Efficient CTL Verification via Horn Constraints Solving

Tewodros A. Beyene
fortiss GmbH
Munich, Germany
beyene@fortiss.org

Corneliu Popeea
CQSE GmbH
Munich, Germany
popeea@cqse.eu

Andrey Rybalchenko
Microsoft Research
Cambridge, UK
rybal@microsoft.com

Temporal logics has long been recognised as a fundamental approach to the formal specification and verification of reactive systems. In this paper, we take on the problem of automatically verifying temporal property, given by a CTL formula, for a given (possibly infinite-state) program. We propose a method based on encoding the problem as a set of Horn constraints. The method takes a program, modeled as a transition system, and a property given by a CTL formula as input. It first generates a set of forall-exists quantified Horn constraints and well-foundedness constraints by exploiting the syntactic structure of the CTL formula. Then, the generated set of constraints are solved by applying an off-the-shelf Horn constraints solving engine. The program is said to satisfy the property if and only if the generated set of constraints has a solution. We demonstrate the practical promises of the method by applying it on a set of challenging examples. Although our method is based on a generic Horn constraint solving engine, it is able to outperform state-of-art methods specialised for CTL verification.

1 Introduction

Since Pnueli's pioneering work [25], temporal logics has long been recognised as a fundamental approach to the formal specification and verification of reactive systems [13, 22]. Temporal logics allow precise specification of complex properties. There has been decades of effort on temporal verification of finite state systems [3, 5, 6, 21]. For CTL and other state-based properties, the standard procedure is to adapt bottom-up (or tableaux) techniques for reasoning on finite-state systems. In additional, various classes of temporal logics support model-checking whose success over the last twenty years is allowing large and complex (finite) systems to be verified automatically [3, 7, 18, 23]. In recent decades, however, the research focus has shifted to infinite-state systems in general and on software systems in particular as ensuring correctness for software is in high demand. Most algorithms for verifying CTL properties on infinite-state systems typically involve first abstracting the state space into a finite-state model, and then applying finite reasoning strategies on the abstract model. There is also a lot of effort on algorithms that are focused on a particular fragment of CTL, such as the universal fragment [24] and the existential fragment [17], or some particular classes of infinite-state systems such as pushdown processes [26–29] or parameterised systems [12, 14].

In this paper, we take on the problem of automatically verifying CTL properties for a given (possibly infinite-state) program. We propose a method based on solving a set of forall-exists quantified Horn constraints. Our method takes a program P modeled by a transition system $(init(v), next(v,v'))$ and a property given by a CTL formula $\varphi(v)$, and then it checks if P satisfies $\varphi(v)$, i.e., if $(init(v), next(v,v')) \models_{CTL} \varphi(v)$. The method first generates a set of forall-exists quantified Horn constraints with well-foundedness conditions by exploiting the syntactic structure of the CTL formula $\varphi(v)$. It then solves the generated set of Horn constraints by applying an off-the-shelf solving engine E-HSF [1] for such constraints. We claim that P satisfies $\varphi(v)$ if and only if the generated set of Horn constraints has a solution. We demonstrate

Submitted to:
HCVS 2016

the practical applicability of the method by presenting experimental evaluation using examples from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

The rest of the paper is organised as follows. We start by revising the syntax and semantics of CTL and by a brief introduction of forall-exists quantified Horn constraints and their solver E-HSF in Section 2. In Section 3, we present our CTL proof system that generates a set of forall-exists quantified Horn constraints for a given verification problem. We illustrate application of the proof rules on an example in Section 4. The experimental evaluation of our method is given in Section 5. Finally, we present a bief discussion on related works in Section 6 and concluding remarks in Section 7.

2 Preliminiaries

2.1 CTL basics

In this section, we review the syntax and the semantics of the logic CTL following [20]. Let $\mathscr{T}$ be a first order theory and $\models_{\mathscr{T}}$ denote its satisfaction relation that we use to describe sets and relations over program states. Let c range over assertions in $\mathscr{T}$ and x range over variables. A CTL formula φ is defined by the following grammar using the notion of a path formula ϕ.

$$\varphi ::= c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\,\phi \mid E\,\phi$$
$$\phi ::= X\varphi \mid G\varphi \mid \varphi U \varphi$$

As usual, we define $F\varphi = (trueU\varphi)$. The satisfaction relation $P \models \varphi$ holds if and only if for each s such that $init(s)$ we have $P,s \models \varphi$. We define $P,s \models \varphi$ as follows using an auxiliary satisfaction relation $P,\pi \models \phi$.

$$\begin{array}{ll}
P,s \models c & \text{iff } s \models_{\mathscr{T}} c \\
P,s \models \varphi_1 \wedge \varphi_2 & \text{iff } P,s \models \varphi_1 \text{ and } P,s \models \varphi_2 \\
P,s \models \varphi_1 \vee \varphi_2 & \text{iff } P,s \models \varphi_1 \text{ or } P,s \models \varphi_2 \\
P,s \models A\,\phi & \text{iff for all } \pi \in \Pi_P(s) \text{ holds } P,\pi \models \phi \\
P,s \models E\,\phi & \text{iff exists } \pi \in \Pi_P(s) \text{ such that } P,\pi \models \phi \\
P,\pi \models X\varphi & \text{iff } \pi = s_1,s_2,\ldots \text{ and } P,s_2 \models \varphi \\
P,\pi \models G\varphi & \text{iff } \pi = s_1,s_2,\ldots \text{for all } i \geq 1 \text{ holds } P,s_i \models \varphi \\
P,\pi \models \varphi_1 U \varphi_2 & \text{iff } \pi = s_1,s_2,\ldots \text{ and exists } j \geq 1 \text{ such that} \\
 & \qquad P,s_j \models \varphi_2 \text{ and } P,s_i \models \varphi_1 \text{ for } 1 \leq i \leq j
\end{array}$$

In this paper, we represent a satisfaction relation $P \models \varphi$ by the relation $P \models_{CTL} \varphi$ to explicitly indicate that φ is a CTL formula. We call such relation a CTL satisfaction, and φ is said to be its formula.

2.2 The solving algorithm E-HSF

Our proof rules are automated using the E-HSF engine for resolving forall-exists Horn-like clauses extended with well-foundedness criteria.

We skip the syntax and semantics of the clauses targeted by this system — see [1] for more details. Instead, we illustrate these clauses with the following example:

$$\begin{array}{ll}
x \geq 0 \rightarrow \exists y : x \geq y \wedge rank(x,y), & rank(x,y) \rightarrow ti(x,y), \\
ti(x,y) \wedge rank(y,z) \rightarrow ti(x,z), & dwf(ti).
\end{array}$$

Intuitively, these clauses represent an assertion over the interpretation of "query symbols" *rank* and *ti* (the predicate *dwf* represents disjunctive well-foundedness, and is not a query symbol). The semantics of these clauses maps each predicate symbol occurring in them into a constraint over v.

Specifically, the above set of clauses has a solution that maps both $rank(x,y)$ and $ti(x,y)$ to the constraint $(x \geq 0 \wedge y \geq x-1)$.

E-HSF resolves clauses like the above using a CEGAR scheme to discover witnesses for existentially quantified variables. The refinement loop collects a global constraint that declaratively determines which witnesses can be chosen. The chosen witnesses are used to replace existential quantification, and then the resulting universally quantified clauses are passed to a solver for such clauses. At this step, we can benefit from emergent tools in the area of solving Horn clauses over decidable theories, e.g., HSF [15] or μZ [19]. Such a solver either finds a solution, i.e., a model for uninterpreted relations constrained by the clauses, or returns a counterexample, which is a resolution tree (or DAG) representing a contradiction. E-HSF turns the counterexample into an additional constraint on the set of witness candidates, and continues with the next iteration of the refinement loop. Notably, this refinement loop conjoins constraints that are obtained for all discovered counterexamples. This way E-HSF guarantees that previously handled counterexamples are not rediscovered and that a wrong choice of witnesses can be mended.

For the existential clause above, E-HSF introduces a witness/Skolem relation *rel* over variables x and y, i.e., $x \geq 0 \wedge rel(x,y) \rightarrow x \geq y \wedge rank(x,y)$. In addition, since for each x such that $x \geq 0$ holds we need a value y, we require that such x is in the domain of the Skolem relation using an additional clause $x \geq 0 \rightarrow \exists y : rel(x,y)$. In the E-HSF approach, the search space of a skolem relation $rel(x,y)$ is restricted by a template function $\text{TEMPL}(rel)(x,y)$. In general, E-HSF requires such template functions to be given by the user.

3 Proof system

Our CTL verification method encodes the verification problem as a problem of solving forall-exists quantified Horn constraints with well-foundedness conditions. This is done by applying a proof system that consists of various proof rules for handling different kinds of CTL formulas. This proof system is based on a deductive proof system for CTL* from [20] which is adapted in this work to be suitable from the perspective of constraint generation for a CTL satisfaction.

Given a transition system $(init(v), next(v,v'))$ and a CTL formula $\varphi(v)$, the appropriate proof rules are used from the proof system to generate the corresponding set of Horn constraints for the CTL satisfaction $(init(v), next(v,v')) \models_{CTL} \varphi(v)$. There are two sets of proof rules in the proof system.

3.1 Proof rules for decomposition

These proof rules are applied recursively to a CTL satisfaction whose formula is neither an assertion nor a basic CTL state formula. The proof rules decompose the given CTL formula into new sub-formulas by following the nesting structure of the formula. Then, the original satisfaction is reduced to new satisfactions over the new sub-formulas and a Horn constraint relating the new satisfactions.

There are different proof rules depending on the outer-most operator of the formula. One case is when the given formula $f(\psi(v))$ nests another formula $\psi(v)$ such that the outer-most operator f is a pair of a temporal path operator and a unary temporal state operator, i.e., $f \in \{AX, AG, AF, EX, EG, EF\}$. The corresponding proof rule RULECTLDECOMPUNI is given in Figure 1 that shows how such satisfactions

are decomposed. Another case is when the given formula has a structure $f(\psi_1(v),\ \psi_2(v))$ nesting the

Given a CTL formula $f(\psi(v))$ where $f \in \{AX, AG, AF, EX, EG, EF\}$, and a transition system $(p(v), next(v,v'))$, find an assertion $q(v)$ such that:

$$\frac{(p(v), next(v,v')) \models_{CTL} f(q(v)) \quad (q(v), next(v,v')) \models_{CTL} \psi(v)}{(p(v), next(v,v')) \models_{CTL} f(\psi(v))}$$

Figure 1: Proof rule RULECTLDECOMPUNI

formulas $\psi_1(v)$ and $\psi_2(v)$ such that the outer-most operator f is either a pair of a temporal path operator and the state operator until or a disjunction/conjunction, i.e., $f \in \{AU, EU, \wedge, \vee\}$. Note that when f is $\wedge$ (resp. $\vee$), the given formula $f(\psi_1(v),\ \psi_2(v))$ corresponds to $\psi_1(v) \wedge \psi_2(v)$ (resp $\psi_1(v) \vee \psi_2(v)$). The corresponding proof rule RULECTLDECOMPBIN is given in Figure 2 that shows how such satisfactions are decomposed.

Given a CTL formula $f(\psi_1(v),\ \psi_2(v))$ where $f \in \{AU, EU, \wedge, \vee\}$, and a transition system $(p(v), next(v,v'))$, find assertions $q_1(v)$ and $q_2(v)$ such that:

$$\frac{\begin{array}{c} p(v) \rightarrow f(q_1(v), q_2(v)), \\ (q_1(v), next(v,v')) \models_{CTL} \psi_1(v) \quad (q_2(v), next(v,v')) \models_{CTL} \psi_2(v) \end{array}}{(p(v), next(v,v')) \models_{CTL} f(\psi_1(v), \psi_2(v))}$$

Figure 2: Proof rule RULECTLDECOMPBIN

3.2 Proof rules for constraints generation

This set of proof rules are applied to a CTL satisfaction whose formula is either an assertion or a basic state formula. Any CTL satisfaction can be decomposed into a set of such simple CTL satisfactions by applying the proof rules from the previous section. The next step will be to generate forall-exists quantified Horn constraints (possibly with well-foundedness condition) that constrain a set of auxiliary assertions over program states.

The simplest of all is the proof rule RULECTLINIT, see Figure 3, which is applied when the CTL formula is an assertion.

The proof rules RULECTLEX (see Figure 4), RULECTLEG (see Figure 5), and RULECTLEU (see Figure 6) are applied for generating Horn constraints when the CTL formula is a basic state formula with existential path operator.

The corresponding proof rules for generating Horn constraints when the CTL formula is a basic state formula with universal path operator are given in the appendix section.

For a CTL formula given by the assertion $\psi(v)$, and a transition system $(p(v), next(v,v'))$:

$$\frac{p(v) \rightarrow \psi(v)}{(p(v), next(v,v')) \models_{CTL} \psi(v)}$$

Figure 3: Proof rule RULECTLINIT

$$\frac{p(v) \rightarrow \exists v' : next(v,v') \wedge q(v')}{(p(v), next(v,v')) \models_{CTL} EX\ q(v)}$$

Figure 4: Proof rule RULECTLEX

4 Constraint generation

The contraint generation procedure performs a top-down, recursive descent through the syntax tree of the given CTL formula. At each level of recursion, the procedure takes as input a CTL satisfaction $(p(v), next(v,v')) \models_{CTL} \varphi$, where φ is a CTL formula, and assertions $p(v)$ and $next(v,v')$ describe a set of states and a transition relation, respectively. The constraint generation procedure applies proof rules from the proof system presented in the previous section to recursively decompose complex satisfactions and eventually generate forall-exists quantified Horn constraints with well-foundedness conditions. Before starting the actual constraint generation, the procedure recursively re-writes the input satisfaction of a given CTL formula with arbitrary structure into a set of satisfactions of simple CTL formulas where each simple formula is either a basic CTL state formula or an assertion over the background theory. The procedure then takes each satisfaction involving simple formula, introduces auxiliary predicates and generates a sequence of forall-exists quantified Horn constraints and well-foundedness constraints (when needed) over these predicates.

Complexity and Correctness The procedure performs a single top-down descent through the syntax tree of the given CTL formula φ. The run time and the size of the generated constraints is linear in the size of φ. Finding a solution for the generated Horn constraints is undecidable in general. In practice however, our solving algorithm E-HSF often succeeds in finding a solution (see Section **??**). We formalize the correctness of the constraint generation procedure in the following theorem.

Theorem 1. *For a given program P with $init(v)$ and $next(v,v')$ over v and a CTL formula φ the Horn constraints generated from $(p(v), next(v,v')) \models_{CTL} \varphi$ are satisfiable if and only if $P \models \varphi$.*

The proof can be found in [20].

Example Let us consider the program given in Figure 7. It contains the variable *rho* which is assigned a non-deterministic value at Line 4. This assignment results in the program control to move

Find an assertion $inv(v)$ such that:

$$\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \rightarrow \exists v' : next(v,v') \wedge inv(v') \\ inv(v) \rightarrow q(v) \end{array}$$

$$\hline$$

$$(p(v), next(v,v')) \models_{CTL} EG\ q(v)$$

Figure 5: Proof rule RULECTLEG

Find assertions $inv(v)$, $rank(v,v')$ and $ti(v,v')$ such that:

$$\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \wedge \neg r(v) \rightarrow q(v) \wedge \exists v' : next(v,v') \wedge inv(v') \wedge rank(v,v') \\ rank(v,v') \rightarrow ti(v,v') \\ ti(v,v') \wedge rank(v',v'') \rightarrow ti(v,v'') \\ dwf(ti) \end{array}$$

$$\hline$$

$$(p(v), next(v,v')) \models_{CTL} EU(q(v), r(v))$$

Figure 6: Proof rule RULECTLEU

non-deterministically following the evaluation of the condition at Line 6. It is common to verify such programs with respect to various CTL properties as the non-determinism results in different computation paths of the program. Now, we would like to verify the example program with respect to the CTL property $AG(EF\ (WItemsNum \geq 1))$, i.e., from every reachable state of the program, there exists a path to a state where *WItemsNum* has a positive integer value.

We can make the following observations about the program. The value of the variable *WItemsNum* is not set initially. Therefore, the property is checked for any arbitrary initial value of *WItemsNum*. The verification problem is more interesting for the case when *WItemsNum* has a non-positive integer value. This is because depending on how the variable *rho* is instantiated at Line 4, we may get a path that will not reach a state where *WItemsNum* gets a positive integer value. For example, if we assume *WItemsNum* has the value 0 initially and *WItemsNum* is instantiated to the value 1, the program control swings between the two internal loops by keeping the value of *WItemsNum* the same. This resulting path will not reach the state with $WItemsNum \geq 1$. However, if *rho* is assigned a non-positive value, no matter what the value of *rho* is initially, it will eventually reach a value greater than 5 before exiting the first nested loop. Such path will eventually reach the state with $WItemsNum \geq 1$ and hence the program satisfies the CTL property $AG(EF\ (WItemsNum \geq 1))$.

Our method abstracts away from the concrete syntax of a programming language by modeling a

```
       int main () {
1:       while(1) {
2:         while(1) {
             rho = nondet();
3:           if (WItemsNum<=5) {
4:             if (rho>0) break; }
5:           WItemsNum++;
6:         }
7:         while(1) {
8:           if (!(WItemsNum>2)) break;
9:           WItemsNum--;
10:        }
11:      }
12:    }
```

Figure 7: An example program

program as a transition system. The transition system for our example program is given below.

$$
\begin{aligned}
v &= (w, pc).\\
\mathit{init}(v) &= (pc = 1).\\
\mathit{next}(v, v') &= (pc = \ell_1 \wedge pc'_1 = \ell_2 \wedge w' = w \vee\\
&\quad pc = \ell_2 \wedge pc'_1 = \ell_3 \wedge w' = w \vee\\
&\quad pc = \ell_3 \wedge w \leq 5 \wedge pc'_1 = \ell_4 \wedge w' = w \vee\\
&\quad pc = \ell_3 \wedge w > 5 \wedge pc'_1 = \ell_5 \wedge w' = w \vee\\
&\quad pc = \ell_4 \wedge pc'_1 = \ell_5 \wedge w' = w \vee\\
&\quad pc = \ell_4 \wedge pc'_1 = \ell_7 \wedge w' = w \vee\\
&\quad pc = \ell_5 \wedge pc'_1 = \ell_6 \wedge w' = w + 1 \vee\\
&\quad pc = \ell_6 \wedge pc'_1 = \ell_3 \wedge w' = w \vee\\
&\quad pc = \ell_7 \wedge pc'_1 = \ell_8 \wedge w' = w \vee\\
&\quad pc = \ell_8 \wedge w \leq 2 \wedge pc'_1 = \ell_{11} \wedge w' = w \vee\\
&\quad pc = \ell_8 \wedge w > 2 \wedge pc'_1 = \ell_9 \wedge w' = w \vee\\
&\quad pc = \ell_9 \wedge pc'_1 = \ell_{10} \wedge w' = w - 1 \vee\\
&\quad pc = \ell_{10} \wedge pc'_1 = \ell_8 \wedge w' = w).
\end{aligned}
$$

In the tuple of program variables v, w corresponds to the program variable *WItemsNum* and pc is the program counter variable. The problem of verifying the program with respect to the given property amounts to checking if $(\mathit{init}(v), \mathit{next}(v, v'))$ satisfies $AG(EF(w \geq 1))$, i.e., if the satisfaction $(\mathit{init}(v), \mathit{next}(v, v')) \models_{CTL} AG(EF(w \geq 1))$ holds. Our method first generates a set of Horn constraint corresponding to the verification problem by applying the proof system.

We start constraint generation by considering the nesting structure of $AG(EF(w \geq 1))$. Since $AG(EF(w \geq 1))$ has AG as the outer-most operator, we apply RULECTLDECOMPUNI from Figure 1 to split the original satisfaction $(\mathit{init}(v), \mathit{next}(v, v')) \models_{CTL} AG(EF(w \geq 1))$ into a reduced satisfaction $(\mathit{init}(v), \mathit{next}(v, v')) \models_{CTL}$

$AG(p_1(v))$ and a new satisfaction $(p_1(v), next(v,v')) \models_{CTL} EF(w \geq 1)$. We need to solve for the auxiliary assertion $p_1(v)$ satisfying both of the satisfactions.

One one hand, the assertion $p_1(v)$ corresponds to a set of program states that needs to be discovered form the initial state. This is represented by the new satisfaction $(init(v), next(v,v')) \models_{CTL} AG(p_1(v))$ which is reduced directly to a set of Horn constraints by applying RULECTLAG from Figure 9. This set of Horn constraints is over an auxiliary predicate $inv_1(v)$ and given below.

$$
\begin{array}{l}
init(v) \rightarrow inv_1(v), \\
inv_1(v) \wedge next(v,v') \rightarrow inv_1(v'), \\
inv_1(v) \rightarrow p_1(v).
\end{array}
$$

On the other hand, we require the formula $EF(w \geq 1)$, which was nested in the main formula $AG(EF(w \geq 1))$, must be satisfied from the set of states represented by $p_1(v)$. This is represented by the new satisfaction $(p_1(v), next(v,v')) \models_{CTL} EF(w \geq 1)$. Unlike the reduced satisfaction above, this satisfaction is not always reduced directly to Horn constraints rather it can be reduced into simpler satisfactions if possible. Since $EF(w \geq 1)$ has EF as the outer-most operator, we apply again RULECTLDECOMPUNI from Figure 1 to split the satisfaction $(\mathtt{p}_1(v), next(v,v')) \models_{CTL} EF(w \geq 1)$ into a reduced satisfaction $(\mathtt{p}_1(v), next(v,v')) \models_{CTL} EF(p_2(v))$ and a new satisfaction $(p_2(v), next(v,v')) \models_{CTL} w \geq 1$. Here also, we need to solve for the auxiliary assertion $p_2(v)$ satisfying both of the satisfactions.

The reduced satisfaction $(\mathtt{p}_1(v), next(v,v')) \models_{CTL} EF(p_2(v))$ is reduced directly to a set of Horn constraints by applying RULECTLEF from Figure 11. Due to the existential path quantifier in $(p_1(v), next(v,v')) \models_{CTL} EF(p_2(v))$, we obtain clauses that contain existential quantification. We deal with the eventuality by imposing a well-foundedness condition. This set of Horn constraints is over an auxiliary assertions $inv_2(v)$, $rank(v,v')$, and $ti(v,v')$ and given below.

$$
\begin{array}{l}
p_1(v) \rightarrow inv_2(v), \\
inv_2(v) \wedge \neg p_2(v) \rightarrow \exists v' : next(v,v') \wedge inv(v') \wedge rank(v,v'), \\
rank(v,v') \rightarrow ti(v,v'), \\
ti(v,v') \wedge rank(v,v') \rightarrow ti(v,v''), \\
dwf(ti).
\end{array}
$$

Coming to the new satisfaction $(p_2(v), next(v,v')) \models_{CTL} w \geq 1$, we see that its formula $w \geq 1$ is an assertion with no temporal operators. Since no further decomposition is possible, we apply RULECTLINIT from Figure 3 to generate directly the clause:

$$p_2(v) \rightarrow w \geq 1$$

As the original satisfaction $(init(v), next(v,v')) \models_{CTL} AG(EF(w \geq 1))$ is reduced into the satisfactions $(init(v), next(v,v')) \models_{CTL} AG(p_1(v))$, $(p_1(v), next(v,v')) \models_{CTL} EF(p_2(v))$ and $(p_2(v), next(v,v')) \models_{CTL} w \geq 1$, the constraints for the original satisfaction will be the union of the constraints for each of the decomposed satisfactions. The Horn constraints are over the auxiliary assertions $p_1(v)$, $inv_1(v)$, $p_2(v)$,

$inv_2(v)$, $rank(v,v')$, and $ti(v,v')$, and they are given below.

$$
\begin{array}{l}
init(v) \rightarrow inv_1(v),\\
inv_1(v) \wedge next(v,v') \rightarrow inv_1(v'),\\
inv_1(v) \rightarrow p_1(v),\\
p_1(v) \rightarrow inv_2(v),\\
inv_2(v) \wedge \neg p_2(v) \rightarrow \exists v' : next(v,v') \wedge inv(v') \wedge rank(v,v'),\\
rank(v,v') \rightarrow ti(v,v'),\\
ti(v,v') \wedge rank(v,v') \rightarrow ti(v,v''),\\
dwf(ti)\\
p_2(v) \rightarrow w \geq 1
\end{array}
$$

This will be the final output of our Horn constraint generation procedure.

5 Evaluation

We evaluate our method of CTL verification by applying the implementation of the E-HSF solver on a set of industrial benchmarks from [9, Figure 7]. These benchmarks consists of seven programs: `Windows OS fragment 1`, `Windows OS fragment 2`, `Windows OS fragment 3`, `Windows OS fragment 4`, `Windows OS fragment 5`, `PostgreSQL pgarch` and `Software Updates`. For each of these programs, four slightly different versions are considered for evaluation. In general, the four versions of a given program are the same in terms of the main logic of the program and what the program does, but they may differ on the value assigned to a particular variable or the condition for exiting a loop, etc. This gives us in total a set of 28 programs. Each such program P is given with a CTL property φ, and there are two verification tasks associated with it: $P \models_{CTL} \varphi$ and $P \models_{CTL} \neg\varphi$. The existence of a proof for a property φ for P implies that $\neg\varphi$ is violated by the same program P, and similarly, a proof for $\neg\varphi$ for P implies that φ is violated by P. However, it may also be the case that both $P \not\models_{CTL} \varphi$ and $P \not\models_{CTL} \neg\varphi$ do not hold.

Templates: As discussed in section 2.2, E-HSF requires the template functions to be provided by the user for relations with existentially quantified variables. For the application of CTL verification, which is the main topic of interest in the paper, we claim that the transition relation $next(v,v')$ can be used as a template by adding constraints at each location of non-determinism. There are two kinds of constraints that can be added depending on the two types of possible non-determinism in $next(v,v')$.

- **non-deterministic guards:** this is the case when $next(v,v')$ has a set of more than one disjuncts with the same guard, i.e., there can be more than one enabled moves from a certain state of the program. For each such set, we introduce a fresh case-splitting variable and we strengthen the guard of each disjunct by adding a distinct constraint on the fresh variable. For example, if the set has n disjuncts and B is a fresh variable, we add the constraint $B = i$ for each disjunct i where $1 \leq i \leq n$. To reason about existentially quantified queries, then it will suffices to instantiate B to one of the values in the range $1 \ldots n$. Such reasoning is done by the E-HSF solver.
- **non-deterministic assignments:** this is the case when $next(v,v')$ has a disjunct in which some w', which is a subset of v', is left unconstrained in the disjunct. In such case, we strengthen the

disjunct by adding the constraint $x' = T_x * v + t_x$ as conjunct for each variable x' in w'. Solving for T_x and t_x is done by the E-HSF solver.

In our CTL verification examples, both non-deterministic guards and assignments are explicitly marked in the original benchmark programs using names `rho1`,`rho2`, etc. We apply the techniques discussed above to generate templates from the transition relation of each program. In these examples, linear templates are sufficiently expressive. For direct comparison with the results from [9], we used template functions corresponding to the `rho`-variables. The quantifier elimination in $\exists v' : next(v, v')$ can be automated for the theory of linear arithmetic. For dealing with well-foundedness we use linear ranking functions, and hence corresponding linear templates for DECREASET and BOUNDT.

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		Result	Time(s)	Result	Time(s)
Windows OS fragment 1 (29 LOC)	$AG(p \rightarrow AFq)$	✓	0.3	×	0.3
	$EF(p \wedge EGq)$	✓	0.3	×	0.3
	$AG(p \rightarrow EFq)$	✓	0.3	×	0.3
	$EF(p \wedge AGq)$	✓	0.3	×	0.3
Windows OS fragment 2 (58 LOC)	$AG(p \rightarrow AFq)$	✓	0.4	×	0.3
	$EF(p \wedge EGq)$	✓	0.4	×	0.3
	$AG(p \rightarrow EFq)$	✓	0.4	×	0.3
	$EF(p \wedge AGq)$	✓	0.4	×	0.3
Windows OS fragment 3 (370 LOC)	$AG(p \rightarrow AFq)$	✓	0.6	×	1.2
	$EF(p \wedge EGq)$	✓	9.4	×	0.5
	$AG(p \rightarrow EFq)$	✓	0.7	×	0.8
	$EF(p \wedge AGq)$	✓	0.9	×	1.1
Windows OS fragment 4 (380 LOC)	$AFp \vee AFq$	✓	5.7	×	5.2
	$EGp \wedge EGq$	✓	0.3	×	1.0
	$EFp \wedge EFq$	✓	5.0	×	0.3
	$AGp \vee AGq$	✓	0.3	×	6.4
Windows OS fragment 5 (43 LOC)	$AG(AFp)$	✓	0.3	×	0.3
	$EF(EGp)$	✓	0.3	×	0.3
	$AG(EFp)$	✓	0.3	×	0.3
	$EF(AGp)$	✓	0.3	×	0.3
PostgreSQL pgarch (70 LOC)	$AG(AFp)$	✓	0.4	×	0.3
	$EF(EGp)$	✓	0.3	×	0.4
	$AG(EFp)$	✓	0.3	×	0.3
	$EF(AGp)$	✓	0.3	×	0.3
Software Updates (35 LOC)	$p \rightarrow EFq$	✓	0.6	×	0.2
	$p \wedge EGq$	×	0.3	×	0.4
	$p \rightarrow AFq$	×	0.2	×	0.2
	$p \wedge AGq$	×	0.3	×	0.3

Table 1: CTL verification on industrial benchmarks

We report the results in Table 1. For each program in Column 1, we report the shape of the property ϕ in Column 2. The variables p and q in Column 2 range over the theory of quantifier-free linear integer arithmetic. The result as well as the time it took the E-HSF engine to prove the property φ is

given in Columns 3 and 4, and similarly, the result as well as the time it took the engine to discover a counterexample for the negated property $\neg\varphi$ is given in Columns 5 and 6. The symbol ✓ marks the cases where E-HSF was able to find a solution, i.e., a proof that the CTL property φ is valid, and the symbol $\times$ marks the cases where E-HSF was able to find a counter-example, i.e., a proof that the negated CTL property $\neg\varphi$ is not valid. The number of LOC of each program is also given in Column 1.

The E-HSF engine is able to find proofs that the CTL property φ is valid (and the negated CTL property $\neg\varphi$ is not valid) for all of the programs except the last three programs. For the last three versions of `Software Updates`, not only the negated CTL property $\neg\varphi$ but also the CTL property φ is not valid. This was because φ was satisfied only for some initial states. The method takes a total time of 52 seconds to complete the verifications tasks.

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook [8]	E-HSF	Cook [8]
`Windows OS fragment 1` (`29 LOC`)	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
`Windows OS fragment 2` (`58 LOC`)	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
`Windows OS fragment 3` (`370 LOC`)	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
`Windows OS fragment 4` (`380 LOC`)	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
`Windows OS fragment 5` (`43 LOC`)	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
`PostgreSQL pgarch` (`70 LOC`)	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4

Table 2: Comparison of our results with Cook [8, Figure 11]

Our method also compares favourably with state-of-art automated CTL verification methods. We present in Table 2 the comparison between the our solving algorithm E-HSF and a CTL verification method from Cook [8]. Here also, we use the programs from Table 1, however, for the sake of focusing on the comparison, we exclude programs for which the two methods have different outcomes. For each program in Column 1, we report the shape of the property in Column 2. The time it takes E-HSF to prove the property φ is given in Column 3, and the corresponding time for Cook [8] is given in Column 4. Similarly, the time it takes E-HSF to discover a counterexample for the negated property $\neg\varphi$ is given in Column 5, and the corresponding time for Cook [8] is given in Column 6.

From the result, we can see that while E-HSF takes a total of 48 seconds to finish the task, Cook [8] takes a total of 149 seconds. This amounts to an approximate reduction of 70%. There are a few cases where E-HSF takes longer than Cook [8]. We suspect that a more efficient modeling of the original c program as a transition system can help our method a lot. The presence of many temporary program variables in the transition relation which are not involved in any computation of the program can affect the performance of our method.

In general, although our method uses generic horn constraints solving engine, which is not specific to CTL verification, it is able to out-perform the state-of-art automated CTL verification method.

6 Related work

Verification of properties specified in temporal logics such as CTL has been extensively explored for finite-state systems [3, 5, 6, 21]. There has also been studies on the verification of CTL properties for some restricted types of infinite-state systems. Some examples are pushdown processes [27, 28], pushdown games [29], and parameterised systems [14]. For such restricted systems, the standard procedure is to abstract the infinite-state system model into finite-state model and apply the known methods for finite-state systems. But existing abstraction methods usually do not allow reliable verification of CTL properties where alternation between universal and existential modal operators is common. Many methods of proving CTL properties with only universal path quantifiers are known [4, 10]. There also a few methods mainly focused on proving branching-time properties with only existential path quantifiers. One example is the tool Yasm [17] which implements a proof procedure aimed primarily at the non-nested existential subset of CTL. There are also known techniques for proving program termination (resp. non-termination) [2, 11] which is equivalent with proving the CTL formula $AF\ false$ (resp. $EG\,true$) [16].

The first known automatic proof method that supports both universal and existential branching-time modal operators for (possibly infinite-state) programs is proposed in [9]. The approach is based on reducing existential reasoning to universal reasoning when an appropriate restriction is placed on the the state-space of the system. While this approach comes close to our approach, the refinement procedure for state-space restrictions may make incorrect choices early during the iterative proof search. These choices may limit the choices available later in the search leading to failed proof attempts in some cases.

7 Conclusion

In this paper, we proposed a method of verifying CTL properties with respect to a (possibly infinite-space) program. The method takes a transition system that models the input program and a CTL formula specifying the property to prove as inputs. It first applies known proof systems to generate forall-exists quantified Horn constraints with well-foundedness conditions by the taking the transition system and the CTL formula. Then, it applies the solving algorithms E-HSF to solve the set of Horn constraints. The defining feature of this approach is the separation of concerns between the encoding and the solving of the verification problem. We also demonstrate the practical applicability of the approach by presenting an experimental evaluation using examples from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

References

[1] Tewodros A. Beyene, Corneliu Popeea & Andrey Rybalchenko (2013): *Solving Existentially Quantified Horn Clauses*. In: *CAV*.

[2] Aaron R. Bradley, Zohar Manna & Henny B. Sipma (2005): *Polyranking for Polynomial Loops*. *Automata, Languages and Programming*, pp. 1349–1361.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill & L.J. Hwang (1990): *Symbolic model checking: 1020 states and beyond*. In: *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pp. 428–439.

[4] Sagar Chaki, Edmund M. Clarke, Orna Grumberg, Jol Ouaknine, Natasha Sharygina, Tayssir Touili & Helmut Veith (2005): *State/Event Software Verification for Branching-Time Specifications.* In: *IFM*, 3771, Springer, pp. 53–69.

[5] E. M. Clarke, E. A. Emerson & A. P. Sistla (1986): *Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications*. *ACM Trans. Program. Lang. Syst.* 8(2), pp. 244–263.

[6] Edmund Clarke, Yuan Lu, Broadcom Com, Helmut Veith & Somesh Jha (2002): *Tree-Like Counterexamples in Model Checking*. In: *In Proceedings of the 17 th Annual IEEE Symposium on Logic in Computer Science (LICS02)*, IEEE Computer Society.

[7] Edmund M. Clarke (1991): *Temporal Logic Model Checking: Two Techniques for Avoiding the State Explosion Problem*. In: *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90.

[8] Byron Cook, Heidy Khlaaf & Nir Piterman (2014): *Faster Temporal Reasoning for Infinite-State Programs*.

[9] Byron Cook & Eric Koskinen (2013): *Reasoning about Nondeterminism in Programs*. In: *PLDI*.

[10] Byron Cook, Eric Koskinen & Moshe Vardi (2012): *Temporal Property Verification As a Program Analysis Task*. *Form. Methods Syst. Des.* 41(1), pp. 66–82.

[11] Byron Cook, Andreas Podelski & Andrey Rybalchenko (2006): *Termination proofs for systems code*. In: *PLDI*.

[12] Stphane Demri, Alain Finkel, Valentin Goranko Govert & Van Drimmelen (2010): *Model checking CTL* over flat Presburger counter systems*. *JANCL*.

[13] E. Allen Emerson (1990): *Handbook of Theoretical Computer Science (Vol. B)*. chapter Temporal and Modal Logic.

[14] E. Allen Emerson & Kedar S. Namjoshi (1996): *Automatic Verification of Parameterized Synchronous Systems (Extended Abstract)*. In: *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, Springer-Verlag, pp. 87–98.

[15] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In: *TACAS*. Available at `http://dx.doi.org/10.1007/978-3-642-28756-5_46`.

[16] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko & Ru-Gang Xu (2008): *Proving non-termination*. In: *POPL*.

[17] Arie Gurfinkel, Ou Wei & Marsha Chechik (2006): *Yasm: A Software Model-Checker for Verification and Refutation.* In Thomas Ball & Robert B. Jones, editors: *CAV*, Lecture Notes in Computer Science, Springer, pp. 170–174.

[18] Zyad Hassan, Aaron R. Bradley & Fabio Somenzi (2012): *Incremental, Inductive CTL Model Checking*. In: *Proceedings of the 24th International Conference on Computer Aided Verification*.

[19] Krystof Hoder, Nikolaj Bjørner & Leonardo Mendonça de Moura (2011): *Z- An Efficient Engine for Fixed Points with Constraints*. In: *CAV*, pp. 457–462.

[20] Yonit Kesten & Amir Pnueli (2005): *A compositional approach to CTL* verification*. *Theor. Comput. Sci.* 331(2-3), pp. 397–428. Available at `http://dx.doi.org/10.1016/j.tcs.2004.09.023`.

[21] Orna Kupferman, Moshe Y. Vardi & Pierre Wolper (2000): *An Automata-theoretic Approach to Branching-time Model Checking*. J. ACM 47(2), pp. 312–360.

[22] Zohar Manna & Amir Pnueli (1992): *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA.

[23] Kenneth L. McMillan (1993): *Symbolic Model Checking*.

[24] Wojciech Penczek, Bozena Wozna & Andrzej Zbrzezny (2002): *Bounded Model Checking for the Universal Fragment of CTL*. Fundam. Inf.

[25] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science.

[26] Fu Song & Tayssir Touili (2011): *Efficient CTL model-checking for pushdown systems*. In: In CONCUR.

[27] Fu Song & Tayssir Touili (2013): *PoMMaDe: Pushdown Model-checking for Malware Detection*. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, pp. 607–610.

[28] Igor Walukiewicz (2000): *Model Checking CTL Properties of Pushdown Systems.* In Sanjiv Kapoor & Sanjiva Prasad, editors: *FSTTCS*, Lecture Notes in Computer Science, Springer, pp. 127–138.

[29] Igor Walukiewicz (2001): *Pushdown processes: Games and model-checking*. Information and computation 164(2), pp. 234–263.

A More proof rules for constraints generation

The proof rules RULECTLAX (see Figure 8), RULECTLAG (see Figure 9), and RULECTLAU (see Figure 10) are applied for generating Horn constraints when the CTL formula is a basic state formula with universal path operator.

$$\frac{p(v) \wedge next(v,v') \rightarrow q(v')}{(p(v), next(v,v')) \models_{CTL} AX\ q(v)}$$

Figure 8: Proof rule RULECTLAX

Find an assertion $inv(v)$ such that:

$$\frac{\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \wedge next(v,v') \rightarrow inv(v') \\ inv(v) \rightarrow q(v) \end{array}}{(p(v), next(v,v')) \models_{CTL} AG\ q(v)}$$

Figure 9: Proof rule RULECTLAG

Find assertions $inv(v)$, $rank(v,v')$ and $ti(v,v')$ such that:

$$\frac{\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \wedge \neg r(v) \wedge next(v,v') \rightarrow \quad q(v) \wedge inv(v') \wedge rank(v,v') \\ rank(v,v') \rightarrow ti(v,v'), \\ ti(v,v') \wedge rank(v',v'') \rightarrow ti(v,v''), \\ dwf(ti). \end{array}}{(p(v), next(v,v')) \models_{CTL} AU(q(v), r(v))}$$

Figure 10: Proof rule RULECTLAU

Our proof system is not exhaustive in terms of having proof rules for all kinds of basic state formula that can be defined in CTL. However, we utilize equivalence between CTL formulas to generate Horn constraints for a basic state formula whose proof rule is not given in the proof system. The equivalence between the formulas $EU(true, q(v))$ and $EF(q(v))$ is used to define RULECTLEF (see Figure 11) from

RULECTLEU. In the same way, the equivalence between the formulas $AU(true, q(v))$ and $AF(q(v))$ is used to define RULECTLAF (see Figure 12) from RULECTLAU.

$$\frac{(p(v), next(v,v')) \models_{CTL} EU(true, q(v))}{(p(v), next(v,v')) \models_{CTL} EF\ q(v)}$$

Figure 11: Proof rule RULECTLEF

$$\frac{(p(v), next(v,v')) \models_{CTL} AU(true, q(v))}{(p(v), next(v,v')) \models_{CTL} AF\ q(v)}$$

Figure 12: Proof rule RULECTLAF