Faster Temporal Reasoning for Infinite-State Programs

Byron Cook Microsoft Research & University College London
Heidy Khlaaf University College London
Nir Piterman University of Leicester

***Abstract*—In this paper, we describe a new symbolic model checking procedure for CTL verification of infinite-state programs. Our procedure exploits the natural decomposition of the state space given by the control-flow graph in combination with the nesting of temporal operators to optimize reasoning performed during symbolic model checking. An experimental evaluation against competing tools demonstrates that our approach not only gains orders-of-magnitude performance improvement, but also allows for scalability of temporal reasoning for larger programs.**

I. Introduction

Branching-time temporal logics like CTL allow us to reason about safety, termination, and nontermination via the system's interaction with inputs and nondeterminism in a way that linear-time temporal logics like LTL do not. This style of reasoning can be useful in applications ranging from planning, games, security analysis, disproving, and environment synthesis [19], [29]. CTL-based tools also have been used as the basis for higher-performance LTL tools [13].

In this paper we propose a new symbolic CTL model checker for infinite-state programs. We adapt the well-known bottom-up strategy for finite-state CTL model checking [9] to infinite-state programs using precondition synthesis as the enabling technology. We leverage recent techniques for proving safety, termination, and nontermination of programs to synthesize preconditions asserting the satisfaction of CTL sub-formulae of an input property. The key insight to our approach is the exploitation of the natural decomposition of the state space given by the control flow graph. That is, using a counterexample-guided precondition synthesis strategy, we compute program-location-specific preconditions. Our model checker drastically improves performance by reducing the amount of redundant and irrelevant reasoning performed through information sharing extracted from reachability analysis. That is, several preconditions for each program location can be computed simultaneously.

Take for example the fact that the set of states respecting a property such as $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$ *before* a program command is very often the same or nearly the same as the set of states respecting $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$ *after* the command. In comparison to existing tools (*e.g.* [10], [4]) we reduce the amount of reasoning performed as part of the procedure. We can infer whether a command is likely to affect the truth of $\mathsf{EF}\ \mathsf{y} < \mathsf{z}$. So, sequential locality implies that the precondition of a location is easily computed if the preconditions of its successors are known.

This approach gives way to compositional reasoning. For instance, given a program and a desired property, we can, in parallel, synthesize preconditions, desired environments, and plans of individual procedures of a program with the goal of composing the found preconditions into a proof of the whole program. The advantage to this approach is that the program verification tools never have to examine the program as a whole, but instead find a modular proof using compositional reasoning. Recent success in this style of reasoning can be found in areas such as proving correctness of non-blocking algorithms [20], and the analysis of biological models [11].

We also suggest a new method of treating existential path quantification in the infinite-state setting. Existential formulas are handled by considering their universal dual, allowing counterexamples of said duals to serve as a witness asserting the satisfaction of the existential CTL formula.

An experimental evaluation using examples from the benchmark suites of the competing tools (which are drawn from industrial benchmarks) demonstrates orders-of-magnitude performance improvements in many cases. This evaluation is discussed at the conclusion of the paper.

Related work. Model checking has been extensively studied in the context of finite-state systems (*e.g.* [3], [5], [7], [8], [25]) as well as for various types of systems with limitations on their infiniteness (*e.g.* pushdown systems [17], parameterized systems [16], timed systems [2], etc.). In recent years new tools have been developed for proving temporal properties of integer programs, *e.g.* [12], [32], [33], [34], [22], [10], [4]. These tools go beyond, *e.g.* SMV, which is restricted to finite-state programs [6].

In this work we are aiming to prove CTL properties with nested combinations of existential and universal path quantifiers of integer programs. Song & Touili [32] perform a coarse one-time abstraction that takes programs and produces pushdown automata, however the abstraction produced is imprecise and leads to significant information loss. Gurfinkel *et al.* [22] do not reliably support mixtures of nested universal/existential path quantifiers, etc. The two tools closest in their feature set to our setting are from Cook & Koskinen [10] and Beyene *et al.* [4]. Cook & Koskinen implement the Kesten and Pnueli [24] deductive proof system using an incremental reduction to program analysis tools. Beyene *et al.* [4] implement the same idea as Cook & Koskinen using a reduction to Horn-clause reasoning. Neither Cook & Koskinen nor Beyene *et al.* make use of the locality in programs as we do. Effectively,

these tools carryout unnecessary computation in their analysis.

In addition, our new approach to the treatment of existential path quantification based on dualization contrasts to that of Cook & Koskinen, which attempts to find a non-trivial restriction on the state-space such that AF can be used to reason about EF, or AG can be used to reason about EG. Our approach also contrasts to the tool of Beyene *et al.* [4], as their tool requires existential quantification over predicates to be supported by the underlying constraint solver, whereas our technique can make use of off-the-shelf verification tools.

Limitations. We do not support programs with heap, nor do we support recursion or concurrency. The heap-based programs we consider during our experimental evaluation have been abstracted using the over-approximation from the technique of Magill *et al.* [26]. Note that this abstraction can lead to unsoundness when we use the existential subset of CTL. Our comparison to existing tools remains fair, as each of the previous tools uses the same abstraction. Effective techniques for proving temporal properties of programs with heap remains an open research question.

As our technique heavily relies on calculating pre-images, it is important that fragments of the underlying program logic are closed under pre-impages, *e.g.* integer linear arithmetic, a fragment of integer arithmetic. Our procedure is not complete as we use a series of incomplete subroutines.

II. Preliminaries

Programs. As is standard [27], we treat programs as control-flow graphs, where edges are annotated by the updates they perform to variables. A program is a triple $P = (\mathcal{L}, E, \mathsf{Vars})$, where $\mathcal{L}$ is a set of locations, E is a set of edges/transitions, and Vars is a set of variables. Each edge $\tau = (\ell, \rho, \ell')$ in E, where $\ell, \ell' \in \mathcal{L}$ and ρ is a condition, specifies possible transitions in the program. A *valuation* associates with the variables in Vars values in Vals. A *trace* or a *path* of a program is either a finite or an infinite sequence of edges allowed by the program. The condition ρ is an assertion in terms of Vars and Vars', a primed copy of Vars, where constants range over Vals. Intuitively, Vars refers to the values of variables before the update and Vars' refers to the values of variables after the update. The set of locations includes the first location ℓ_0 that has no incoming transitions from other program locations. That is, for every $\tau = (\ell, \rho, \ell') \in E$ we have $\ell' \neq \ell_0$. Transitions exiting ℓ_0 have their conditions expressed in terms of Vars'. Locations with incoming transitions from ℓ_0 are *initial locations*. This allows us to encode more complex initial conditions. In figures we usually omit ℓ_0 and add edges with no source to locations having an incoming transition from ℓ_0. The program gives rise to a transition system $T = (S, R)$, where S is the set of program states of the form $S = (\mathcal{L} - \{\ell_0\}) \times (\mathsf{Vars} \to \mathsf{Vals})$ and $R \subseteq S \times S$. That is, a program state is a pair (ℓ, s) where $\ell \neq \ell_0$ and s is a valuation, i.e., a function from program variables to values. The program can transition from (ℓ, s_1) to (ℓ', s_2) if there is a transition $(\ell, \rho, \ell') \in E$ such that $(s_1, s_2) \models \rho$. Here the valuation (s_1, s_2) is a function from $\mathsf{Vars} \cup \mathsf{Vars}'$ to Vals such that for every $v \in \mathsf{Vars}$ we have $(s_1, s_2)(v) = s_1(v)$ and $(s_1, s_2)(v') = s_2(v)$. A state (ℓ, s) is initial if there is a transition (ℓ_0, ρ, ℓ) such that $(s_{-1}, s) \models \rho$, where s_{-1} is some arbitrary state. Notice that in such a case ρ is expressed in terms of Vars' and hence the state s_{-1} does not affect the evaluation of ρ. For example, Figure 1 includes the representation of the program **while** $\mathsf{x} \leq 0$ **do if** * **then** x := x + 1; **fi**; **done**; y := 1; with initial condition $\mathsf{x} = * \wedge \mathsf{y} = 0$ and where $*$ indicates a non-deterministic value.

A finite set of program locations $C \subseteq \mathcal{L}$ is called a *cut-point set* if $\ell_0, \ell_n \in C$, where $n \in \mathbb{N}$ and every cycle in the program's graph contains at least one cut-point.

CTL. We are interested in verifying state-based temporal properties in computational tree logic (CTL) [9]. A CTL formula is of the form

$$\varphi ::= \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AG}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi] \mid \mathsf{EF}\varphi \mid \mathsf{EG}\varphi \mid \mathsf{E}[\varphi \mathsf{U} \varphi]$$

where α is an atomic proposition (*e.g.* $\mathsf{x} < \mathsf{y}$).

To give intuition behind the semantics of CTL, here $P, s \models \mathsf{AF}\varphi$ asserts that in program P and in all possible executions starting from s the property φ will eventually hold in some future state reachable from s, whereas $P, s \models \mathsf{EF}\varphi$ asserts that there is a state in which φ holds and that it can be reached from s. The formula $\mathsf{AG}\varphi$ asserts that φ must hold throughout all possible executions, while $\mathsf{EG}\varphi$ asserts that there exists an execution such that φ would be true throughout. $\mathsf{A}\varphi_1 \mathsf{W} \varphi_2$ asserts that for all executions, φ_1 has to hold until φ_2 holds, signifying that φ_2 does not necessarily have to hold as long as φ_1 holds. Contrarily, $\mathsf{E}\varphi_1 \mathsf{U} \varphi_2$ asserts that there exists an execution in which φ_1 has to hold *at least until* at some position φ_2 holds. AU and EW are represented as syntactic sugar as usual.

For a program P and a CTL property φ, we say that φ holds in P, denoted by $P \models \varphi$ if for every initial state s we have $P, s \models \varphi$.

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 non-empty 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, a set of ranking functions $\{f_1, \ldots, f_n\}$ denotes the disjunctively well-founded relation $\{(s, s') \mid f_1(s') \prec f_1(s) \vee \ldots \vee f_n(s') \prec f_n(s)\}$.

Counterexamples. In our setting new ranking functions can be automatically synthesized by examining counterexamples produced by an underlying safety prover (discussed in more detail in Section IV). Due to the recursive nature of our procedure it is only necessary to handle counterexamples to formulas of nesting depth 1. For example, $\mathsf{A}\varphi$, where φ is a path formula that includes no nesting of additional operators, or $\alpha_1 \vee \alpha_2$, where α_1 and α_2 are assertions. A counterexample for an atomic proposition α is a state in which α does not hold. A counterexample for a conjunction $\varphi_1 \wedge \varphi_2$ is a state

where either φ_1 or φ_2 does not hold. A counter example for disjunction $\varphi_1 \vee \varphi_2$ is a state where both sub-formulas do not hold. A counterexample to an $\mathsf{AG}\varphi$ property is a path to a place where φ does not hold. A counterexample to an $\mathsf{AF}\varphi$ property is a "lasso": a stem path to a particular program location, then a (not necessarily simple) cycle which returns to the same program location, and the property φ does not hold along the stem and the cycle. Finally, a counterexample to $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ is a path to a place where there is a sub-counterexample to φ_1 as well as one to φ_2. A counterexample to $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ can be of the same form as that of $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, as well as one where φ_1 holds while φ_2 does not hold anywhere along the path.

Calculating pre-images. Let $\pi = (\ell_0, \rho_0, \ell'_0), (\ell_1, \rho_1, \ell'_1), \ldots, (\ell_n, \rho_{n-1}, \ell'_n)$ be a path. We compute a pre-image for every possible suffix of π. That is, we denote $pre_{n+1} = S$ and $pre_i = pre((\ell_i, \rho_i, \ell'_i), \ldots, (\ell_n, \rho_n, \ell'_n))$ as the set of states such that $pre_i = \{s \mid \exists s' \in pre_{i+1} \text{ s.t. } ((\ell_i, s), (\ell'_i, s')) \models \rho_i\}$. Generaly speaking, given an assertion α (in terms of Vars) representing pre_{i+1}, and an assertion ρ_i (in terms of Vars and Vars') we must compute an assertion representing pre_i. Let α' denote $\exists\ \mathsf{Vars}.\ \mathsf{Vars} = \mathsf{Vars}' \wedge \alpha$. We thus consider $\exists\ \mathsf{Vars}'(\mathsf{Vars} = \mathsf{Vars}' \wedge \exists\ \mathsf{Vars}.\ (\mathsf{Vars} = \mathsf{Vars}' \wedge (\alpha' \wedge \rho_i)))$. We use Fourier-Motzkin for quantifier elimination.

III. Intuition and Example

We first informally explain our technique and demonstrate it with an example.

Intuition. The idea of the procedure is to find for each sub-formula φ a precondition $\wp\langle\varphi\rangle$ that ensures its satisfaction. To utilize sequential locality of a counterexample's control-flow graph further on, a precondition $\wp\langle\varphi\rangle$ is thus partitioned to $\wp\langle\ell_i, \varphi\rangle$ for every location ℓ_i in the program. Thus, $\wp\langle\varphi\rangle$ takes the form $\bigwedge_{\ell_i}(\mathsf{pc} = \ell_i \Rightarrow \wp\langle\ell_i, \varphi\rangle)$. Here $\mathsf{pc} = \ell_i$ is used to assert that the state is at location ℓ_i in the program's control-flow graph. We find preconditions by iteratively recursing over the structure of the given CTL formula. That is, we start by finding the precondition of the innermost sub-formula followed by search for the preconditions of the outer sub-formulas dependent on it. We note that the precondition of an atomic proposition is the proposition itself, hence from this point on, we shall treat the precondition of an atomic proposition and the atomic proposition itself synonymously.

Consider a universal CTL formula. Initially, we approximate its precondition as true. We then search for counterexamples from every possible reachable program location. Failures to the proof attempt will result in the strengthening of the precondition through adding the negation of the pre-image of the discovered counterexample. We use the control-flow graph of a counterexample to simultaneously synthesize preconditions of multiple locations. That is, a counterexample that consists of multiple program locations can be utilized to update the precondition of each contained program location. This is done by iterating along the counterexample path, and for each suffix computing a pre-image from a program location onwards.

Each counterexample found further strengthens a precondition, we thus eliminate said counterexample and search for

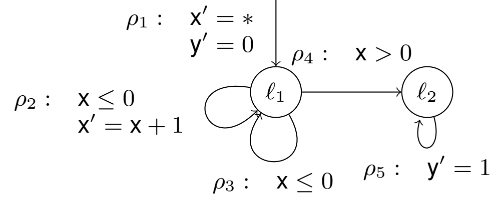

Fig. 1: The control-flow graph of an example program for which we wish to prove the CTL property $\mathsf{AGEF}\ \mathsf{y} = 1$.

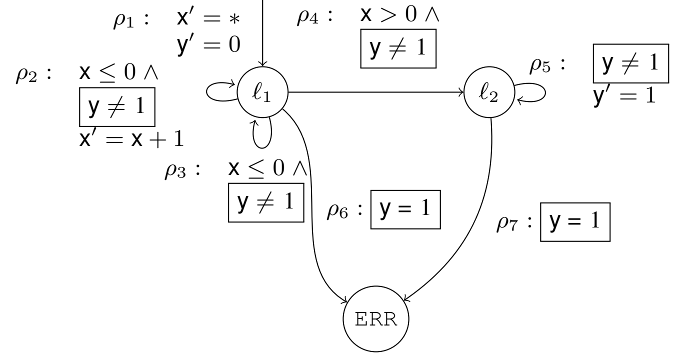

Fig. 2: The transformation of the program from Figure 1 for the property $\mathsf{EF}\ \mathsf{y} = 1$ using its dual $\mathsf{AG}\ \mathsf{y} \neq 1$.

other proof failures for the given CTL property. Eventually, the precondition will imply the correctness of the sub-formula when no further counterexamples are returned.

Existential sub-formulas are handled by considering their universal dual. We thus seek a set of counterexamples generated from the property's universal dual to serve as an existential witness. Hence we begin with an initial precondition approximation false. More directly, pre-images of counterexamples to the negation of the sub-formula serve as a witnesses to the satisfaction of our existential formula. Counterexamples are similarly treated in the existential case, we iteratively calculate their pre-images followed by their elimination until no more counterexamples are generated. As before, we utilize a counterexample's control flow graph to simultaneously update preconditions of multiple locations.

Example. Consider the program in Figure 1 and the property $\varphi \equiv \mathsf{AGEF}\ \mathsf{y} = 1$, which states that for all states, it is always possible that eventually $\mathsf{y} = 1$. The approach followed by nearly all tools supporting CTL would be to find, in this instance, a set of states $\wp$ such that $\mathsf{AG}\wp$ holds, and such that $\wp \models \mathsf{EF}\ \mathsf{y} = 1$ holds. In this paper we suggest a strategy based on precondition synthesis.

Consider the sub-formula $\psi \equiv \mathsf{EF}\ \mathsf{y} = 1$. For the proposition $\mathsf{y} = 1$, for every program location ℓ_i we have $\wp\langle\ell_i, \mathsf{y} = 1\rangle \triangleq \mathsf{y} = 1$. We now attempt to prove that $\wp \not\models \mathsf{AG}\ \mathsf{y} \neq 1$ given that AG is EF's universal dual. We start with $\wp\langle\psi\rangle \triangleq \mathsf{false}$ as only failures to proving $\mathsf{AG}\ \mathsf{y} \neq 1$ can necessitate that there exists a witness such that $\mathsf{EF}\ \mathsf{y} = 1$. Failures to the proof attempt will result in refinements to $\wp$ through the iterative calculation of the pre-image of each discovered counterexample. Recall that we are interested in counterexamples starting from all program locations:

$$\wp\langle\psi\rangle \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \wp\langle\ell_1, \psi\rangle) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \wp\langle\ell_2, \varphi\rangle).$$

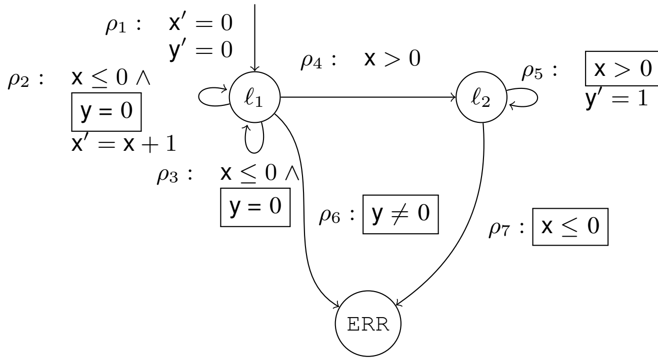

Fig. 3: The transformation of the program from Figure 1 for the sub-property AGEF y = 1 to be utilized in the verification algorithm. The nested property EF y = 1 is substituted with its precondition resulting in a transformation for AG $((\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \vee (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0))$ instead.

We begin with ℓ_1. To check $\mathsf{AG}\ \mathsf{y} \neq 1$ we use a source-to-source transformation that reduces checking of universal CTL properties to safety [10]. The transformation returns the program in Figure 2 (new conditions outlined), on which we use a safety prover to check reachability of ERR. We get counterexample CEX_1: $\langle \ell_0, \rho_1, \ell_1\rangle$, $\langle \ell_1, \rho_3, \ell_1\rangle$, $\langle \ell_1, \rho_2, \ell_1\rangle$, $\langle \ell_1, \rho_4, \ell_2\rangle$, $\langle \ell_2, \rho_5, \ell_2\rangle$, $\langle \ell_2, \rho_7, \mathrm{ERR}\rangle$.

We then calculate the pre-image of CEX_1 for multiple locations along the counterexample. We do so by iterating along the counterexample path, and for every reachable location $\ell \in \mathcal{L}$ in CEX_1, we compute a pre-image utilizing the suffix of CEX_1 from ℓ onwards. Thus we can avoid redundant reasoning by utilizing sequential locality based upon the program's control-flow graph to compute a refinement for ℓ_2 from a counterexample generated for ℓ_1. In this case, we compute $\wp \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0)$

One existential witness may not be sufficient to find all states that satisfy ψ in the respective locations, we thus rule out CEX_1 by adding $\neg\wp\langle \ell_i, \psi\rangle$ to each transition from ℓ_i to the error state. We re-run our safety checker and find that we do not generate anymore counterexamples, thus completing our precondition synthesis for $\mathsf{EF}\ \mathsf{y} = 1$.

Note that the technique used by Cook & Koskinen [10] imposes that they spend time computing both $\wp\langle \ell_1, \psi\rangle$, $\wp\langle \ell_2, \psi\rangle$ separately while the technique used by Beyene *et al.* [4] solves a constraint based on an entire path when it's only necessary to reason about a single state.

We now modify φ by using $\wp\langle\psi\rangle$ and get $\varphi' = \mathsf{AG}\ ((\mathsf{pc} = \ell_1 \Rightarrow \mathsf{y} = 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} > 0))$. The constructed transformation reducing the property φ' to safety can be seen in Figure 3. Note that in this particular transformation, the outlined instrumented conditions correspond to each of the location preconditions generated for $\mathsf{EF}\ \mathsf{y} = 1$. As φ' is universal, we begin with the initial precondition $\wp\langle\varphi\rangle \triangleq \mathsf{true}$. Failures to the proof attempt will result in strengthening the precondition by adding *negated* pre-images of discovered counterexamples. In this case no counterexamples are returned and we get $\wp\langle\varphi\rangle \triangleq \mathsf{true}$. This proves that $\mathsf{AGEF}\ \mathsf{y} = 1$ holds.

IV. PROCEDURE

In this section we describe the details of our CTL model checking procedure. Figure 4 depicts VERIFY, which wraps

```
1  let VERIFY (φ,P) : bool =
2
3      (ℒ,E,Vars) = P
4      ℘ = TEMPORALWP(φ,P)
5      return ∀(ℓ0,ρ,ℓ) ∈ E ∀s . (s,s) ⊨ ρ ⇒ s ⊨ ℘⟨ℓ,φ⟩
```

Fig. 4: Procedure VERIFY, which wraps TEMPORALWP and then checks all initial states.

```
1  let rec TEMPORALWP(ψ,P) : map =
2    ℘ = INITIALIZEMAP (ψ,P)
3    ℳ = ∅
4    κ = [ ]
5    (ℒ,E,Vars) = P
6    if ψ = α  is atomic  then
7      foreach {ℓ | (ℓ,t,ℓ′) ∈ E}
8         ℘⟨ℓ,ψ⟩ = pre(t,α) ; ℘⟨ℓ,¬ψ⟩ = ¬pre(t,α)
9      done
10   else
11     match (ψ) with
12     | ψ′1∧ψ′2 | ψ′1 ∨ ψ′2 | ψ′1Uψ′2 | ψ′1Wψ′2 →
13        ℘ = ℘ ∪ TEMPORALWP(ψ′1,P)∪ TEMPORALWP(ψ′2,P)
14     | AFψ′1 | AGψ′1 | ¬ψ′1 →
15        ℘ = ℘ ∪ TEMPORALWP(ψ′1,P)
16     C = FINDCUTPOINTS(P)
17     foreach ℓ ∈ C  do
18        P′ = TRANSFORM(⟨ℓ,ψ⟩,ℳ,P,℘)
19        CEX, ℳ = REFINE(P′,ψ,℘,ℳ)
20        while CEX ≠ ∅ do
21          ℘,P′ = PROPAGATE(CEX,P′,κ,ψ,ℓ,℘)
22          κ = CEX :: κ
23          CEX,ℳ = REFINE(P′,ψ,℘,ℳ)
24        done
25     done
26   ℘
```

Fig. 5: Procedure TEMPORALWP getting a temporal property and a program and returning the map from program locations and sub-formulas to assertions.

the main procedure TEMPORALWP in Figure 5. Other subroutines used in TEMPORALWP are in Figures 6–10.

We exploit the natural decomposition of the state space given by the control flow graph. That is, using a counterexample-guided precondition synthesis strategy, we compute program-location-specific preconditions. In our approach the table $\wp$ is the key data structure which maps pairs of program locations and sub-formulae to assertions which represent the current candidate *precondition* that would guarantee the sub-formulae at a respective location. That is, $\wp\langle \ell, \varphi\rangle$ should be a sufficient and most general precondition to prove that φ holds at location ℓ. If such is not the case, a counterexample is produced and the procedure attempts to refine $\wp$

```
1  let INITIALIZEMAP (ψ,P) : map =
2
3    ℘ = ∅
4    (ℒ,E,Vars) = P
5    if ψ = Eψ′ then
6      foreach ℓ ∈ ℒ do
7        ℘⟨ℓ,ψ⟩ = false;
8        ℘⟨ℓ,¬ψ⟩ = true
9      done
10   else
11     foreach ℓ ∈ ℒ do
12       ℘⟨ℓ,ψ⟩ = true;
13       ℘⟨ℓ,¬ψ⟩ = false
14     done
15   return ℘
```

Fig. 6: Initializing the map from program locations and sub-formulas to assertions.

```
1  let REFINE(P,ψ,℘,M) : map =
2
3    CEX = REACHABLE(P,ERR)
4    while P can reach ERR do
5      if CEX contains stem and lasso then
6        if ∃ witness f showing CEX′ w.f. then
7          M = M ∪ {f}
8        else
9          return CEX, M
10     else
11        return CEX, M
12     CEX = REACHABLE((TRANSFORM(⟨ℓ,ψ⟩,M,P,℘),ℓ0,ERR)
13   done
```

Fig. 7: Procedure REFINE getting a program, a temporal property, the map from locations and temporal properties to assertions, and a set of ranking functions and returning a counter example reaching location ERR and a (possibly) larger set of ranking functions.

```
1  let PROPAGATE(CEX,P,κ,ψ,n,℘): map =
2    α = true
3    (L,E,Vars) = P
4    foreach (ℓ,ρn,ℓ′) ∈ CEX reachable from n do
5      if CEX in κ ∧ ℓ = n then
6        α = STRENGTHEN(pre(ℓ,CEX),CEX)
7      else
8        α = pre(ℓ,CEX)
9      if ψ = Eψ′ then
10       ℘⟨ℓ,ψ⟩ = ℘⟨ℓ,ψ⟩ ∨ α
11       ℘⟨ℓ,¬ψ⟩ = ℘⟨ℓ,¬ψ⟩ ∧ ¬α
12     else
13       ℘⟨ℓ,ψ⟩ = ℘⟨ℓ,ψ⟩ ∧ ¬α
14       ℘⟨ℓ,¬ψ⟩ = ℘⟨ℓ,¬ψ⟩ ∨ α
15     if ℓ′ = ERR then
16       ρ ∈ E = ρ ∧ ¬℘⟨ℓ,ψ⟩
17   done
18   ℘,P
```

Fig. 8: Procedure PROPAGATE getting a counter example, the program, a list of previous counter examples, the location to which the counter example is applicable, and the map of previously discovered preconditions and returning an updated map and updated program. The map of preconditions is updated by adding the weakest preconditions of the current counter example. The program is updated by eliminating handled counter example from reaching the ERR location again.

given the counterexample path. Each precondition synthesized substitutes its temporal sub-property in the original formula, and we then continue with the next most outer formula. After a short description of TEMPORALWP and a brief description of each of its subroutines, we give an in depth explanation of TEMPORALWP.

TEMPORALWP: performs both a recursive and a refinement-based computation to construct $\wp$. It starts by initializing the map of preconditions using procedure INITIALIZEMAP (Figure 6) and then calling itself recursively for each sub-formula (lines 7–9 and 11–15). TRANSFORM and REFINE are part of the model checking procedure for the current sub-formula while PROPAGATE (Figure 8) updates the map by synthesizing the pre-images given a counterexample. We then reduce the amount of redundant and irrelevant reasoning performed through information sharing extracted from

```
1  let STRENGTHEN(α,CEX) : AP =
2
3    W = {v | v gets updated in CEX}
4    QE(∃W.α)
```

Fig. 9: If divergence is suspected due to infinitely many counterexamples, the sub-procedure strengthens the candidate precondition towards the limit.

```
1  let TRANSFORM(⟨k,φ⟩,M,P,℘) : Program =
2
3  (L,E,Vars) = P
4  match(φ) with
5  | ψ∧ψ′ →
6    α1 = ℘⟨k,¬ψ⟩ ; α2 = ℘⟨k,¬ψ′⟩
7    E = E ∪ (k, α1 ∨ α2, ERR)
8  | ψ∨ψ′ →
9    α1 = ℘⟨k,¬ψ⟩ ; α2 = ℘⟨k,¬ψ′⟩
10   E = E ∪ (k, α1 ∧ α2, ERR)
11 | A[ψWψ′] →
12    foreach (ℓ,ρ,ℓ′) ∈ E reachable from k do
13       α1 = ℘⟨ℓ,ψ⟩ ; α2 = ℘⟨ℓ,ψ′⟩
14       ρ = ρ ∧ α1 ∧ ¬α2
15       E = E ∪ (ℓ, ¬α1 ∧ ¬α2, ERR)
16 | E[ψUψ′] →
17    P = TRANSFORM(⟨k,A[¬ψ′W(¬ψ ∧ ¬ψ′)]⟩,M,P,℘)
18 | AFψ →
19    foreach ('ℓ,ρ,k) ∈ E do
20       ρ = ρ ∧dup=false
21    foreach (ℓ,ρ,ℓ′) ∈ E reachable from k do
22       α = ℘⟨ℓ,ψ⟩
23       ρ = (ρ ∧ ¬α) ∨ (ρ ∧ ¬dup∧¬α
24             ∧ dup=true ∧ ('s = ℓ × Vars → Vals))
25       c = dup ∧ ¬α ∧ ¬(∃f ∈ M. f(s) ≺ f('s))
26       E = E ∪ (ℓ, c, ERR)
27 | EGψ →
28    P = TRANSFORM(⟨k,AF¬ψ⟩,M,P,℘)
29 | AGψ →
30    foreach (ℓ,ρ,ℓ′) ∈ E reachable from k do
31       α = ℘⟨ℓ,ψ⟩
32       ρ = ρ ∧ α
33       E = E ∪ (ℓ, ¬α, ERR)
34 | EFψ →
35    P = TRANSFORM(⟨k,AG¬ψ⟩,M,P,℘)
36
37 P
```

Fig. 10: Reduction of model checking of temporal properties to safety and ranking function synthesis.

reachability information. That is, several preconditions for each program location can be computed simultaneously. When TEMPORALWP returns to VERIFY, it is only necessary to check if the precondition of the outermost temporal sub-property is satisfied by the initial states of the program.

TRANSFORM: implements the reduction of model checking to safety checking and well-foundedness, inspired by the procedure from [10]. The TRANSFORM transformation utilizes the map $\wp$, which maps the preconditions synthesized previously for sub-properties and their negations (lines 6,9,13,22, and 31). The program is then transformed according to the CTL sub-property by modifying the program from a given program location $k \in \mathcal{L}$. The reduction is only applied from a location k onwards (see loop invariants in lines 12, 21, and 30), that is, we only wish to verify the sub-property starting from transitions stemming from k. Whenever φ does not hold for a location ℓ, a new reachable transition to an error location ERR is added.

As mentioned, existential path quantifiers are handled by considering their universal dual. For both existential and universal properties, our mapping function is also updated with the precondition for the negation of the property on line 8 in TEMPORALWP and lines 11 and 14 in PROPAGATE. This allows us to conveniently access the negation of the property when encoding existential properties as their universal duals.

REFINE: uses a safety prover (similar to IMPACT [28]) to obtain counterexamples from the transformed system, if a

counterexample exists. A produced counterexample to a liveness property (such as AF) contains a lasso fragment, we then attempt to find an accompanying set of ranking functions $\mathcal{M}$ that will show that the counterexample is not valid. We thus attempt to enlarge the set of ranking functions $\mathcal{M}$ using the well known method of [14]. Otherwise, the absence of a set of ranking function indicates the existence of a recurrent set. Note that proving liveness is undecidable, thus techniques used in [14] are incomplete.

A. TEMPORALWP*: computing* $\wp$

In order to synthesize a precondition for a temporal property ψ, we first recursively accumulate the preconditions generated when considering the sub-properties of ψ at lines 8, 13, and 15. The base case, an atomic proposition α, is computed as is standard, *e.g.,* in [15]. For the sake of clarity, we omit the descriptions of both FINDCUTPOINTS and the use of sequential locality in PROPAGATE till later, as we solely wish to describe the fundamental procedure underlying our precondition synthesis for each temporal sub-property. We will then discuss how these sub-procedures provide the key to making use of the program's control-flow graph to construct multiple preconditions.

Given the omission of FINDCUTPOINTS, let C be the set of all locations in a program P, that is $\mathcal{L}$. We wish to synthesize a precondition for each $\ell \in \mathcal{L}$ such that the precondition asserts the satisfaction of ψ. Hence, we iterate over these locations (line 17) and generate a transformed program corresponding to each location using the subroutine TRANSFORM at line 18.

Recall that TRANSFORM allows us to reduce the checking of temporal properties to a program analysis task from a given program location. Each transformed program is then verified through the subroutine REFINE (line 19). A counterexample-guided precondition refinement loop then begins at line 20, where we iteratively refine a precondition for $\ell \in \mathcal{L}$ until no more counterexamples are found. We now discuss the refinement process for each type of quantifier separately below.

Universal precondition synthesis. For a universal CTL sub-property ψ, a precondition $\wp\langle \ell, \psi \rangle$ for a program location ℓ is initialized to true (Figure 6 line 12). If REFINE returns a counterexample, we refine $\wp\langle \ell, \psi \rangle$ by taking the negation of pre-image of the returned counterexample at location ℓ in PROPAGATE on line 21. Given our temporary omission of sequential locality in PROPAGATE, consider the loop in Figure 9 on line 3 to only iterate over one element, that is the current ℓ. As we are handling a universal sub-property its precondition is then strengthened to become $\wp\langle \ell, \psi \rangle = \wp\langle \ell, \psi \rangle \wedge \neg pre(\ell, \mathsf{CEX})$ (line 13 in PROPAGATE).

We then rule out the aforementioned counterexample by adding the assumption $\neg pre(\ell, \mathsf{CEX})$ to each ingoing transition to the error location on the counterexample path, as shown on lines 15 and 16 in PROPAGATE. We then continue to unfold the loop in TEMPORALWP whenever a new counterexample is discovered while iteratively refining $\wp\langle \ell, \varphi \rangle$, resulting in:

$$\wp\langle \ell, \varphi \rangle = \bigwedge_{n\in\mathbb{N}} \neg pre(\mathsf{CEX_n})$$

Existential precondition synthesis. For an existential CTL property, a precondition must entail an existential witness satisfying the sub-property ψ at program location ℓ. We thus verify the universal dual of the existential property (as instrumented by our encoding) and seek a set of counterexamples generated from the property's universal dual to serve as an existential witnesses.

A precondition $\wp\langle \ell, \varphi \rangle$ for a program state is initially false (line 7 in Figure 6). If a counterexample is returned, $\wp\langle \ell, \varphi \rangle$ is refined through the disjunction of the pre-image of the counterexample returned, that is $\wp\langle \ell, \psi \rangle = \wp\langle \ell, \psi \rangle \vee pre(\ell, \mathsf{CEX})$ (line 10 in Figure 8).

We rule out the aforementioned counterexample by adding the assumption $\neg pre(\ell, \mathsf{CEX})$, and continue to unfold the loop with each newly discovered counterexample while iteratively refining $\wp\langle \ell, \psi \rangle$. Note that finding one witness is not sufficient to satisfy an existential property, as $\wp\langle \ell, \psi \rangle$ must characterize *all* the states satisfying the sub-property ψ at a location. Thus,

$$\wp\langle \ell, \psi \rangle = \bigvee_{n\in\mathbb{N}} pre(\mathsf{CEX_n})$$

Upon the return of our precondition method to its caller, $\wp$ will contain the precondition for the most outer temporal property of the original CTL property φ.

In our procedure, divergence can occur due to the possibility of generating infinitely many counterexamples. In practice this is rare, but not unheard of. We thus implement the following heuristic introduced by [10]:

- If we suspect we are looking at a sequence of repeated counterexamples that will result in divergence, we call the procedure STRENGTHEN (Figure 9, line 5 in PROPAGATE). The sub-procedure strengthens the candidate precondition towards the limit.
- STRENGTHEN takes the calculated pre-image α, then proceeds to quantify out all variables that are updated proceeding the program location ℓ by applying quantifier elimination (QE).
- This heuristic can lead to unsoundness, as STRENGTHEN may over-approximate the set of states, causing $\wp$ to be potentially unsound for temporal properties involving existential path quantifiers. To check that the guessed candidate precondition is in fact a real precondition, *e.g.* that $\wp \Rightarrow \mathsf{EG}\ \wp'$, we can use the approach from Beyene *et al.* [4] to double check the small lemma.
- If the check succeeds we continue, if the check fails we raise an exception.

Reducing redundant and irrelevant reasoning. Given that our approach synthesizes counterexample guided preconditions over program locations, we utilize sequential locality to simultaneously calculate preconditions for the set of locations that are arranged and can be accessed from a CEX starting from a given location ℓ. Our propagation sub-procedure PROPAGATE (Figure 8) is called from TEMPORALWP at line 21. We iterate along the counterexample path, and for every reachable location $\ell \in \mathcal{L}$, we compute a pre-image utilizing a suffix of CEX from ℓ onwards. In more informal terms, every program location along the path can utilize the same counterexample to show that the property does or does not hold. Practically, the computation of a pre-image is performed by going backwards

over the counter example.

PROPAGATE alone does not eliminate redundant or irrelevant reasoning, as we would still iterate over locations whose preconditions have already been computed for. We thus calculate a cut-point set C such that $C \subseteq \mathcal{L}$ and every cycle in the program's graph contains at least one cut-point (line 16 in TEMPORALWP). That is, we only wish to synthesize a precondition over each program location $\ell \subseteq C$. We choose to verify the set of cut-points [18] instead of all program locations, as cut-points provide locality across program locations given the nature of cycles. We will thus be able to propagate a cut-point precondition to all locations reachable from a cycle of a generated counterexample. Other program analysis inspired techniques may be used for the selection of initial locations to be verified. A cycle independent analysis can be run for those locations unreachable from program cut-points.

We now state the correctness of our procedure. A full correctness proof is included in Appendix.

Proposition 4.1: If the algorithm in Figure 5 terminates, for every sub-formula ψ of φ, every location $\ell \in \mathcal{L}$, and every reachable state s we have: $s \models \wp\langle\ell, \psi\rangle$ implies $P, (\ell, s) \models \psi$ and $s \models \neg\wp\langle\ell, \psi\rangle$ implies $P, (\ell, s) \models \neg\psi$.

Proof Sketch: We prove the proposition by induction on the structure of the formula. It is clear for an atomic proposition and for Boolean operators. Consider a universal path formula. As the counter examples obtained from the underlying program analysis tool are real counter examples, it follows that their pre-images do not satisfy the formula. We then get additional counter examples, which are all sound. The termination of the loop searching for counter examples implies that the disjunction of all pre-images is sound and complete. Existential path formulas are dual.

Corollary 4.2: For every symbolic program P we have $P \models \varphi$ iff for every $(\ell_0, \rho, \ell) \in E$ we have $\rho \Rightarrow \wp\langle\ell, \varphi\rangle$.

V. EVALUATION

In this section we discuss the results of our experiments with an implementation of the procedure from Figure 4. Our implementation is built as an extension to the open source project T2 [1], which uses a safety prover similar to IMPACT [28] alongside previously published techniques for discovering ranking functions, etc [31], [21] to prove both liveness and safety properties. The source code for our tool can be found at `http://heidyk.com/T2source/`.

We have compared our tool to that of Cook & Koskinen [10] and Beyene *et al.* [4]. The benchmarks used are the same as those used in [10] and [4]. These benchmarks were originally created by Cook & Koskinen using the examples drawn from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the SoftUpdates patch system [23]. The benchmarks can be found at `http://www.cims.nyu.edu/~ejk/ctl/`. For each program and CTL property φ, we verify both φ and $\neg\varphi$. The various tools were executed on an Intel x64-based 2.8 GHz single-core processor.

Program commands. We now discuss the format in which we interpret a program's commands. A deterministic assignment statement of the form $v' = \mathsf{exp}$ where $v' \in \mathsf{Vars}'$ and exp is an expression over program variables is translated to the condition $v' = \mathsf{exp} \ \wedge \ \forall x \in \mathsf{Vars}\backslash\{v\}.\ x = x'$. A nondeterministic assignment $v' = *$ is translated to $\forall x \in \mathsf{Vars}\backslash\{v\}.\ x = x'$. A conditional statements exp is encoded to $\mathsf{exp} \wedge \forall x \in \mathsf{Vars}.\ x = x'$.

In Figure 11 we display the comparison of our results. For each program and its set of CTL properties, we display the number of lines of code (LOC), and report the time it took to verify a CTL property (Time column) in seconds. A ✓ in the "Result" column indicates that the tool was able to verify the property. Likewise, an × indicates that the tool failed to prove the property. A timeout or memory exception is indicated by T/O. A timeout is triggered if verification of an experiment exceeds 3000 seconds. The symbol "–" in the Time and Result column indicates that the experiment was not run.

Overall, our tool demonstrates a significant increase in performance and scalability. Contrary to existing tools, our tool produces no timeouts and programs are often verified in under a second or less. The aforementioned tools often take minutes (the former more-so than the latter). Furthermore, the previous tools produce T/Os in cases where the temporal formula is complex, the size of the program is large, or both. Although a few of our results are on par with Beyene *et al.*, one can speculate from our evaluations that said tool is not well equipped to handle larger programs. Contrarily, our tool demonstrates the potential for scalability. On average, we show orders-of-magnitude performance improvement over existing tools, particularly when dealing with larger programs.

In a few cases our tool produces results that differ with one of the previous tools, due to bugs in the previous tools. As an example, in the S/W Updates case we are unable to repeat the result of Cook & Koskinen on $c > 5 \ \wedge \ \mathsf{AG}(r \leq 5)$ and $c > 5 \ \wedge \ \mathsf{EG}(r \leq 5)$. Our result agrees with that of Beyene *et al.*. Finally, OS frag. 2 requires fairness, and it is unclear how [10] and [4] verified said program, as all these tools lack support for fairness. Cook & Koskinen acknowledge their bug.

VI. CONCLUDING REMARKS

In this paper we have described a procedure for CTL model checking that takes advantage of the structure of control-flow graphs available in programs. Our procedure works recursively on the structure of the property and computes (location-based) preconditions for the satisfaction of each sub-formula. The idea is to use a decomposition based on program-location (thus facilitating the use of program analysis techniques), but to maintain the current state of the intermediate lemmas in a way their results can be used to quickly facilitate the computation of results for nearby program locations. As is evident from the outcome of our experimental evaluation, our method leads to dramatic performance improvement over competing tools that support CTL verification for infinite-state programs. Additionally, we wish to further experiment with the scalability that our methodology can perhaps provide.

REFERENCES

[1] "T2 source code," http://research.microsoft.com/t2.

			φ						$\neg\varphi$					
			Fig. 4		Beyene [4]		Cook [10]		Fig. 4		Beyene [4]		Cook [10]	
Program	**LOC**	**Property** (φ)	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**	**Time**	**Result**
OS frag. 6	1050	$\mathsf{AG}(b = 1 \Rightarrow \mathsf{AF}(u = 0))$	67.3	✓	T/O	–	T/O	–	82.9	×	T/O	–	T/O	–
OS frag. 6	1050	$\mathsf{EG}(b = 1 \Rightarrow \mathsf{EF}(u = 0))$	36.2	✓	T/O	–	T/O	–	38.8	×	T/O	–	–	–
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	5.9	✓	43.4	✓	38.9	✓	6.2	×	40.4	×	18.0	×
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	6.8	✓	35.45	✓	90.0	✓	3.4	×	36.57	×	107.3	×
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	4.7	✓	T/O	–	T/O	–	3.1	×	T/O	–	T/O	–
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	2.3	✓	2.52	✓	1680.7	✓	6.0	×	2.52	×	1930.0	×
OS frag. 4	370	$\mathsf{AF}(io = 1) \vee \mathsf{AF}(ret = 1)$	18.5	✓	270.6	✓	34.3	✓	13.9	×	58.06	×	18.8	×
OS frag. 4	370	$\mathsf{EG}(io \neq 1) \wedge \mathsf{EG}(ret \neq 1)$	13.5	✓	T/O	–	7.6	✓	14.2	×	T/O	–	61.3	×
OS frag. 4	370	$\mathsf{EF}(io = 1) \wedge \mathsf{EF}(ret = 1)$	14.7	✓	T/O	–	1261.0	✓	4.8	×	T/O	–	T/O	–
OS frag. 4	370	$\mathsf{AG}(io \neq 1) \vee \mathsf{AG}(ret \neq 1)$	8.0	✓	0.1	✓	–	–	3.7	×	1.3	×	–	–
PgSQL arch	90	$\mathsf{AG}(\mathsf{AF}(w = 1))$	2.0	✓	0.7	✓	T/O	–	1.3	×	1.4	×	38.1	×
PgSQL arch	90	$\mathsf{AG}(\mathsf{EF}(w = 1))$	2.0	✓	0.7	✓	T/O	–	0.0	×	0.2	×	42.7	×
PgSQL arch	90	$\mathsf{EF}(\mathsf{AG}(w \neq 1))$	2.0	✓	0.1	✓	T/O	–	2.4	×	0.7	×	30.2	×
PgSQL arch	90	$\mathsf{EF}(\mathsf{EG}(w \neq 1))$	0.1	✓	0.1	✓	35.2	✓	0.1	×	0.5	×	45.3	×
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{AF}(u = 1))$	0.8	×	1.4	✓	2.1	✓	0.2	✓	0.7	×	1.8	×
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{EF}(u = 1))$	2.0	×	1.3	✓	3.7	✓	0.2	×	0.4	×	1.5	×
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{AG}(u \neq 1))$	0.8	✓	0.1	✓	5.6	✓	0.2	×	0.7	×	8.7	×
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{EG}(u \neq 1))$	1.0	✓	0.1	✓	1.2	✓	1.2	×	1.8	×	6.5	×
OS frag. 5	58	$\mathsf{AG}(\mathsf{AF}(w \geq 1))$	1.0	✓	0.6	✓	569.7	✓	0.2	×	0.4	×	65.1	×
OS frag. 5	58	$\mathsf{AG}(\mathsf{EF}(w \geq 1))$	1.0	✓	0.7	✓	T/O	–	0.0	×	0.1	×	T/O	–
OS frag. 5	58	$\mathsf{EF}(\mathsf{AG}(w < 1))$	0.1	✓	0.5	✓	255.8	✓	0.1	×	0.2	×	85.5	×
OS frag. 5	58	$\mathsf{EF}(\mathsf{EG}(w < 1))$	0.1	✓	0.1	✓	351.1	✓	0.0	×	0.2	×	1471.7	×
S/W Updates	36	$c > 5 \Rightarrow \mathsf{AF}(r > 5)$	0.1	×	5.27	✓	70.2	✓	1.1	✓	0.8	×	32.4	×
S/W Updates	36	$c > 5 \Rightarrow \mathsf{EF}(r > 5)$	0.1	✓	0.2	×	18.5	✓	0.8	×	0.1	×	1.3	×
S/W Updates	36	$c > 5 \wedge \mathsf{AG}(r \leq 5)$	0.1	×	0.1	×	0.3	✓	1.1	✓	0.1	×	0.5	×
S/W Updates	36	$c > 5 \wedge \mathsf{EG}(r \leq 5)$	0.4	×	0.1	×	4.5	✓	0.7	✓	0.1	×	0.4	×
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	1.0	✓	0.3	✓	4.6	✓	1.4	×	0.7	×	9.1	×
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	0.1	✓	0.3	✓	9.5	✓	0.1	×	0.3	×	1.5	×
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	0.1	✓	0.1	✓	105.7	✓	0.1	×	0.4	×	18.1	×
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	0.1	✓	0.1	✓	3.5	✓	0.7	×	0.3	×	12.5	×

Fig. 11: The results of applying our CTL model checking procedure on benchmarks from [10], [4]. For each program we verify a set of properties (φ) and their negations ($\neg\varphi$) and compare our results with [10], [4]. A timeout (T/O) is triggered if verification of a benchmark exceeds 3000 seconds.

[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, 126(2), 1994.

[3] O. Bernholtz, M. Y. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking (extended abstract)," in *CAV'94*. Springer, 1994.

[4] T. A. Beyene, C. Popeea, and A. Rybalchenko, "Solving existentially quantified horn clauses," in *CAV'13*. Springer, 2013.

[5] J. Burch, E. Clarke *et al.*, "Symbolic model checking: 10^{20} states and beyond," *Information and computation*, 98(2), 1992.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV'02*. Springer, 2002.

[7] E. Clarke, S. Jha, Y. Lu, and H. Veith, "Tree-like counterexamples in model checking," in *LICS*, 2002.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *TOPLAS*, 1986.

[9] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop on Logic of Programs*, Springer, 1981.

[10] B. Cook and E. Koskinen, "Reasoning about nondeterminism in programs," in *PLDI'13*. ACM, 2013.

[11] B. Cook, J. Fisher, E. Krepska, and N. Piterman, "Proving stabilization of biological systems," in *VMCAI'11*. Springer, 2011.

[12] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi, "Proving that programs eventually do something good," in *POPL'07*, 2007.

[13] B. Cook and E. Koskinen, "Making prophecies with decision predicates," in *POPL'11*. ACM, 2011.

[14] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI'06*. ACM, 2006.

[15] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1(1), 1959.

[16] E. A. Emerson and K. S. Namjoshi, "Automatic verification of parameterized synchronous systems (extended abstract)," in *CAV'96*. Springer, 1996.

[17] J. Esparza, A. Kucera, and S. Schwoon, "Model checking ltl with regular valuations for pushdown systems," *Information and Computation*, 186, 2003.

[18] R. W. Floyd, "Assigning meaning to programs," in *Mathematical Aspects of Computer Science*, ser. Proc. of Symposia in Applied Mathematics. American Mathematical Society, 1967.

[19] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning*, Springer, 2000.

[20] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that non-blocking algorithms don't block," in *POPL'09*. ACM, 2009.

[21] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," *SIGPLAN Not.*, 43, 2008.

[22] A. Gurfinkel, O. Wei, and M. Chechik, "Yasm: A software model-checker for verification and refutation," in *CAV'06*. Springer, 2006.

[23] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *VSTTE'12*, 2012.

[24] Y. Kesten and A. Pnueli, "A compositional approach to ctl* verification," *Theor. Comput. Sci.*, 331(2-3), 2005.

[25] O. Kupferman, M. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," *Journal of the ACM*, 47(2), 2000.

[26] S. Magill, J. Berdine, E. M. Clarke, and B. Cook, "Arithmetic strengthening for shape analysis," in *SAS'07*. Springer, 2007.

[27] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety*. Springer Verlag, 1995, vol. 2.

[28] K. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.

[29] H. Peng, Y. Mokhtari, and S. Tahar, "Environment synthesis for compositional model checking," in *Computer Design: VLSI in Computers and Processors*, 2002.

[30] A. Podelski and A. Rybalchenko, "Transition invariants," in *LICS'04*. IEEE, 2004.

[31] ——, "Transition invariants," in *LICS*, 2004.

[32] F. Song and T. Touili, "Pushdown model checking for malware detection," in *TACAS'12*. ACM, 2012.

[33] I. Walukiewicz, "Pushdown processes: Games and model checking," in *CAV'96*. Springer, 1996.

[34] ——, "Model checking ctl properties of pushdown systems," in *FSTTCS'00*. Springer, 2000.