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 many model checking tools that support temporal logic, performance is hindered by redundant reasoning performed in the presence of nested temporal operators. In particular, tools supporting the state-based temporal logic CTL often symbolically partition the system's state space using the sub-formulae of the input temporal formula. This can lead to repeated work when tools are applied to infinite-state programs, as often the characterization of the state-spaces for nearby program locations are similar and interrelated. In this paper, we describe a new symbolic procedure for CTL verification of infinite-state programs. Our procedure uses the structure of the program's control-flow graph in combination with the nesting of temporal operators in order 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 speed improvement, but allows for scalability of temporal reasoning for larger programs.

1. Introduction

Branching-time temporal logics allow us to reason about a system's interaction with inputs and nondeterminism in a way that linear-time temporal logics do not. Such reasoning is crucial to applications including planning, games, security analysis, disproving, environment synthesis, and many others. Unfortunately, the search for scalable and high-performance temporal-logic proof tools for infinite-state programs remains an open problem. The problem with existing tools is that performance is often hindered by redundant reasoning performed in the presence of nested temporal operators. For example, tools supporting the state-based temporal logic CTL [8] invariably recurse over the structure of the input property and reason about the sets of system states that respect the various sub-formulae. Consider a property such as AG ($x<y \Rightarrow$ (EF $y<z$)), which states that that whenever $x<y$, then it is possible that eventually $y<z$. When checking that this property holds of the input program, the tool from Beyene *et al.* [4] solves for constraints characterizing the sets of states respecting each of the sub-formulae, *i.e.* it computes representations of the "$x<y$"-states, the "$y<z$"-states, the "EF $y<z$"-states, the "$x<y \Rightarrow$ (EF $y<z$)"-states, etc. Other tools supporting CTL do the same, *e.g.* [4, 6, 7, 11].

In this paper, we describe a new symbolic CTL model checker for infinite-state programs. Our approach makes use of the structure of the program's control-flow graph during a bottom-up analysis over the property's structure. Our approach reduces the amount of reasoning perfomed as part of the procedure, suggesting that current competing tools, *e.g.* [4, 11] perform redundant reasoning. Our strategy makes use of the fact that the set of states that respect a property such as EF $y<z$ *before* a program command is very often the same or nearly the same as the set of states respecting EF $y<z$ *after* the command.

Our method leads to dramatic performance improvements and allows for scalability to larger programs: An experimental evaluation using examples from the benchmark suites of the competing tools (which are drawn from industrial benchmarks) demonstrates orders-of-magnitude performance improvement in many cases.

Related work. Model checking has been extensively studied in the context of finite-state systems (*e.g.* [3, 5, 9, 10, 22]) as well as for various types of systems with limitations on the infiniteness (*e.g.* pushdown systems [16], parameterized systems [15], etc). In recent years powerful new tools have been developed for proving temporal properties of full-blown infinite-state programs, *e.g.* [4, 11, 13, 19, 28–30].

In this work we are aiming to prove CTL properties with nested combinations of existential and universal path quantifiers of infinite-state programs. A number of CTL model checking tools for programs do not meet these criteria. For example, SMV (and in general BDD based tools) are restricted to finite-state programs [7]. Song & Touili [28] perform a coarse one-time abstraction to pushdown automata, and Gurfinkel *et al.* [19] 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 [11] and Beyene *et al.* [4]. Cook & Koskinen essentially implement the Kesten and Pnueli [21] 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. The problem with these tools is that they perform redundant reasoning when proving nested temporal properties. For example, Cook & Koskinen and Beyene *et al.* walk recursively over the structure of the temporal formula, whereas we look up the current precondition in what would be the recursive case. This prevents an exponential state explosion which would be caused otherwise by nesting the recursive verification of each sub-property contained within a CTL property. We also introduce a new approach to the treatment of existential path quantification based on dualization. This is in contrast to 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 from Beyene *et al.* [4], as it uses existential quantification techniques in the underlying constraint-solver.

Limitations. While perhaps our approach could be adapted to work in the general setting of recursive heap-manipulating concurrent programs, we are currently limited to sequential non-recursive programs that use commands expressed in linear-arithmetic. The programs we consider during our experimental evaluation have been abstracted from heap-manipulating programs using the technique of Magill *et al.* [23].

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

2. Preliminaries

First, we begin by briefly discussing our formal representation of programs and methods, as well as CTL.

Programs. As is standard [24], 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 labeled triples, and Vars is a set of variables. Each triple $\tau : (\ell, \rho, \ell')$ in E specifies possible transitions in the program. The condition ρ is an assertion in terms of Vars and Vars', a primed copy of Vars. 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 initial location ℓ_0 that has no incoming transitions from any prior 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 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 $s_1, s_2 \models \rho$. A state (ℓ, s) is initial if there is a transition (ℓ_0, ρ, ℓ) such that $\emptyset, s \models \rho$. See Figure 2 for an example representation of the program **while** $\mathsf{x} \leq 0$ **do** $\mathsf{x} := \mathsf{x} + 1$; **done**; $\mathsf{y} := 1$; with initial condition $\mathsf{x} = 0 \wedge \mathsf{y} = 0$. A *trace* or a *path* of a program is either a finite or an infinite sequence of program states allowed by the program.

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

CTL. We are interested in verifying state-based temporal properties in the form of CTL [8] (*a.k.a.* Computational tree logic). 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.* $\mathsf{x} < \mathsf{y}$).

Here $P, s \models \mathsf{AF}\varphi$ asserts that φ will eventually hold in a future reachable state from s, whereas $\mathsf{EF}\varphi$ asserts that φ can potentially eventually hold in a future reachable state. $\mathsf{AG}\varphi$ 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. AU and EW are represented as syntactic sugar as usual.

CTL entailment. For a transition system T and a CTL property φ, we say that φ holds in T, denoted by $T \models \varphi$ if $\forall s \in I.R, s \models \varphi$.

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 finite set of ranking functions for a relation R is equivalent to containment of R within a finite union of well-founded relations [27]. 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 4). A counterexample CEX in ACTL is defined follows:

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

where π is a trace through the transformed program. We only require counterexamples in ACTL as recall that existential formulas utilize counterexamples acquired from their (universal) dual. Note that often tools will not report a concrete trace but rather a *path*, *i.e.* a sequence relations and program counter values corresponding to a set of traces. The counterexample structure for an atomic proposition CEX_α is simply a state in which α does not hold. Counterexamples for conjunction and disjunction are as expected. A counterexample to an AG property is a path to a place where there is a counterexample to the sub-property. A counterexample to an AF property is a "lasso": a stem path to a particular program location, then a cycle which returns to the same program location, and a sub-counterexample along that cycle in which the sub-property does not hold. Finally, an AW counterexample is a path to a place where there is a sub-counterexample to the first property as well as a sub-counterexample to the second property.

Calculating weakest preconditions. As is standard in [14], for any statement T and a postcondition Q, the weakest precondition of T with respect to Q, denoted by $wp(T, Q)$, characterizes all preconditions such that the initial state ensures that the execution of T terminates in a final state satisfying Q. For our generated counterexamples, we employ a recursive definition of weakest preconditions for unstructured programs proposed by [2] as follows:

$$wp(\mathsf{assert}\ P, Q) = P \wedge Q$$
$$wp(\mathsf{assume}\ P, Q) = P \Rightarrow Q$$
$$wp(S; T, Q) = wp(S, wp(T, Q))$$

In our case, T is a relation corresponding to program counter values.

3. Examples

Before we formally describe our CTL proving procedure, we first informally demonstrate a few key ideas with illustrative examples. Later on we will provide an in-depth account of these examples in Section 5.

Sharing intermediate results. Our first example illustrates how we can avoid redundant computation while reasoning about subformulae of a temporal formula. Consider the program in Figure 1. Imagine that our goal is to prove the property $\mathsf{AFAG}\ \mathsf{x}{=}0$. This property states that, for all initial states, the program will definitely reach a state wherein $\mathsf{x}{=}0$ holds forever. The approach followed by nearly all tools supporting CTL would be to find, in this instance, a set of states $\wp$ such that $\mathsf{AF}\wp$ holds, and such that $\wp \models \mathsf{AG}\ \mathsf{x}{=}0$

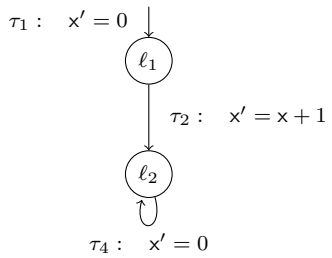

Figure 1: The control-flow graph of an example program for which we wish to prove the CTL property $\mathsf{AFAG}\ \mathsf{x} = 0$.

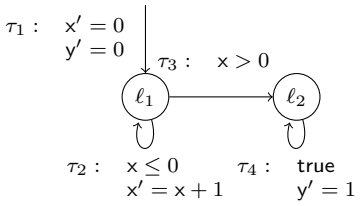

Figure 2: The control-flow graph of an example program for which we wish to prove the CTL property $\mathsf{AGEF}\ \mathsf{y} = 1$.

holds. In our approach we use a precondition synthesis based strategy: we initially let $\wp \triangleq \mathsf{true}$ and attempt to prove $\wp \models \mathsf{AG}\ \mathsf{x}{=}0$. Failures to the proof attempt will result in refinements to $\wp$ through the iterative calculation of the negated weakest precondition of each discovered counterexample. It is trivial to prove $\wp \models \mathsf{x} = 0$. Given the simplicity of our example, the weakest precondition happens to be equivalent to the atomic proposition itself. Hence from this point on, we shall treat the weakest precondition of the atomic proposition and the atomic proposition itself synonymously.

Eventually this will result in $\wp \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \mathsf{false}) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} = 0)$ after the elimination of counterexamples. We then repeat the same procedure and proceed to prove $\mathsf{AF}\ (\mathsf{pc} = \ell_1 \Rightarrow \mathsf{false} \wedge \ell_2 \Rightarrow \mathsf{x} = 0)$, thus proving that the original formula $\mathsf{AFAG}\ \mathsf{x}{=}0$ holds.

We operate by iteratively proving lemmas at each program location. This facilitates the use of high performance program analysis techniques. That is, we are attempting to find $\wp$'s for each location in the program: $\wp_{\ell_1}$, and $\wp_{\ell_2}$. Thus in our tool $\wp$ will be defined as

$$\wp \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \wp_{\ell_1}) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \wp_{\ell_2})$$

where $\mathsf{pc} = \ell_1$ is used to assert that the state is at location ℓ_1 in the program's control-flow graph. Notice that in our example, $\wp_{\ell_1} \triangleq \mathsf{false}$ and $\wp_{\ell_2} \triangleq x = 0$. The problem with the tools from Cook & Koskinen [11] or Beyene *et al.* [4] is that they spend time computing both $\wp_{\ell_1}$, $\wp_{\ell_2}$. In our approach we attempt to prove and refine $\mathsf{AG}\ \mathsf{x} = 0$ at location ℓ_1, but we are careful to adapt the refinement for ℓ_2 simultaneously. Thus we can avoid the redundant reasoning.

Existential path formulae. In order to support existential path formulae (*e.g.* EG) we use a strategy that eliminates counterexamples to the negation of the existential property using additional constraints to the precondition.

Consider the program in Figure 2. Imagine that we are trying to prove the property $\mathsf{AGEF}\ \mathsf{y}{=}1$. This property states that, for all states, it is always possible that eventually $\mathsf{y}{=}1$. As before, we are looking for a $\wp$ such that $\mathsf{AG}\wp$ holds for the program such that

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

Figure 3: CTL model checking procedure VERIFY, which utilizes the subroutine in Figure 4 to generate weakest preconditions for temporal properties.

```
1  let rec TEMPORALWP(ψ, P) : map =
2    ℘ = INIT (ψ)
3    M = ∅
4    (L, R, Vars) = P
5    if ψ = α  is atomic  then
6      foreach {ℓ | (ℓ, t, ℓ') ∈ R
7          ℘⟨ℓ, ψ⟩ = wp(t, α); ℘⟨ℓ, ¬ψ⟩ = ¬wp(t, α)
8    else
9      match (ψ) with
10     | ψ'1 ∧ ψ'2
11     | ψ'1 ∨ ψ'2
12     | ψ'1 U ψ'2
13     | ψ'1 W ψ'2 →
14         ℘ = ℘ ∪ TEMPORALWP(ψ'1, P) ∪ TEMPORALWP(ψ'2, P)
15     | AFψ'1 | AGψ'1 | ¬ψ'1 →
16         ℘ = ℘ ∪ TEMPORALWP(ψ'1, P)
17     C = findCutPoints(P)
18     foreach ℓ ∈ C  do
19        P' = TRANSFORM(⟨ℓ, ψ⟩, M, P, ℘)
20        CEX, M = REFINE(P', ψ, ℘, ℓ0, M, ERR)
21        while CEX ≠ ∅ do
22          α = wp(ℓ, CEX)
23          foreach (s, tn, s') ∈ CEX reachable from ℓ do
24            if ψ = Eψ' then
25                ℘⟨s, ψ⟩ = ℘⟨s, ψ⟩ ∨ wp(s, CEX)
26                ℘⟨s, ¬ψ⟩ = ℘⟨s, ¬ψ⟩ ∧ ¬wp(s, CEX)
27            else
28                ℘⟨s, ψ⟩ = ℘⟨s, ψ⟩ ∧ ¬wp(s, CEX)
29                ℘⟨s, ¬ψ⟩ = ℘⟨s, ¬ψ⟩ ∨ wp(s, CEX)
30          foreach {'ℓ | ('ℓ, t, ℓ) ∈ R'} do
31             t = t ∧ assume(¬α)
32          CEX, M = REFINE(P', ψ, ℘, ℓ0, M, ERR)
33
34   ℘
```

Figure 4: Precondition synthesis procedure utilizing sequential locality. Two parameters are given: a program and a sub-property. The procedure returns a function that maps sub-properties to their synthesized preconditions. A precondition of a CTL sub-property is automatically synthesized from counterexamples and then is successively replaced by a condition over program states.

$\wp \models \mathsf{EF}\ \mathsf{y}{=}1$ holds. Our attempts to prove $\wp \models \mathsf{EF}\ \mathsf{y}{=}1$ are via negation: we attempt to prove $\wp \not\models \mathsf{AG}\ \mathsf{y}{\neq}1$. We thus must start with $\wp \triangleq \mathsf{false}$ 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 weakest precondition of each discovered counterexample. In this case weakest preconditions to the failure are disjuncted onto $\wp$. Each counterexample serves as a subset of the existential witness for $\mathsf{EF}\ \mathsf{y}{=}1$. The successive discovery of all possible counterexamples indicates the existential witness of $\wp \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \mathsf{x} \geq 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \mathsf{x} \geq 0)$

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

Figure 5: An existing safety prover similar to IMPACT is employed as a reachability checker to ERR. A ranking function refinement procedure then follows for when proving liveness (AF), where the input transition system P is assumed to be a program.

```
1  let INIT (ψ) : map =
2
3    ℘ = ∅
4    if ψ = Eψ′ then
5      foreach ℓ ∈ L do
6        ℘⟨ℓ, ψ⟩ = false; ℘⟨ℓ, ¬ψ⟩ = true
7    else
8      foreach ℓ ∈ L do
9        ℘⟨ℓ, ψ⟩ = true; ℘⟨ℓ, ¬ψ⟩ = false
10   return ℘
```

Figure 6: Preconditions of universal CTL formulas are initialized to true as counterexamples are utilized to strengthen the initial condition. Given that existential formulas are handled by considering their universal dual, counterexamples serve as a witness thus weakening the initial condition false.

4. Procedure

In this section we describe the details of our CTL model checking procedure. See Figure 3, which depicts the main procedure, VERIFY. Figure 4 defines the subroutine TEMPORALWP used in VERIFY, while Figures 5, 6, and 7 are subroutines used in TEMPORALWP.

In our approach the table $\wp$ is the key data structure which maps pairs of program locations and sub-formulae to formulae which represent the current candidate *precondition* that would guarantee the property at that location. That is, our hope is that $\wp\langle\ell, \varphi\rangle$ is a sufficient 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 $\wp$ given the counterexample path. A brief description of each of the subroutines is provided below, followed by more details pertaining to TEMPORALWP.

TEMPORALWP performs both a recursive and a refinement-based computation to construct $\wp$. We recursively enumerate over each sub-property wherein a corresponding precondition is synthesized over program locations to be stored in $\wp$, asserting the satisfaction of the aforementioned sub-property. In TEMPORALWP, the procedure INIT in 6 initializes the precondition for a CTL formula such that it could be iteratively refined with each counterexample acquired. When TEMPORALWP returns to VERIFY, it is then only necessary to check if the precondition of the most outer temporal sub-property is satisfied by the initial states of the program.

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

Figure 7: A source-to-source transformation that reduces the checking of temporal properties to safety through finding nested conditions for sub-properties from the function $\wp$. Existential quantifiers are handled by considering their (universal) dual.

TRANSFORM employs a source-to-source transformation that reduces the checking of temporal properties to safety and well-foundedness.

The TRANSFORM transformation utilizes the function $\wp$, which maps the preconditions synthesized previously for sub-properties and their negations (lines 6,9,13,23, and 30). 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 29), 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.

We support existential quantifiers by dualizing them; effectively allowing us to extend the transformation, which only handles universal quantifiers, to handling existential. Note that the precondition of a counterexample to a universal property corresponds as the precondition, thus a witness, of its existential dual. This will be discussed more extensively further below.

REFINE uses a safety prover (similar to IMPACT) to obtain counterexamples of the transformed system, if a counterexample exists. In case that the counter example to a liveness property (such as AF) contains a lasso fragment we may be able to find an accompanying set of ranking functions $\mathcal{M}$ that will show

that the counter example is not valid. We thus atempt to enlarge the set of ranking functions $\mathcal{M}$ using the well known method of [12].

We now discuss TEMPORALWP in some more detail.

4.1 TEMPORALWP: computing $\wp$

See Figure 4. In order to synthesize a precondition for a sub-property ψ, we first recursively accumulate the preconditions generated when considering the sub-properties of ψ at lines 7,14, and 16. We compute our base case, an atomic proposition α, as is standard in [2] wherein the weakest precondition of a given a transitional relation t is a function mapping any postcondition α to a precondition. Our transformation TRANSFORM then allows us to reduce the checking of temporal properties starting from the innermost temporal property.

We then 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 17). We wish to synthesize a precondition over these cut-points, hence we generate a transformed program corresponding to each cut-point using the subroutine TRANSFORM at line 19. Each transformed program is then verified through the subroutine REFINE (lines 20 and 32). Our counterexample-guided precondition refinement loop begins at line 18, where we iteratively refine a precondition for $C \subseteq \mathcal{L}$ until no more counterexamples are found. The preconditions for each $C \subseteq \mathcal{L}$ are then accumulated to define the precondition over program locations asserting the satisfaction of the aforementioned sub-property. We discuss the refinement process for each type of quantifier separately below.

We utilize sequential locality to simultaneously calculate several preconditions for the set of locations that are arranged and can be accessed from a CEX starting from the cut-point ℓ. That is, the computation of the precondition along the counterexample is used to instrument every reachable location from $\ell \in C$ along the counterexample. Our propagation loop begins at line 23, and iterates along the counterexample path. In more informal terms, every state along the path can utilize the same counterexample to show that the property does or does not hold.

We choose to verify the set of cut-points [17] instead of all program locations, as a sequential locality analysis allows us to simultaneously calculate several preconditions for a set of locations reachable from an attained counterexample. 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 contained within a cycle of a generated counterexample. This is discussed in more detail below. 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.

Universal precondition synthesis. For a universal CTL sub-property ψ, a precondition $\wp_{\ell,\psi}$ for a program location ℓ is initially true. If a counterexample is returned, we refine $\wp_{\ell,\psi}$ by taking the negation of weakest precondition of the counterexample returned at location ℓ. As shown on line 28, our precondition then becomes $\wp_{\ell,\psi} = true \wedge \neg wp(\ell, \mathsf{CEX})$.

Next, we will propagate a cut-point precondition to all locations contained within a cycle of a generated counterexample in lines 23-29. We then rule out the aforementioned counterexample by adding the assumption $\neg wp(\ell, \mathsf{CEX})$ at the cut-point program state as shown on line 31. Note that there may be multiple transitions leading to a program location, hence the assumption must be instrumented at each ingoing transition.

We then continue to unfold the loop whenever a new counterexample is discovered while iteratively refining $\wp_{\ell,\psi}$, resulting in the precondition

$$\wp_{\ell,\psi} = \bigwedge_{n \in \mathbb{N}} \neg wp(\mathsf{CEX}_\mathsf{n})$$

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 an 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 witness.

A precondition $\wp_{\ell,\psi}$ for a program state is initially false (line 5). If a counterexample is returned, $\wp_{\ell,\psi}$ is refined through the disjunction of the weakest precondition of the counterexample returned, that is $\wp_{\ell,\psi} = \mathsf{false} \vee wp(\ell, \mathsf{CEX})$ as shown on line 25. We then propagate the precondition to the set of locations reachable along the counterexample path, as described in the universal synthesis above from lines 23-29. We rule out the aforementioned counterexample by adding the assumption $\neg wp(\ell, \mathsf{CEX})$, and continue to unfold the loop with each newly discovered counterexample while iteratively refining $\wp_{\ell,\psi}$. Note that finding one witness is not sufficient to satisfy an existential property, as $\wp_{\ell,\psi}$ must characterize *all* the states satisfying the sub-property phi at a location. We thus have

$$\wp_{\ell,\psi} = \bigvee_{n \in \mathbb{N}} wp(\mathsf{CEX}_\mathsf{n})$$

The remaining resumes in the same manner as for universal properties. For both existential and universal properties, our mapping function is also updated with the precondition for the negation of the property on lines 7, 26, and 27. This allows us to conveniently access the negation of the property in Figure 7 when encoding existential properties as their universal duals. 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.

4.2 Soundness and relative completeness.

Notation. For a program P, a location ℓ, and a condition p over Vars, we denote by $P[p@\ell]$ the program obtained from P by splitting the location ℓ to ℓ^+ and ℓ^- and adding the condition p on all transitions entering ℓ^+ and adding the condition $\neg p$ on all transitions entering ℓ^-. That is, $P[p@\ell] = (\mathcal{L}', E', \mathsf{Vars})$, where $\mathcal{L}' = (\mathcal{L} - \{\ell\}) \cup \{\ell^+, \ell^-\}$, E' contains the following transitions, and p' is a primed copy of p.

- If $(\ell_1, \rho, \ell_2) \in E$ and $\ell_1, \ell_2 \neq \ell$ then $(\ell_1, \rho, \ell_2) \in E'$.
- If $(\ell_1, \rho, \ell) \in E$ and $\ell_1 \neq \ell$ then $(\ell_1, \rho \wedge p', \ell^+) \in E'$ and $(\ell_1, \rho \wedge \neg p', \ell^-) \in E'$.
- If $(\ell, \rho, \ell_2) \in E$ and $\ell_2 \neq \ell$ then $(\ell^+, \rho, \ell_2), (\ell^-, \rho, \ell_2) \in E'$.
- If $(\ell, \rho, \ell) \in E$ then $(\ell^*, \rho \wedge p', \ell^+), (\ell^*, \rho \wedge \neg p', \ell^-) \in E'$, where $* \in \{+, -\}$.

This transformation has two distinct locations representing ℓ, one where the precondition p does hold and one where the precondition p does not hold. The modified program has the same set of computations. This way we can reason about the correctness of our preconditions by considering the locations ℓ^+ and ℓ^-.

PROPOSITION 4.1. *If the algorithm in Figure 4 terminates, for every sub-formula ψ of φ and every location $\ell \in \mathcal{L}$ we have*

$$P[\wp_{\ell,\psi}@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi) \wedge \mathsf{AG}(\ell^- \Rightarrow \neg\psi)$$

Proof: We prove the proposition by induction on the structure of the formula. It is clear for an atomic proposition. For a non-atomic formula, as the counter examples obtained from the underlying program analysis tool are real counter examples, it follows that their

preconditions 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-conditions is sound and complete.

We now discuss the proof in more detail. Consider an atomic proposition α. By construction, for every location ℓ we have $\wp_{\ell,\alpha} = \alpha$ and $\wp_{\ell,\neg\alpha} = \neg\alpha$. Clearly, $P[\alpha@\ell] \models \mathsf{AG}(\ell^+ \rightarrow \alpha)$ and $P[\neg\alpha@\ell] \models \mathsf{AG}(\ell^+ \rightarrow \neg\alpha)$.

We now proceed by induction on the nesting depth of formulas.

- If $\psi = \psi_1 \vee \psi_2$. Consider a location ℓ. Denote $\wp_1 = \wp_{\ell,\psi_1}$ and $\wp_2 = \wp_{\ell,\psi_2}$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_1)$ and $P[\wp_2@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_2)$.
  Suppose that $P[\wp_\psi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \psi)$. Then, there is a state (ℓ', s) that is reachable in the program such that $(\ell', s) \not\models \ell^+ \Rightarrow \psi$. Clearly, $\ell' = \ell^+$.
  However, the encoding of ψ in Figure 7 adds the transition $(l, \neg\wp_1 \wedge \neg\wp_2, \texttt{ERR})$ to P. Then, the precondition synthesis terminates only when $\wp_{\ell,\psi}$ is strong enough to guarantee that ERR is not reachable in the modified program.
  However, it must be the case that the same state (ℓ, s) that serves as counter example to $\mathsf{AG}(\ell \Rightarrow \psi)$ would serve as a counter example to $\mathsf{EG}\neg\texttt{ERR}$ in the modified program.
  The dual argument (relating to l^-) is similar and thus omitted.
- If $\psi = \psi_1 \wedge \psi_2$ the proof is similar to the previous case.
- If $\psi = \mathsf{A}[\psi_1 \mathsf{W} \psi_2]$
  Consider a location ℓ. Denote $\wp_1 = \wp_{\ell,\psi_1}$ and $\wp_2 = \wp_{\ell,\psi_2}$. By induction, we know that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_i)$ and that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\psi_i)$.
  Suppose that $P[\wp_\psi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \psi)$. Then, there is a state (ℓ', s) that is reachable in the program such that $(\ell', s) \not\models \ell^+ \Rightarrow \psi$. Clearly, $\ell' = \ell^+$.
  However, the encoding of ψ in Figure 7 adds to every location ℓ' the transition $(\ell', \neg\wp_1 \wedge \neg\wp_2, \texttt{ERR})$ to P. It also changes every other transition to two transitions one augmented by $\wp_1 \wedge \neg\wp_2$ to the same target and one augmented by $\wp_2$ that leads to locations from where the error is no longer reachable. Then, the precondition synthesis terminates only when $\wp_{\ell,\psi}$ is strong enough to guarantee that ERR is not reachable in the modified program.
  The completeness of the case of $\mathsf{A}[\psi_1 \mathsf{W} \psi_2]$ follows from the proof of $\mathsf{E}[\psi_1 \mathsf{U} \psi_2]$ below.
- If $\psi = \mathsf{E}[\psi_1 \mathsf{U} \psi_2]$ This is the dual of $\mathsf{A}[\psi_1 \mathsf{W} \psi_2]$ above.
  Consider a location ℓ. Denote $\wp_1 = \wp_{\ell,\psi_1}$ and $\wp_2 = \wp_{\ell,\psi_2}$. By induction, we know that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_i)$ and that $P[\wp_i@\ell] \models \mathsf{AG})\ell^- \Rightarrow \neg\psi_i)$.
  Suppose that $P[\wp_\psi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \psi)$. Then, there is a state (ℓ', s) that is reachable in the program such that $(\ell', s) \not\models \ell^+ \Rightarrow \psi$. Clearly, $\ell' = \ell^+$.
  However, the encoding of ψ in Figure 7 treats EU as the dual of AW. Thus, it adds to every location ℓ a transition $(\ell, \neg\wp_1 \wedge \neg\wp_2, \texttt{ERR})$ and every other transition is replaced with two transitions one augmented by $\wp_1 \wedge \neg\wp_2$ leading to the same target and one augmented by $\wp_2$, which is then leading to a region where the transitions to ERR are no longer reachable. Then, the precondition synthesis extracts counter examples that reach the ERR state. Clearly, a path reaching ERR is a path that violates the dual AW and thus satisfies ψ. Thus, from every state satisfying the weakest precondition of this counter example the formula ψ holds.
- If $\psi = \mathsf{AF}\psi_1$
  Consider a location ℓ. Denote $\wp_1 = \wp_{\ell,\psi_1}$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_1)$ and that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\psi_1)$.
  Suppose that $P[\wp_\psi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \psi)$. Then, there is a state (ℓ', s) that is reachable in the program such that $(\ell', s) \not\models \ell^+ \Rightarrow \psi$. Clearly, $\ell' = \ell^+$.
  However, the encoding of ψ in Figure 7 adds a transition to ERR only in case that a loop is found that does not have a ranking function. It also adds a transition to SAFE in the case that $\wp_1$ holds. Otherwise, from every state either it duplicates the state and searches for a loop to that state or continues. From the soundness for this program analysis procedure for ACTL[1], we know that when the program analysis task returns that the system is safe. It follows that the precondition synthesized is strong enough to ensure that the error location is not reached implying that there are no loops where ψ_1 does not hold.
  The completeness of the case of $\mathsf{AF}\psi_1$ follows from the proof of $\mathsf{EG}\psi_1$ below.
- If $\psi = \mathsf{EG}\psi_1$.
  Consider a location ℓ. Denote $\wp_1 = \wp_{\ell,\psi_1}$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \psi_1)$ and that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\psi_1)$.
  Suppose that $P[\wp_\psi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \psi)$. Then, there is a state (ℓ', s) that is reachable in the program such that $(\ell', s) \not\models \ell^+ \Rightarrow \psi$. Clearly, $\ell' = \ell^+$.
  However, the encoding of ψ in Figure 7 treats EG as the dual of AF. Thus, it adds transitions to error whenever a loop is found that does not visit $\wp_1$. Then, the precondition synthesis extracts counter examples that reach the ERR state. Clearly, a path reaching ERR is a path that violates the dual AF and thus satisfies ψ. Thus, from every state satisfying the weakest precondition of this counter example the formula ψ holds.

□

COROLLARY 4.2. *For every symbolic program P we have $P \models \varphi$ iff for every $(\ell_0, \rho, \ell) \in E$ we have $\rho \Rightarrow \wp_{\ell,\varphi}$.*

PROPOSITION 4.3. *Our algorithm is relatively complete.*

Proof: We rely on the following algorithms. We assume that REFINE finds paths to the error state instrumented in the program. We assume that ranking functions that rule out counterexamples to liveness properties can be found (line 27 in Figure 7). We assume that the computed weakest preconditions (lines 21–28 in Figure 4) are accurate.

Relative completeness follows from the structure of counter examples. A counterexample analyzed by wp is not a trace of the program but rather a fragment of the control-flow graph. As such, the extracted precondition is a precondition for all traces that use the fragment. In particular, if the fragment contains a loop, an extracted precondition would correspond to all possible traces through the loop. It follows from the finiteness of the control-flow graph of the program the number of different counterexamples that can be found by REFINE is finite (corresponding to the number of subgraphs of the control-flow graph).

Other sources of possible non-termination of our procedure are the well-foundedness checking and the search performed by REFINE. If these are assumed to be complete then our technique is complete as well. □

5. Examples in detail

We now walk through the two examples from Figures 1 and 2 of Section 3 in detail, connecting them to our procedure from the preceding section.

[1] A formula is in ACTL if the only temporal operators it uses are universal, i.e., AX, AW, AF, AU, or AG.

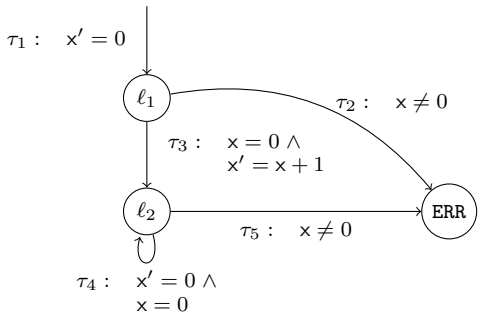

Figure 8: The transformation of the program in Figure 1 for the sub-property AG x=0 which is utilized in the verification algorithm.

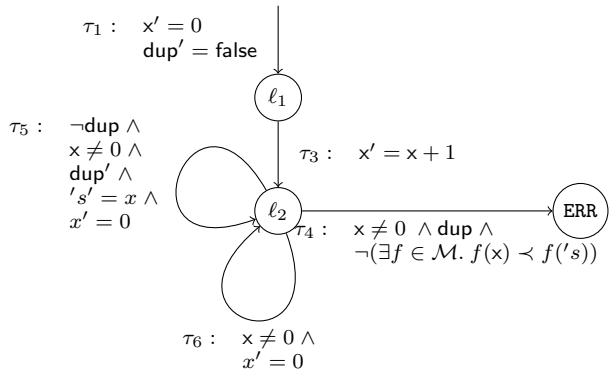

Figure 9: The transformation of the program from Figure 1 for the sub-property AFAG x=0 to be utilized in the verification algorithm. The nested property AG x = 0 is substituted with its precondition resulting in a transformation for AF (pc = ℓ_1 ⇒ false ∧ ℓ_2 ⇒ x = 0) instead.

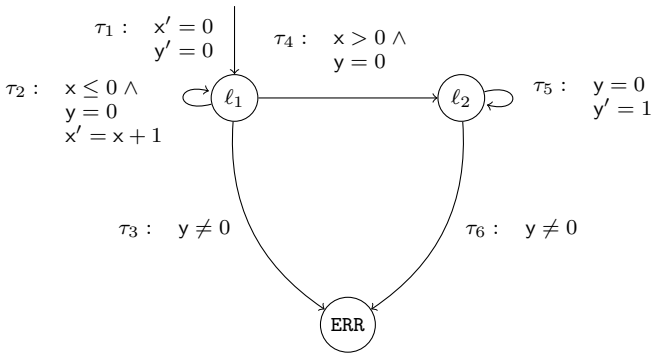

Figure 10: The transformation of the program from Figure 2 for the sub-property EF y = 1 using its dual AG y = 0 which is utilized in the verification algorithm.

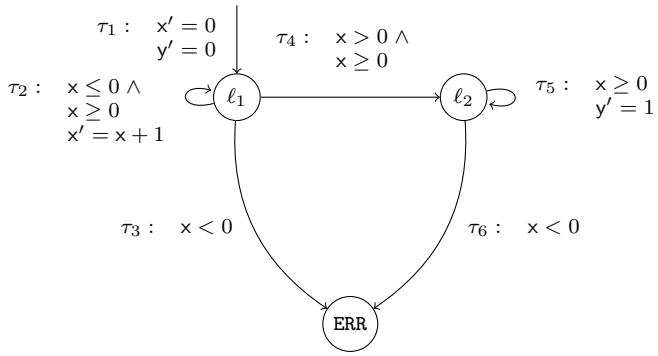

Figure 11: The transformation of the program from Figure 2 for the sub-property AGEF y =1 to be utilized in the verification algorithm. The nested property EF y = 1 is substituted with its precondition resulting in a transformation for AG (pc = ℓ_1 ⇒ x ≥ 0 ∧ pc = ℓ_2 ⇒ x ≥ 0) instead.

Sharing intermediate results. Recall the program in Figure 1. We show how the algorithm establishes that the CTL property AFAG x=0 holds for the program. We synthesize the preconditions starting from sub-properties. We start with the atomic proposition $\mathsf{x} = 0$ and then the temporal sub-property AG x=0.

On lines 5-7 in Figure 4 we trivially obtain the weakest precondition for the atomic proposition $\mathsf{x} = 0$. Given the simplicity of our example, the weakest precondition happens to be equivalent to the atomic proposition itself. Hence from this point on, we shall treat the weakest precondition of the atomic proposition and the atomic proposition itself synonymously.

Let $\wp_{\mathsf{AG\ x=0}}$ be the precondition for the sub-property AG x=0. A precondition for this sub-property at a certain program location ℓ is denoted by $\wp_{\ell,\mathsf{AG\ x=0}}$.

On line 18 in Figure 4 we begin synthesizing $\wp_{\mathsf{AG\ x=0}}$ over each location in the program. We begin with the location ℓ_1 (recall that ℓ_0 is omitted and the incoming arrow to ℓ_1 represents the transition $(\ell_0, x' = 0, \ell_1)$). The call to TRANSFORM on line 19 in Figure 4, when applied on ℓ_1, returns the program in Figure 8.

The transition system in Figure 8 is then verified on line 20 using an existing safety prover similar to IMPACT [25] to verify whether or not ERR is reachable. As AG x=0 does not hold from ℓ_2 a CEX_1 is returned below

$$\langle \ell_0, \tau_1, \ell_1 \rangle$$
$$\langle \ell_1, \tau_3, \ell_2 \rangle$$
$$\langle \ell_2, \tau_5, \mathtt{ERR} \rangle$$

We then calculate the weakest precondition of CEX_1 in the different locations reachable on the counterexample from lines 23-29. Starting from ℓ_2 in CEX_1, the calculated result is $wp(\ell_2, \mathsf{CEX}_1) \triangleq \mathsf{x} \neq 0$. The negation of this condition is conjoined with the current value of $\wp_{\ell_2,\mathsf{AG\ x=0}}$ on line 28. Then, from ℓ_1, the calculated result is $wp(\ell_1, \mathsf{CEX}_1) \triangleq \mathsf{true}$ (computed in line 22). The negation of this result is conjoined with the current value of $\wp_{\ell_1,\mathsf{AG\ x=0}}$ on line 28. After our propagation we thus have

$$\wp_{\mathsf{AG\ x=0}} \triangleq (\mathsf{pc} = \ell_1 \Rightarrow \wp_{\ell_1,\mathsf{AG\ x=0}}) \wedge (\mathsf{pc} = \ell_2 \Rightarrow \wp_{\ell_2,\mathsf{AG\ x=0}})$$

We then rule out CEX_1 on lines 30 and 30 by adding $\neg wp(\ell_1, \mathsf{CEX}_1) = \mathsf{false}$ as an assumption on all transitions entering ℓ_1 and re-run the safety prover, continuing onto the next loop refinement iteration. Once CEX_1 is eliminated, we do not generate anymore counterex-

amples and can thus conclude that

$$\wp_{\mathsf{AG}\ \mathsf{x}\,=\,0} \triangleq (\mathsf{pc} = \ell_1 \ \Rightarrow \mathsf{false}) \wedge (\mathsf{pc} = \ell_2 \ \Rightarrow \mathsf{x} = 0)$$
$$\wp_{\mathsf{AG}\ \mathsf{x}\,=\,0} \Rightarrow \mathsf{AG}\ \mathsf{x} = 0$$

Now that we have synthesized $\wp_{\mathsf{AG}\ \mathsf{x}\,=\,0}$, we continue onto verifying the outer-most temporal property $\mathsf{AFAG}\ \mathsf{x} = 0$ by substituting $\mathsf{AG}\ \mathsf{x} = 0$ with $\wp_{\mathsf{AG}\ \mathsf{x}\,=\,0}$, resulting in the temporal property $\mathsf{AF}\ \wp_{\mathsf{AG}\ \mathsf{x}\,=\,0}$, that is, $\mathsf{AF}\ (\mathsf{pc}{=}\ell_1 \Rightarrow \mathsf{false} \wedge \mathsf{pc}{=}\ell_2 \Rightarrow \mathsf{x}{=}0)$, which is equivalent to $\mathsf{AF}\ (\mathsf{pc}{=}\ell_2 \wedge \mathsf{x}{=}0)$. The substitution is performed on line 22 in Figure 7 when called from line 19 in Figure 4, resulting in Figure 9.

Transitions τ_4, τ_5, and τ_6 include the (negation of the) precondition $(\mathsf{pc}{=}\ell_2 \wedge \mathsf{x}{=}0)$, as for AF, we only wish to continue if $\mathsf{pc}{=}\ell_2 \wedge \mathsf{x}{=}0$ has not been satisfied as of yet. Transitions τ_5 and τ_6 correspond to further exploration of the loop (τ_6) and duplication of the current state (τ_5) as required for identification of loops violating the reachability of $\mathsf{x}{=}0$. Then, transition τ_4 reaches ERR if $\mathsf{x} \neq 0$ and there exists no well-founded relation. In the absence of a well-founded relation the identified loop (dup) corresponds to a real loop in the program and a counter-example to reaching $\mathsf{x}{=}0$. Recall that on line 20 in Figure 4 we verify Figure 9 using a refinement procedure utilizing a safety prover alongside termination techniques, allowing us to discover the necessary well-founded relation required for a sub-property to hold.

In the case of our program, when verifying reachability of ERR on the program in Figure 9, no counterexamples are returned as at ℓ_2, we satisfy the requirement that (eventually) $\mathsf{x} = 0$. That is, $\mathsf{AF}\ \wp_{\mathsf{AG}\ \mathsf{x}\,=\,0}$ does hold. This implies

$$\mathsf{true} \Rightarrow \mathsf{AF}\ \wp_{\mathsf{AG}\ \mathsf{x}\,=\,0}$$

We have now shown that $\mathsf{AFAG}\ \mathsf{x} = 0$ holds for the transition system in Figure 1.

Existential path formulae. Recall the program in Figure 2. We are trying to identify the precondition for $\mathsf{EF}\ \mathsf{y} = 1$, denoted $\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1}$. In line 19 in Figure 4 we well TRANSFORM on program location ℓ_1 and the property $\mathsf{AG}\ \mathsf{y}{=}0$. The transformation in line 33 in Figure 7 returns the program in Figure 10. We then verify the resulting program (line 20 in Figure 4) using our safety prover to verify whether or not ERR is reachable. As $\mathsf{AG}\ \mathsf{y}\ {=}0$ does not hold at ℓ_2 the counter example CEX_1 below is returned.

$$\langle \ell_0, \tau_1, \ell_1 \rangle$$
$$\langle \ell_1, \tau_2, \ell_1 \rangle$$
$$\langle \ell_1, \tau_4, \ell_2 \rangle$$
$$\langle \ell_2, \tau_5, \ell_2 \rangle$$
$$\langle \ell_2, \tau_6, \mathtt{ERR} \rangle$$

We then begin our refinement loop and calculate the weakest precondition of CEX_1 on line 22 for ℓ_1. The calculated result is $wp(\ell_1, \mathsf{CEX}_1) \triangleq \mathsf{x} \geq 0$ and is disjuncted to $\wp_{\ell_1, \mathsf{EF}\ \mathsf{y}\,=\,1}$ on line 25. Recall that unlike the universal example, we do not negate $wp(\ell_1, \mathsf{CEX}_1)$ as our counterexample derived from verifying its universal dual indicates a subset of the existential witness. On line 23, we then begin to propagate our synthesized precondition using CEX_1 starting from ℓ_2, as ERR is reachable from ℓ_2. After our propagation we thus have

$$\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1} \triangleq (\mathsf{pc} = \ell_1 \ \Rightarrow \wp_{\ell_1, \mathsf{EF}\ \mathsf{y}\,=\,1}) \wedge (\mathsf{pc} = \ell_2 \ \Rightarrow \wp_{\ell_2, \mathsf{EF}\ \mathsf{y}\,=\,1})$$

One existential witness may not be sufficient to find all states that satisfy $\mathsf{EF}\ \mathsf{y} = 1$ in the respective locations, we thus rule out CEX_1 on lines 30 and 31 by adding $\neg wp(\ell_1, \mathsf{CEX}_1) = \mathsf{x} \leq 0$ as an assumption and begin the next loop iteration. We re-run the safety prover and do not generate anymore counterexamples and can thus conclude that

$$\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1} \triangleq (\mathsf{pc} = \ell_1 \ \Rightarrow x \geq 0) \wedge (\mathsf{pc} = \ell_2 \ \Rightarrow x \geq 0)$$
$$\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1} \Rightarrow \mathsf{EF}\ \mathsf{y} = 1$$

Now that we have synthesized $\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1}$, we continue onto verifying the outer-most temporal property $\mathsf{AGEF}\ \mathsf{y} = 1$ by substituting $\mathsf{EF}\ \mathsf{y} = 1$ with $\wp_{\mathsf{EF}\ \mathsf{y}\,=\,1}$, resulting in the temporal property $\mathsf{AG}\ \mathsf{x} \geq 0$. The substitution is carried out on line 28 in Figure 7. By calling TRANSFORM on line 19 in Figure 4 we obtain the program in Figure 11. Transitions τ_2 and τ_5 reflect $(\mathsf{pc} = \ell_1 \ \Rightarrow \mathsf{x} \geq 0) \wedge (\mathsf{pc} = \ell_2 \ \Rightarrow \mathsf{x} \geq 0)$. Transitions τ_3 and τ_6 correspond to $\neg((\mathsf{pc} = \ell_1 \ \Rightarrow \mathsf{x} \geq 0) \wedge (\mathsf{pc} = \ell_2 \ \Rightarrow \mathsf{x} \geq 0))$ and reach ERR. The safety prover returns no counterexamples, as $\mathsf{AG}\ \wp_{\mathsf{EF}\ \mathsf{y}\,=\,1}$ does hold. This implies

$$\mathsf{true} \Rightarrow \mathsf{AG}\ \wp_{\mathsf{EF}\ \mathsf{y}\,=\,1}$$

We have now shown that $\mathsf{AGEF}\ \mathsf{y}{=}1$ holds for the transition system in Figure 2.

6. Evaluation

In this section we discuss the results of our experiments with an implementation of the procedure from Figure 3. Our implementation is built as an extension to the open sourcecode of T2 [1], which uses a safety prover similar to IMPACT [25] alongside previously published techniques for discovering ranking functions, etc [18, 26].

We have compared our tool to that of Cook & Koskinen [11] and Beyene *et al.* [4]. The benchmarks used are the same as those used in [11] 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 [20]. For each program and CTL property φ, we verify both φ and $\neg\varphi$. Our tool was executed on hardware identical to the hardware used by Cook & Koskinen and Beyene *et al.*: an Intel x64-based 2.8 GHz single-core processor.

In Figure 12 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 $\times$ indicates that the tool failed to prove the property. A timeout or memory exception is indicated by T/O. When the symbol "–" appears in the Time and Result column this indicates that the experiment was not run.

Overall, our tool demonstrates a significant increase in performance. 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, our tool demonstrates orders-of-magnitude performance improvement over existing tools.

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.*, and Cook & Koskinen acknowledge the bug in their tool. OS frag. 2 needs support for fairness, which our implementation does not have.

Program	LOC	Property	Our procedure (Fig. 3) Time	Our procedure (Fig. 3) Result	Beyene *et al.* [4] Time	Beyene *et al.* [4] Result	Cook & Koskinen [11] Time	Cook & Koskinen [11] Result
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	1.4	✓	1.20	✓	4.6	✓
OS frag. 1	29	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	0.1	✓	4.8	✓	9.5	✓
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	0.2	✓	0.6	✓	105.7	✓
OS frag. 1	29	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	1.0	✓	0.6	✓	3.5	✓
OS frag. 1	29	$\neg(\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1)))$	1.4	×	2.7	×	9.1	×
OS frag. 1	29	$\neg(\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1)))$	0.1	×	0.1	×	1.5	×
OS frag. 1	29	$\neg(\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1)))$	0.1	×	0.4	×	18.1	×
OS frag. 1	29	$\neg(\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1)))$	0.7	×	5.2	×	12.5	×
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{AF}(u = 1))$	2.1	×	6.1	✓	2.1	✓
OS frag. 2	58	$\mathsf{AG}(s = 1 \Rightarrow \mathsf{EF}(u = 1))$	0.2	✓	12.9	✓	3.7	✓
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{AG}(u \neq 1))$	2.1	✓	44.7	✓	5.6	✓
OS frag. 2	58	$\mathsf{EF}(s = 1 \wedge \mathsf{EG}(u \neq 1))$	0.2	✓	1.4	✓	1.2	✓
OS frag. 2	58	$\neg(\mathsf{AG}(s = 1 \Rightarrow \mathsf{AF}(u = 1)))$	0.2	✓	0.2	×	1.8	×
OS frag. 2	58	$\neg(\mathsf{AG}(s = 1 \Rightarrow \mathsf{EF}(u = 1)))$	0.2	×	0.2	×	1.5	×
OS frag. 2	58	$\neg(\mathsf{EF}(s = 1 \wedge \mathsf{AG}(u \neq 1)))$	0.2	×	3.8	×	8.7	×
OS frag. 2	58	$\neg(\mathsf{EF}(s = 1 \wedge \mathsf{EG}(u \neq 1)))$	1.2	×	3.6	×	6.5	×
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1))$	6.9	✓	51.3	✓	38.9	✓
OS frag. 3	370	$\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1))$	2.8	✓	67.6	✓	90.0	✓
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1))$	2.9	✓	67.9	✓	T/O	–
OS frag. 3	370	$\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1))$	6.2	✓	132.0	✓	1680.7	✓
OS frag. 3	370	$\neg(\mathsf{AG}(a = 1 \Rightarrow \mathsf{AF}(r = 1)))$	6.2	×	120.0	×	18.0	×
OS frag. 3	370	$\neg(\mathsf{AG}(a = 1 \Rightarrow \mathsf{EF}(r = 1)))$	3.4	×	3.9	×	107.3	×
OS frag. 3	370	$\neg(\mathsf{EF}(a = 1 \wedge \mathsf{AG}(r \neq 1)))$	3.1	×	3.8	×	T/O	–
OS frag. 3	370	$\neg(\mathsf{EF}(a = 1 \wedge \mathsf{EG}(r \neq 1)))$	6.0	×	45.9	×	1930.0	×
OS frag. 4	370	$\mathsf{AF}(io = 1) \vee \mathsf{AF}(ret = 1)$	12.9	✓	2284	✓	34.3	✓
OS frag. 4	370	$\mathsf{EG}(io \neq 1) \wedge \mathsf{EG}(ret \neq 1)$	10.2	✓	T/O	–	7.6	✓
OS frag. 4	370	$\mathsf{EF}(io = 1) \wedge \mathsf{EF}(ret = 1)$	8.6	✓	T/O	–	1261.0	✓
OS frag. 4	370	$\mathsf{AG}(io \neq 1) \vee \mathsf{AG}(ret \neq 1)$	3.0	✓	0.1	✓	–	–
OS frag. 4	370	$\neg(\mathsf{AF}(io = 1) \vee \mathsf{AF}(ret = 1))$	13.9	×	T/O	–	18.8	×
OS frag. 4	370	$\neg(\mathsf{EG}(io \neq 1) \wedge \mathsf{EG}(ret \neq 1))$	14.2	×	136.6	×	61.3	×
OS frag. 4	370	$\neg(\mathsf{EF}(io = 1) \wedge \mathsf{EF}(ret = 1))$	4.8	×	1.4	×	T/O	×
OS frag. 4	370	$\neg(\mathsf{AG}(io \neq 1) \vee \mathsf{AG}(ret \neq 1))$	3.7	×	874.5	×	–	–
OS frag. 6	1050	$\mathsf{AG}(b = 1 \Rightarrow \mathsf{AF}(u = 0))$	67.3	✓	T/O	–	T/O	–
OS frag. 6	1050	$\mathsf{EG}(b = 1 \Rightarrow \mathsf{EF}(u = 0))$	36.2	✓	T/O	–	T/O	–
OS frag. 6	1050	$\neg(\mathsf{AG}(b = 1 \Rightarrow \mathsf{AF}(u = 0)))$	82.9	×	T/O	–	T/O	–
OS frag. 6	1050	$\neg(\mathsf{EG}(b = 1 \Rightarrow \mathsf{EF}(u = 0)))$	38.8	×	T/O	–	–	–
OS frag. 5	58	$\mathsf{AG}(\mathsf{AF}(w \geq 1))$	0.2	✓	3.0	✓	569.7	✓
OS frag. 5	58	$\mathsf{AG}(\mathsf{EF}(w \geq 1))$	0.1	✓	3.3	✓	T/O	–
OS frag. 5	58	$\mathsf{EF}(\mathsf{AG}(w < 1))$	0.1	✓	0.7	✓	255.8	✓
OS frag. 5	58	$\mathsf{EF}(\mathsf{EG}(w < 1))$	0.1	✓	0.5	✓	351.1	✓
OS frag. 5	58	$\neg(\mathsf{AG}(\mathsf{AF}(w \geq 1)))$	0.2	×	0.1	×	65.1	×
OS frag. 5	58	$\neg(\mathsf{AG}(\mathsf{EF}(w \geq 1)))$	0.0	×	0.1	×	T/O	–
OS frag. 5	58	$\neg(\mathsf{EF}(\mathsf{AG}(w < 1)))$	0.1	×	0.1	×	85.5	×
OS frag. 5	58	$\neg(\mathsf{EF}(\mathsf{EG}(w < 1)))$	0.0	×	0.1	×	1471.7	×
PgSQL arch	90	$\mathsf{AG}(\mathsf{AF}(w = 1))$	1.9	✓	2.8	✓	T/O	–
PgSQL arch	90	$\mathsf{AG}(\mathsf{EF}(w = 1))$	0.0	✓	4.5	✓	T/O	–
PgSQL arch	90	$\mathsf{EF}(\mathsf{AG}(w \neq 1))$	2.1	×	3.4	✓	T/O	–
PgSQL arch	90	$\mathsf{EF}(\mathsf{EG}(w \neq 1))$	0.1	✓	2.2	✓	35.2	✓
PgSQL arch	90	$\neg(\mathsf{AG}(\mathsf{AF}(w = 1)))$	1.3	×	0.1	×	38.1	×
PgSQL arch	90	$\neg(\mathsf{AG}(\mathsf{EF}(w = 1)))$	0.0	×	0.1	×	42.7	×
PgSQL arch	90	$\neg(\mathsf{EF}(\mathsf{AG}(w \neq 1)))$	2.4	✓	0.7	×	30.2	×
PgSQL arch	90	$\neg(\mathsf{EF}(\mathsf{EG}(w \neq 1)))$	0.1	×	5.0	×	45.3	×
S/W Updates	36	$c > 5 \Rightarrow \mathsf{AF}(r > 5)$	0.1	×	3.2	✓	70.2	✓
S/W Updates	36	$c > 5 \Rightarrow \mathsf{EF}(r > 5)$	0.1	✓	0.2	×	18.5	✓
S/W Updates	36	$c > 5 \wedge \mathsf{AG}(r \leq 5)$	0.1	×	0.1	×	0.3	✓
S/W Updates	36	$c > 5 \wedge \mathsf{EG}(r \leq 5)$	1.0	×	0.1	×	4.5	✓
S/W Updates	36	$\neg(c > 5 \Rightarrow \mathsf{AF}(r > 5))$	1.1	✓	0.1	×	32.4	×
S/W Updates	36	$\neg(c > 5 \Rightarrow \mathsf{EF}(r > 5))$	0.8	×	0.1	×	1.3	×
S/W Updates	36	$\neg(c > 5 \wedge \mathsf{AG}(r \leq 5))$	1.1	✓	0.3	×	0.5	×
S/W Updates	36	$\neg(c > 5 \wedge \mathsf{EG}(r \leq 5))$	0.7	✓	1.3	×	0.4	×

Figure 12: The results of applying our CTL model checking procedure on benchmarks from [4, 11]. For each program we verify a set of properties and their negations and compare our results with [4, 11]. The results from [4, 11] were not re-run, however experiments with our approach were run on an identical hardware configuration.

7. Concluding remarks

In this paper we have described an optimization to the standard recursively defined procedure for CTL that takes advantage of the structure of control-flow graphs available in programs. 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.

[2] M. Barnett and K. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 82–87. ACM, 2005.

[3] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking (extended abstract). In D. L. Dill, editor, *CAV'94*, volume 818, pages 142–155. Springer, 1994. ISBN 3-540-58179-0.

[4] T. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV*, volume 8044 of *LNCS*, pages 869–882, 2013.

[5] J. Burch, E. Clarke, et al. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[6] S. Chaki, E. M. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/event software verification for branching-time specifications. In J. Romijn, G. Smith, and J. van de Pol, editors, *IFM'05*, volume 3771, pages 53–69, 2005. ISBN 3-540-30492-4.

[7] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An open-source tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404, pages 359–364. Springer, 2002. ISBN 3-540-43997-8.

[8] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[9] E. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *LICS*, pages 19–29, 2002.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8:244–263, April 1986. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/5397.5399`.

[11] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–230. ACM, 2013.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *PLDI'06*, pages 415–426, 2006. ISBN 1-59593-320-4.

[13] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL'07*, pages 265–276, 2007.

[14] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[15] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV'96*, volume 1102, pages 87–98, 1996. ISBN 3-540-61474-5.

[16] J. Esparza, A. Kucera, and S. Schwoon. Model checking ltl with regular valuations for pushdown systems. *Information and Computation*, 186:355–376, November 2003. ISSN 0890-5401. . URL `http://dl.acm.org/citation.cfm?id=957691.957699`.

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

[18] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. *SIGPLAN Not.*, 43:147–158, January 2008. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/1328897.1328459`.

[19] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In *CAV'06*, volume 4144, pages 170–174, 2006. ISBN 3-540-37406-X.

[20] 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*, volume 7152, pages 278–293, 2012.

[21] Y. Kesten and A. Pnueli. A compositional approach to ctl* verification. *Theor. Comput. Sci.*, 331(2-3):397–428, 2005.

[22] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2): 312–360, 2000.

[23] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In H. R. Nielson and G. Filé, editors, *Proceedings of the 14th International Static Analysis Symposium (SAS 2007)*, volume 4634, pages 419–436. Springer, 2007. ISBN 978-3-540-74060-5.

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

[25] K. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[26] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[27] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[28] F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, 2012.

[29] I. Walukiewicz. Pushdown processes: Games and model checking. In *CAV*, volume 1102, pages 62–74, 1996.

[30] I. Walukiewicz. Model checking ctl properties of pushdown systems. In S. Kapoor and S. Prasad, editors, *FSTTCS*, volume 1974, pages 127–138, 2000.