(\kappa, \Phi \wedge \Phi') &\equiv \{(\kappa, \Phi \wedge \Phi')\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi') \\
\mathsf{sub}(\kappa, \mathsf{AF}\Phi) &\equiv \{(\kappa, \mathsf{AF}\Phi)\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \\
\mathsf{sub}(\kappa, \mathsf{A}[\Phi\,\mathsf{W}\,\Phi']) &\equiv \{(\kappa, \mathsf{A}[\Phi\,\mathsf{W}\,\Phi'])\} \cup \mathsf{sub}(\mathsf{L}\kappa, \Phi) \cup \mathsf{sub}(\mathsf{R}\kappa, \Phi')
\end{aligned}$$

Our proof system for ∀CTL relates formulae with *sets-of-states* rather than individual states, and is defined as follows:

Definition 1 (Proof system for ∀CTL)

$$\frac{I \subseteq [\![\alpha]\!]^{\mathsf{S}}}{\langle R, I\rangle \vdash \alpha} \quad \frac{\langle R, I\rangle \vdash \Phi_1 \langle R, I\rangle \vdash \Phi_2}{\langle R, I\rangle \vdash \Phi_1 \wedge \Phi_2} \quad \frac{I = I_1 \cup I_2 \langle R, I_1\rangle \vdash \Phi_1 \langle R, I_2\rangle \vdash \Phi_2}{\langle R, I\rangle \vdash \Phi_1 \vee \Phi_2}$$

$$\frac{\mathsf{walk}_I^{\mathcal{F}} \text{ is well-founded } \langle R, \mathcal{F}\rangle \vdash \Phi}{\langle R, I\rangle \vdash \mathsf{AF}\Phi} \quad \frac{\langle R, (\mathsf{walk}_I^{\mathcal{F}})|_1\rangle \vdash \Phi_1 \langle R, \mathcal{F}\rangle \vdash \Phi_2}{\langle R, I\rangle \vdash \mathsf{A}[\Phi_1\,\mathsf{W}\,\Phi_2]}$$

where walk is as follows:

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

The notation $\langle R, I\rangle \vdash \Phi$ denotes that a property Φ is valid for a set of states I. The 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. Disjunction is treated by *partitioning* the state space rather than *enumerating* the state space: the states are split into two sets, one in which Φ_1 holds and one in which Φ_2 holds. As we will see, this treatment is more amenable to reasoning about infinite-state systems.

The remaining rules involve the existence of a set of states called 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. In $\mathsf{AF}\Phi$, frontier $\mathcal{F}$ characterizes the places where Φ holds, requiring that all paths from I eventually reach a state in $\mathcal{F}$. We formalize this idea by defining the inductive relation $\mathsf{walk}_I^{\mathcal{F}}$ given in Definition 1. $\mathsf{walk}_I^{\mathcal{F}}$ is the subset of R that includes every possible transition along every trace from I up to $\mathcal{F}$. In our treatment 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 efficiently prove AF properties because we can discover well-founded relations that are over-approximations of $\mathsf{walk}_I^{\mathcal{F}}$. The final rule in Definition 1 is for the A W 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 $\mathsf{A}[\Phi_1 \,\mathsf{W}\, \Phi_2]$, 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.

In the following lemma we show that if a property holds in our relational semantics, then it also holds in the standard semantics of ∀CTL and vice-versa.

Lemma 1 *For every* $\Phi, I, R, \langle R, I\rangle \vdash \Phi \iff \forall s \in I. R, s \vDash \Phi$.

Example The aim of this paper is an automatic method for proving temporal logic properties of programs. Here we will give an example manual derivation for the program in Sect. 2. This highlights the fact that our proof system from Definition 1 allows for finite derivations when state-spaces may be infinite, but expert readers may choose to skip onward to Sect. 4. For simplicity, we can describe the program as the transition system $M = (S, R, I)$:

$$S = \mathbf{B} \times \mathbf{N} \times \{\ell_0, \ell_1\} \quad \text{where } s \in S \text{ is denoted } \begin{bmatrix} \mathtt{x} \\ \mathtt{n} \\ \mathtt{pc} \end{bmatrix},$$

$$R = \left\{ \left(\begin{bmatrix} \mathsf{F} \\ 0 \\ \ell_0 \end{bmatrix}, \begin{bmatrix} \mathsf{T} \\ \mathtt{n} \\ \ell_1 \end{bmatrix} \right) \middle| \mathtt{n} > 0 \right\} \cup \left\{ \left(\begin{bmatrix} \mathsf{T} \\ \mathtt{n} \\ \ell_1 \end{bmatrix}, \begin{bmatrix} \mathsf{T} \\ \mathtt{n\text{-}1} \\ \ell_1 \end{bmatrix} \right) \middle| \mathtt{n} > 0 \right\}$$
$$\cup \left\{ \left(\begin{bmatrix} \mathsf{T} \\ 0 \\ \ell_1 \end{bmatrix}, \begin{bmatrix} \mathsf{F} \\ 0 \\ \ell_0 \end{bmatrix} \right) \right\},$$

$$I = \left\{ \begin{bmatrix} \mathsf{F} \\ 0 \\ \ell_0 \end{bmatrix} \right\}.$$

where $\mathsf{T} = \mathsf{true}$ and $\mathsf{F} = \mathsf{false}$. We can construct the following derivation in our proof system, in order to show that $\langle R, I\rangle \vdash \mathsf{AG}[\mathtt{x} \Rightarrow \mathsf{AF}\ \neg\mathtt{x}]$:

$$\dfrac{\dfrac{X = (\mathsf{walk}_I^{\mathcal{F}_1})|_1 \quad \dfrac{X \subseteq [\![\neg\mathtt{x}]\!]^{\mathsf{S}}}{\langle R, X\rangle \vdash \neg\mathtt{x}} \quad \dfrac{\mathsf{walk}_Y^{\mathcal{F}_2} \text{ is w.f.} \quad \dfrac{\mathcal{F}_2 \subseteq [\![\neg\mathtt{x}]\!]^{\mathsf{S}}}{\langle R, \mathcal{F}_2\rangle \vdash \neg\mathtt{x}}}{\langle R, Y\rangle \vdash \mathsf{AF}\ \neg\mathtt{x}}}{\langle R, (\mathsf{walk}_I^{\mathcal{F}_1})|_1\rangle \vdash (\mathsf{AF}\ \neg\mathtt{x} \vee \neg\mathtt{x})} \quad \dfrac{\mathcal{F}_1 \subseteq [\![\mathsf{false}]\!]^{\mathsf{S}}}{\langle R, \mathcal{F}_1\rangle \vdash \mathsf{false}}}{\langle R, I\rangle \vdash \mathsf{A}[(\mathsf{AF}\ \neg\mathtt{x} \vee \neg\mathtt{x})\ \mathsf{W}\ \mathsf{false}]}$$

where $\mathcal{F}_1 \equiv \emptyset$ $\quad X \equiv \mathbf{B} \times \mathbf{N} \times \{\ell_0\}$

$\mathcal{F}_2 \equiv \mathbf{B} \times \mathbf{N} \times \{\ell_0\}$ $\quad Y \equiv \mathbf{B} \times \mathbf{N} \times \{\ell_1\}$.

What remains are five proof obligations, proved below. Notice that this derivation has finitely many inference rules. This stands in contrast to the standard formulation of ∀CTL, which would have required us to enumerate states and traces or find a finite abstraction *a priori*. Moreover, since elements of sets of states $I, \mathcal{F}_1, \mathcal{F}_2, X, Y$ and elements of relations $\mathsf{walk}_I^{\mathcal{F}_1}, \mathsf{walk}_Y^{\mathcal{F}_2}$ are not mentioned, algorithms and tools can discover finite over-approximations of them (*e.g.* $[\![\mathtt{x} = \mathsf{false}]\!]^{\mathsf{S}}$ or $\mathsf{walk}_Y^{\mathcal{F}_2} \subseteq [\![\mathtt{n}' < \mathtt{n} \wedge \mathtt{n} > 0]\!]^{\mathsf{R}}$).

The obligations and proofs are as follows: (1) $X \cup Y = (\mathsf{walk}_{\mathcal{F}_1}^{I})|_1$. *Since $\mathcal{F}_1 = \emptyset$, the RHS is the set of all (reachable) states. The LHS is the set of all states. In this example the two are equivalent.* (2) $X \subseteq [\![\neg\mathtt{x}]\!]^{\mathsf{S}}$. *Initially when* $\mathsf{pc} = \ell_0$ *then* $\mathtt{x} = \mathsf{false}$. *Moreover,* $\mathtt{x}$ *only becomes* true *when control changes to* ℓ_1, *and then* $\mathtt{x}$ *becomes* false *again whenever control changes back to* ℓ_0. (3) $\mathsf{walk}_Y^{\mathcal{F}_2}$ is well-founded. *Substituting, we must show that* $\mathsf{walk}_{[\![\mathsf{pc}=\ell_1]\!]^S}^{[\![\mathsf{pc}=\ell_0]\!]^S}$ *is well-founded. If we unroll the definition of* walk *we see that* $\mathsf{walk}_{[\![\mathsf{pc}=\ell_1]\!]^S}^{[\![\mathsf{pc}=\ell_0]\!]^S}$ *is the set of all state transitions from* ℓ_1, *returning to* ℓ_1. *This relation is well-founded because there is a ranking function:*

$$f \equiv \lambda \begin{bmatrix} x \\ n \\ \mathsf{pc} \end{bmatrix}. n$$

where the well-order is simply the natural numbers. (4) $\mathcal{F}_2 \subseteq [\![\neg\mathtt{x}]\!]^{\mathsf{S}}$. *Same as 2 above.* (5) $\mathcal{F}_1 \subseteq [\![\mathsf{false}]\!]^{\mathsf{S}}$. *Trivial.*

4 ∀CTL verification as safety and liveness

We characterize ∀CTL verification as a *safety* problem in tandem with a search for sufficient termination arguments to solve a *liveness* problem. An output solution to the tandem problems gives us (typically, a symbolic representation of) the states, frontiers, and termination arguments that comprise a derivation in the proof system in Definition 1. A formal description and proof of correctness for our approach is given in Koskinen's dissertation [25]. Here we present only the specialization as a program analysis problem.

Encoding Our encoding, given in Fig. 2, allows modern program analysis tools (abstraction refinement, interpolation, termination argument refinement, and so forth) to perform what is necessary to validate a temporal logic property Φ for a program P. Our encoding

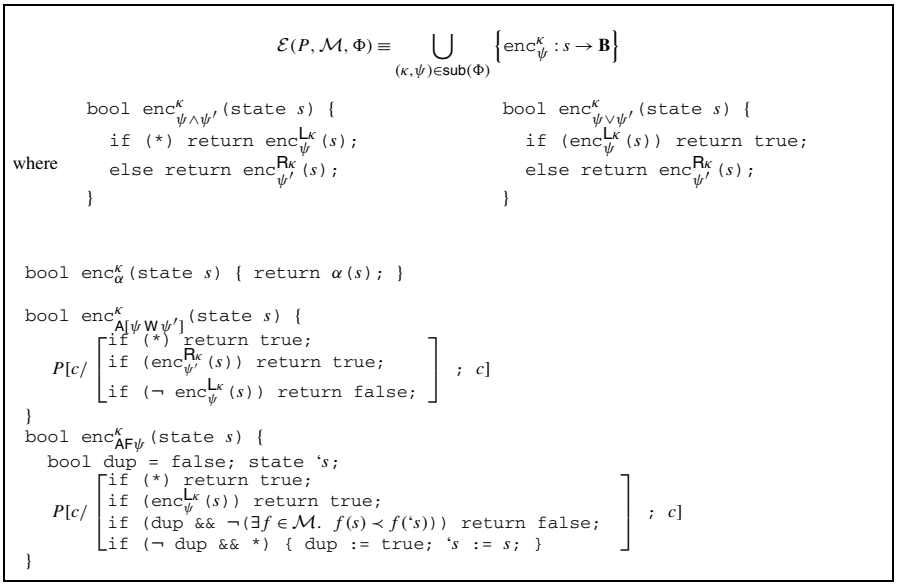

Fig. 2 Encoding ∀CTL verification of program P and property Φ as a program analysis task over a finite set of procedures. We use the notation $P[c/d]$ to mean that each command c in program P is replaced with a new code fragment d

generates a finite number of procedures in a C-like language, one for each subformula. For the example in Sect. 2, the encoding consists of the following set of procedures:

$$\{\ \mathtt{enc}^{\epsilon}_{\mathsf{AG}[\neg\mathsf{x}\vee\mathsf{AF}\neg\mathsf{x}]},\quad \mathtt{enc}^{\mathsf{L}\epsilon}_{[\neg\mathsf{x}\vee\mathsf{AF}\neg\mathsf{x}]},\quad \mathtt{enc}^{\mathsf{LL}\epsilon}_{\neg\mathsf{x}},\quad \mathtt{enc}^{\mathsf{RL}\epsilon}_{\mathsf{AF}\neg\mathsf{x}},\quad \mathtt{enc}^{\mathsf{LRL}\epsilon}_{\neg\mathsf{x}}\ \}$$

The per-subformula encoding is given in Fig. 2. Notice that this encoding is parameterized by a finite set of rank functions $\mathcal{M}$, which is used in the $\mathtt{enc}^{\kappa}_{\mathsf{AF}\psi}$ case. These procedures encode the search for the proof that Φ holds of P: if a sufficient $\mathcal{M}$ is found such that $\mathtt{assert}(\mathtt{enc}^{\epsilon}_{\Phi}(s))$ can be proved safe for all $s \in I$, then Φ holds of P (*i.e.* $P \vDash \Phi$). This is given by the following theorem:

Theorem 1 ($\mathcal{E}$ soundness [25]) *For a program P and ∀CTL property Φ,*

$$\exists\, \textit{finite}\ \mathcal{M}.\ \mathcal{E}(P, \mathcal{M}, \Phi)\ \textit{cannot return}\ \mathsf{false} \quad \Rightarrow \quad P \vDash \Phi.$$

We abuse notation slightly here, using $\mathcal{E}(P, \mathcal{M}, \Phi)$ to mean "$\forall s \in I.\mathtt{enc}^{\epsilon}_{\Phi}(s)$".

Each procedure $\mathtt{enc}^{\kappa}_{\psi} \in \mathcal{E}$ is designed to determine whether a subformula κ, ψ holds of a state s. By passing the state on the stack, we consider multiple branching scenarios. When a particular ψ is a $\wedge$ or AW subformula, then $\mathtt{enc}^{\kappa}_{\psi}$ 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. If $\mathtt{enc}^{\epsilon}_{\Phi}$ can be proved to never return false, then it must be the case that the overall property Φ holds of the initial state s.

Atomic proposition In the atomic proposition case, $\texttt{enc}^{\kappa}_{\alpha}$ involves a simple check to see whether the atomic proposition α holds of the current state.

Conjunction For a subformula $\psi \wedge \psi'$, the encoding establishes two feasible paths: one to $\texttt{enc}^{\mathsf{L}\kappa}_{\psi}$ and one to $\texttt{enc}^{\mathsf{R}\kappa}_{\psi'}$, passing s in each case. If, for example, ψ does not hold of s, then there will be a way for $\texttt{enc}^{\mathsf{L}\kappa}_{\psi}$ to return false. Consequently, there will be a way for $\texttt{enc}^{\kappa}_{\psi\wedge\psi'}$ to return false and we can conclude that the property does not hold. Alternatively, if neither subformula procedure call could possibly return false, then $\texttt{enc}^{\kappa}_{\psi\wedge\psi'}$ cannot return false and we can conclude that the property holds.

Our encoding takes advantage of the fact that program analysis tools for safety are effective at discovering a counterexample *execution* that leads to an assertion violation. To this end, $\mathcal{E}$ maintains the following invariant:

$$INV_1 : \forall s, \psi, \kappa.R, s \not\models \psi \text{ implies } \texttt{enc}^{\kappa}_{\psi}(s) \text{ can return false}$$

Disjunction Consider $\texttt{enc}^{\kappa}_{\psi\vee\psi'}$ and imagine that $\psi \equiv p$, and $\psi' \equiv \mathsf{AF}q$. In this case we want to know that one of the subformulae (*i.e.* p or $\mathsf{AF}q$) holds. A procedure call $\texttt{enc}^{\mathsf{L}\kappa}_{p}(s)$ is made to explore whether p holds as well as a separate procedure call $\texttt{enc}^{\mathsf{R}\kappa}_{\mathsf{AF}q}(s)$ with the same current state s to explore $\mathsf{AF}q$. 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 procedure calls to return false (*i.e.* they satisfy INV_1), then there will be an execution in which $\texttt{enc}^{\kappa}_{p\vee\mathsf{AF}q}(s)$ can return false (also satisfying INV_1). A standard program analysis tool (*e.g.* SLAM [2] or BLAST [5]) will find this case. By maintaining this invariant in each procedure, a proof that the outermost procedure $\texttt{enc}^{\epsilon}_{\Phi}$ cannot return false implies that the property Φ holds of the program P.

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

$$INV_2 : \forall s, \psi, \kappa.\texttt{enc}^{\kappa}_{\psi}(s) \text{ can return true}$$

It is this requirement that necessitates the additional nondeterministic "`if (*) return true`" commands found within $\texttt{enc}^{\kappa}_{\mathsf{A}[\psi\,\mathsf{W}\,\psi']}$ and $\texttt{enc}^{\kappa}_{\mathsf{AF}\psi}$. 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.

Sequencing For $\mathsf{A}[\psi\,\mathsf{W}\,\psi']$, the encoding $\mathcal{E}$ ensures that, along every path from s, as long as ψ' does not hold yet (it may never hold), ψ still holds. We use the notation $P[c/d]$ to mean, informally, that each command c in program P is replaced with a new code fragment d. In this way, the encoding continually steps through the transition relation by executing each subsequent command c, performing this check at each command. Command replacement requires that we take care to treat the program counter correctly. In Fig. 3 this is accomplished with a simple `goto` case split, and Fig. 1 accomplishes this by specializing procedures with respect to the program counter (as described below).

If $\mathsf{A}[\psi\,\mathsf{W}\,\psi']$ does not hold, then there will be a finite sequence of states $s, s_1, \ldots, s_n$ such that ψ holds at each state s_i $(i < n)$ and neither ψ nor ψ' holds at s_n. If this is the case, there

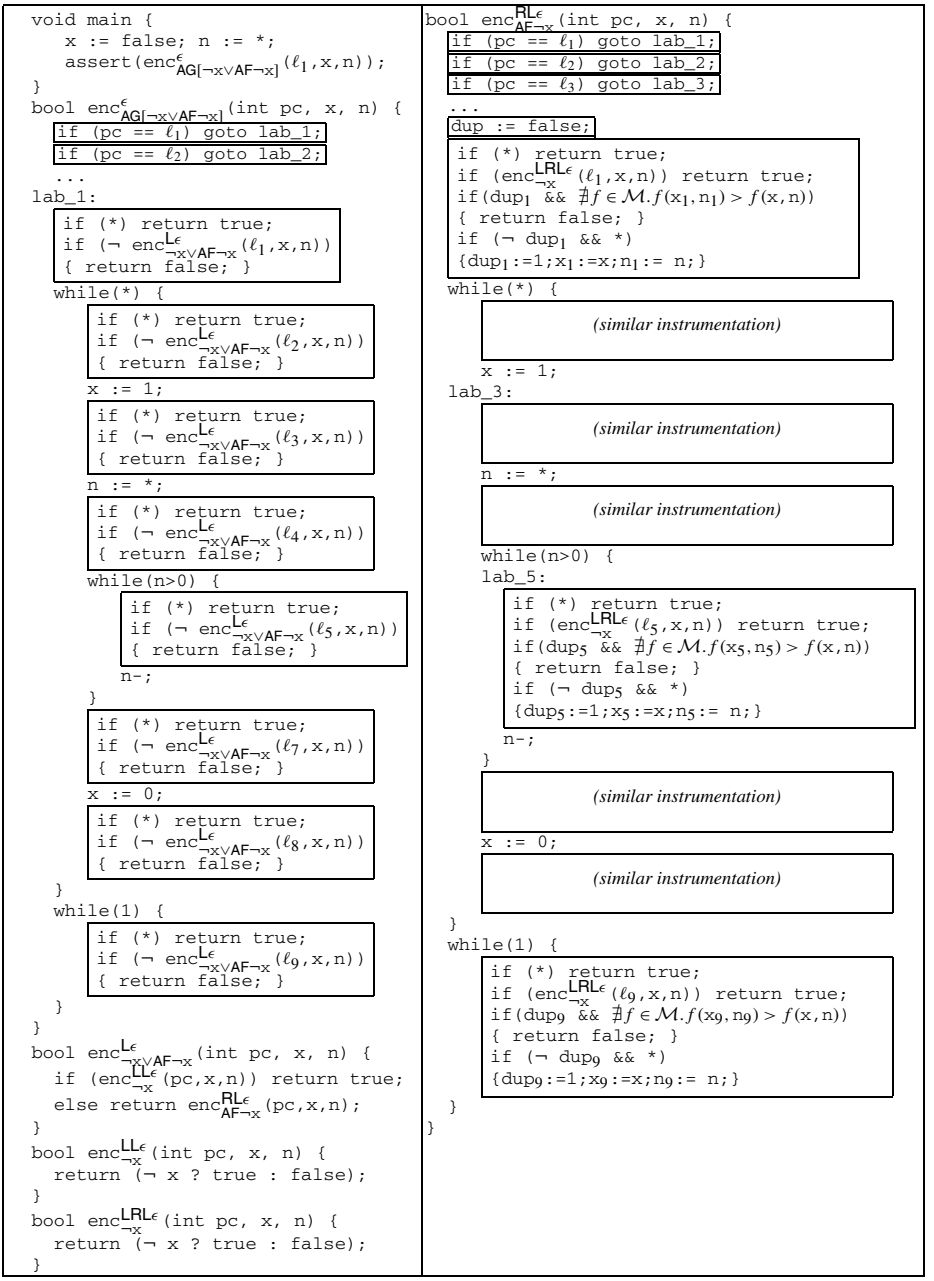

Fig. 3 The encoding $\mathcal{E}$ for the program in Sect. 2 and property $\mathsf{AG}[x \Rightarrow \mathsf{AF}\ \neg x]$. This output can be specialized w.r.t. the program counter, and pruned (via intraprocedural analysis) to obtain the more efficient encoding in Fig. 1

will be such a feasible execution of $\mathcal{E}$, where at s_n both $\texttt{enc}^{\mathsf{R}\kappa}_{\psi'}$ and $\texttt{enc}^{\mathsf{L}\kappa}_{\psi}$ can return false. Consequently, $\texttt{enc}^{\kappa}_{\mathsf{A}[\psi\,\mathsf{W}\,\psi']}$ can return false. Alternatively, if this is not the case, then a proof that $\texttt{enc}^{\kappa}_{\mathsf{A}[\psi\,\mathsf{W}\,\psi']}$ cannot return false implies that $\mathsf{A}[\psi\,\mathsf{W}\,\psi']$ indeed holds of s. Notice that, since $\mathsf{AG}\psi = \mathsf{A}[\psi\,\mathsf{W}\,\mathsf{false}]$ is a special case of $\mathsf{A}\,\mathsf{W}$, the encoding of AG is also a special case of the encoding of $\mathsf{A}\,\mathsf{W}$ (as see in Figs. 1 and 3):

$$
\begin{array}{l}
\texttt{bool } \texttt{enc}^{\kappa}_{\mathsf{AG}\psi}\texttt{(state } s\texttt{) \{} \\
\quad P\left[c/\left[\begin{array}{l}\texttt{if (*) return true;} \\ \texttt{if (}\neg\ \texttt{enc}^{\mathsf{L}\kappa}_{\psi}\texttt{(}s\texttt{)) return} \\ \texttt{false;}\end{array}\right]\ ;\ c\right] \\
\texttt{\}}
\end{array}
$$

Eventuality We use a similar encoding for AF, also shown in Fig. 2. Our encoding must allow a program analysis to demonstrate that all paths must eventually reach a state where the subformula holds. We use two auxiliary variables called dup and 's. While exploring the reachable states in R the encoding may, at every 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 $f \in \mathcal{M}$ that witnesses the well-foundedness of the nonreflexitive transitive closure of this particular subset ($\mathsf{walk}^{\mathcal{F}}_{I}$) of the transition relation.[1]

When applying the encoding to the example in Sect. 2, we obtain the output P' given in Fig. 3. The procedure main initializes the variables and asserts that $\texttt{enc}^{\epsilon}_{\mathsf{AG}[\neg x \vee \mathsf{AF}\neg x]}$ cannot return false. As described above $\texttt{enc}^{\epsilon}_{\mathsf{AG}[\neg x \vee \mathsf{AF}\neg x]}$ uses command replacement, establishing feasible paths to the subproperty procedure at every reachable state. The procedures $\texttt{enc}^{\mathsf{L}\epsilon}_{[\neg x \vee \mathsf{AF}\neg x]}$, $\texttt{enc}^{\mathsf{LL}\epsilon}_{\neg x}$, and $\texttt{enc}^{\mathsf{LRL}\epsilon}_{\neg x}$ are straight-forward, following Fig. 2. Finally, the procedure $\texttt{enc}^{\mathsf{RL}\epsilon}_{\mathsf{AF}\neg x}$ will return false when called from a state from which the subproperty does not eventually hold. This procedure again uses command replacement, instrumenting a check that, if $\texttt{enc}^{\mathsf{LRL}\epsilon}_{\neg x}$ does not yet hold, then there must be a witness $f \in \mathcal{M}$ to the well-foundedness of this region of the transition relation.

Our procedural encoding lets us apply several static optimizations that facilitate the application of current program analysis tools. These optimizations are described in [25], and allow us to reduce the full, expanded encoding (*e.g.* Fig. 3) into a simpler version, more amenable to analysis tools (*e.g.* Fig. 1). For example, because the program state is passed on the stack, a procedure call $\texttt{enc}^{\kappa}_{\psi}$ for a subformula ψ 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.

4.1 Looking for $\mathcal{M}$

In addition to the safety component, we must also solve the liveness component. Specifically, we must find a finite set of ranking functions $\mathcal{M}$ such that a program analysis can prove for every $s \in I$ that $\texttt{enc}^{\epsilon}_{\Phi}(s)$ does not return false. Our top-level procedure adapts a known method [18] in order to iteratively find a sufficient $\mathcal{M}$:

[1] This is an adaptation of a known technique [18]. 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.

Algorithm 2 For a program P and ∀CTL property Φ,

```
let prove(P, Φ) =
  M := ∅
  while (E(P, M, Φ) 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
```

In our implementation new ranking functions are automatically synthesized by examining counterexamples. A counterexample in ∀CTL is tree-like as follows:

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

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 is simply a state in which α does not hold. Counterexamples for conjunction and disjunction are as expected. 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 A W 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 and BLAST that report stack-based traces (through $\mathcal{E}$) 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 a 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 [18]. In industrial examples these programs rarely occur.

If we begin with $\mathcal{M} = \emptyset$, a safety tool will report a lasso-shaped counterexample when applied to Fig. 1. The loop portion of the lasso arises from the inner `while`-loop. From this

counterexample, we can synthesize the rank function

$$f = \left(\lambda \begin{bmatrix} \mathtt{x} \\ \mathtt{n} \\ \mathtt{pc} \end{bmatrix} . \mathtt{n} \right).$$

Letting $\mathcal{M} = \{f\}$, a safety tool will report no safety violations. Thus we can conclude that the property holds.

5 Related work

∀CTL verification has previously been given in the form of finding winning strategies in finite-state games or game-like structures such as alternating automata [4, 26, 34]. The encoding presented in this paper is, effectively, a generalization of prior work to games over infinite state spaces. The relationship to games is discussed in Koskinen's dissertation [25].

Other previous tools and techniques are known for proving temporal properties of finite-state systems (*e.g.* [8, 14, 26]) or classes of infinite-state systems with specific structure (*e.g.* pushdown systems [36, 37] or parameterized systems [20]). Our proposal works for arbitrary transition systems, including programs.

A previous tool proves only trace-based (*i.e.* linear-time) properties of programs [15] 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 [16]. As shown in [16] and Sect. 6, in most cases our new approach is faster for LTL verification than [15] by several orders of magnitude.

When applying traditional bottom-up based methods for state-based logics (*e.g.* [12, 19, 21]) 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 procedures 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). A safety analysis for infinite-state systems will of course over-approximate this set of states, but it will never need to find approximations for unreachable states. By contrast, bottom-up algorithms require that concrete unreachable states be considered. Furthermore, in our technique, only relevant state/subformula pairs are considered. Our encoding will only consider a pair s, Φ where $R, s \vDash \Phi$ is needed to either prove the outermost property, or is part of a valid counterexample.

Chaki *et al.* [10] 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. This strategy facilitates abstraction refinement: after our encoding has been generated, the underlying program analysis tool can iteratively perform abstraction and refinement. The approach due to Schmidt and Steffen [33] is similar.

The tool YASM [24] 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 [23] (*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 Fig. 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. Our encoding, in contrast, is not a recursively defined algorithm that calls a model checker at each recursion level, but rather a transformation that produces a procedural program that encodes the proof search-space. This program is constructed such that it can later be symbolically analyzed using (infinite-state) program analysis techniques. When applied to the encoding, the underlying analysis tool is then given the task of finding the necessary finite abstractions and possibly procedure summaries.

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 Algorithm 2. 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 temporal verification 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 [15]. Sources of examples can be found elsewhere [25]. Heap commands from the original sources have been abstracted away using the approach due to Magill *et al.* [27]. 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 Sect. 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 [24] 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 [23], 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.* [10]. 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

Program	LOC	Property	Prev. tool [15] Time	Res.	Our tool (Sec. 4) Time	Res.
Acq/rel	14	$\mathsf{AG}(a \Rightarrow \mathsf{AF}b)$	103.48	✓	14.18	✓
Fig. 8 of [15]	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	✓
PSQL smsrv	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	>14400.00	**???**	9.56	✓
PSQL smsrv+bug	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	87.31	χ	47.16	χ
PSQL pgarch	61	$\mathsf{AFAG}p$	31.50	✓	15.20	✓
Apache progress	314	$\mathsf{AG}(p \Rightarrow (\mathsf{AFAF}))$	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. 4 Comparison between our tool and Cook *et al.* [15] on ∀CTL verification benchmarks. All of the above ∀CTL properties have equivalent LTL properties so they are suitable for direct comparison with the LTL tool [15]

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 [11] 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.* [15] can be used to prove LTL properties if used in combination with an LTL to Büchi automata conversion tool (*e.g.* [22]). We compare our approach to this tool using ∀CTL challenge problems in Fig. 4. We have chosen properties that are equivalent in ∀CTL and LTL and then directly compared Algorithm 2 to the tool in Cook *et al.* [15]. Experiments were run using Windows Vista and an Intel 2.66 GHz processor. We also previously reported experiments using our approach in combination with an iterated symbolic determinization in order to prove Linear Temporal Logic properties [16].

In Fig. 4 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 "Res." 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 "**???**". 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 a third of the benchmarks.

7 Conclusions

We have introduced a novel temporal reasoning technique for (potentially infinite-state) transition systems, with an implementation designed for those 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 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.

Acknowledgements 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.

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 SK, Ustuner A (2006) Thorough static analysis of device drivers. SIGOPS Oper Syst Rev 40:73–85
3. Berdine J, Chawdhary A, Cook B, Distefano D, O'Hearn PW (2007) Variance analyses from invariance analyses. In: Hofmann M, Felleisen M (eds) Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL 2007). ACM, New York, pp 211–224
4. Bernholtz O, Vardi MY, Wolper P (1994) An automata-theoretic approach to branching-time model checking (extended abstract). In: Dill DL (ed) Proceedings of the 6th international conference on computer aided verification (CAV '94). Lecture notes in computer science, vol 818. Springer, Berlin, pp 142–155
5. Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker blast. Int J Softw Tools Technol Transf 9(5–6):505–525
6. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2003) A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation (PLDI'03). ACM, New York, pp 196–207
7. Bradley A, Manna Z, Sipma H (2005) The polyranking principle. Autom Lang Program, 1349–1361
8. Burch J Clarke E et al (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
9. Calcagno C, Distefano D, O'Hearn PW, Yang H (2009) Compositional shape analysis by means of bi-abduction. In: Shao Z, Pierce BC (eds) Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL 2009). ACM, New York, pp 289–300
10. Chaki S, Clarke EM, Grumberg O, Ouaknine J, Sharygina N, Touili T, Veith H (2005) State/event software verification for branching-time specifications. In: Romijn J, Smith G, van de Pol J (eds) Proceedings of the 5th international conference on integrated formal methods (IFM 2005), vol 3771, pp 53–69
11. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: an opensource tool for symbolic model checking. In: Brinksma E, Larsen KG (eds) Proceedings of the 14th international conference on computer aided verification (CAV'02), vol 2404. Springer, Berlin, pp 359–364
12. Clarke E, Grumberg O, Peled D (1999) Model checking
13. Clarke E, Jha S, Lu Y, Veith H (2002) Tree-like counterexamples in model checking. In: Proceedings of the symposium on logic in computer science (LICS'02), pp 19–29
14. Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8:244–263
15. Cook B, Gotsman A, Podelski A, Rybalchenko A, Vardi MY (2007) Proving that programs eventually do something good. In: Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL 2007), pp 265–276
16. Cook B, Koskinen E (2011) Making prophecies with decision predicates. In: Ball T, Sagiv M (eds) Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'11). ACM, New York, pp 399–410
17. Cook B, Koskinen E, Vardi MY (2011) Temporal property verification as a program analysis task. In: Gopalakrishnan G, Qadeer S (eds) Proceedings of the 23rd international conference on computer aided verification (CAV'11), vol 6806. Springer, Berlin, pp 333–348
18. Cook B, Podelski A, Rybalchenko A (2006) Termination proofs for systems code. In: Schwartzbach MI, Ball T (eds) Proceedings of the ACM SIGPLAN 2006 conference on programming language design and implementation, Ottawa, Ontario, Canada, June 11–14, 2006 ACM, New York, pp 415–426
19. Delzanno G, Podelski A (1999) Model checking in CLP. In: Cleaveland R (ed) Proceedings of the 5th international conference on tools and algorithms for construction and analysis of systems (TACAS '99). Lecture notes in computer science, vol 1579. Springer, Berlin, pp 223–239

20. Emerson EA, Namjoshi KS (1996) Automatic verification of parameterized synchronous systems (extended abstract). In: Alur R, Henzinger TA (eds) Proceedings of the 8th international conference on computer aided verification (CAV '96), vol 1102. Springer, Berlin, pp 87–98
21. Fioravanti F, Pettorossi A, Proietti M, Senni V (2010) Program specialization for verifying infinite state systems: an experimental evaluation. In: Alpuente M (ed) Proceedings of the 20th international symposium on logic-based program synthesis and transformation (LOPSTR'10), vol 6564. Springer, Berlin, pp 164–183
22. Gastin P, Oddoux D (2001) Fast ltl to büchi automata translation. In: Berry G, Comon H, Finkel A (eds) Proceedings of the 13th international conference on computer aided verification (CAV 2001), vol 2102. Springer Berlin, pp 53–65
23. Gurfinkel A (2010) Personal communication
24. Gurfinkel A, Wei O, Chechik, M (2006) Yasm: a software model-checker for verification and refutation. In: Ball T, Jones RB (eds) Proceedings of the 18th international conference on computer aided verification (CAV'06), vol 4144, pp 170–174
25. Koskinen E (2012) Temporal verification of programs. PhD thesis, University of Cambridge. To appear
26. Kupferman O, Vardi M, Wolper P (2000) An automata-theoretic approach to branching-time model checking. J ACM 47(2):312–360
27. Magill S, Berdine J, Clarke EM, Cook B (2007) Arithmetic strengthening for shape analysis. In: Nielson HR, Filé G (eds) Proceedings of the 14th international static analysis symposium (SAS 2007), vol 4634. Springer, Berlin, pp 419–436
28. O'Hearn P, Reynolds J, Yang H (2001) Local reasoning about programs that alter data structures. In: Computer science logic, pp 1–19
29. Podelski A, Rybalchenko A (2004) A complete method for the synthesis of linear ranking functions. In: Steffen B, Levi G (eds) Proceedings of the 5th international conference on verification, model checking, and abstract interpretation (VMCAI'04), vol 2937. Springer, Berlin, pp 239–251
30. Podelski A, Rybalchenko A (2004) Transition invariants. In: Proceedings of the 19th IEEE symposium on logic in computer science (LICS 2004). IEEE Computer Society, New York, pp 32–41
31. Podelski A, Rybalchenko A (2005) Transition predicate abstraction and fair termination. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL 2005)
32. Reps T, Horwitz S, Sagiv M (1995) Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'95), pp 49–61
33. Schmidt DA, Steffen B (1998) Program analysis as model checking of abstract interpretations. In: Levi G (ed) Proceedings of the 5th international static analysis symposium (SAS '98), vol 1503. Springer, Berlin, pp 351–380
34. Stirling C (1996) Games and modal mu-calculus. In: Margaria T, Steffen B (eds) Proceedings of the second international workshop on tools and algorithms for construction and analysis of systems (TACAS '96), vol 1055, pp 298–312
35. Vardi MY (1995) An automata-theoretic approach to linear temporal logic. In: Banff Higher order workshop, pp 238–266
36. Walukiewicz I (1996) Pushdown processes: games and model checking. In: Alur R, Henzinger TA (eds) Proceedings of the 8th international conference on computer aided verification. Lecture notes in computer science, vol 1102. Springer, Berlin, pp 62–74
37. Walukiewicz I (2000) Model checking CTL properties of pushdown systems. In: Kapoor S, Prasad S (eds) Proceedings of the 20th conference on foundations of software technology and theoretical computer science (FST TCS 2000). Springer, Berlin, pp 127–138. 1974