A

Verifying Increasingly Expressive Temporal Logics for Infinite-State Systems

Byron Cook, University College London
Heidy Khlaaf, University College London
Nir Piterman, University of Leicester

Temporal logic is a formal system for specifying and reasoning about propositions qualified in terms of time. It offers a unified approach to program verification as it applies to both sequential and parallel programs and provides a uniform framework for describing a system at any level of abstraction. Thus a number of automated systems have been proposed to exclusively reason about either Computation-Tree Logic (CTL) or Linear Temporal Logic (LTL) in the infinite-state setting. Unfortunately, these logics have significantly reduced expressiveness as they restrict the interplay between temporal operators and path quantifiers, thus disallowing the expression of many practical properties, for example "along some future an event occurs infinitely often". Contrarily, CTL*, a superset of both CTL and LTL, can facilitate the interplay between path-based and state-based reasoning. CTL* thus exclusively allows for the expressiveness of properties involving existential system stabilization and "possibility" properties. Until now, there have not existed automated systems that allow for the verification of such expressive CTL* properties over infinite-state systems. This paper proposes a method capable of such a task, thus introducing the first known fully automated tool for symbolically proving CTL* properties of (infinite-state) integer programs. The method uses an internal encoding that admits reasoning about the subtle interplay between the nesting of temporal operators and path quantifiers that occurs within CTL* proofs. A program transformation is first employed that trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. We then synthesize and quantify preconditions over the transformed program that represent program states that satisfy a CTL* formula.

This paper demonstrates the viability of our approach in practice, thus leading to a new class of fully-automated tools capable of proving crucial properties that no tool could previously prove. Additionally, we consider the linear-past extension to CTL* for infinite-state systems in which the past is linear and each moment in time has a unique past. We discuss the practice of this extension and how it is further supported through the use of history variables. We have implemented our approach and report our benchmarks carried out on case studies ranging from smaller programs to demonstrate the expressiveness of CTL* specifications, to larger code bases drawn from device drivers and various industrial examples.

CCS Concepts: •**Theory of computation → Modal and temporal logics; Verification by model checking;** *Logic and verification;*

Additional Key Words and Phrases: Model Checking, CTL*

ACM Reference Format:
Byron Cook, Heidy Khlaaf, and Nir Piterman, 2017. Verifying Increasingly Expressive Temporal Logics for Infinite-State Systems. *J. ACM* V, N, Article A (January YYYY), 39 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

Contents

1. INTRODUCTION

1.1. Context and Motivation

In [Pnueli 1977], Amir Pnueli introduced the idea of utilizing temporal logic as a unifying approach to program analysis for both sequential and parallel programs. He suggested that temporal reasoning, in which propositions are qualified in terms of time, allows for the logical basis of proving correctness properties of programs. Additionally, temporal logic formalizes the intuitive reasoning that a programmer employs in the design and implementation of programs and systems. This lead to a surge of interest in the use of static analysis techniques to automatically verify various temporal logics, both for finite-state and infinite-state systems.

Pnueli came to find that the prevalent notions of what constitutes the correctness of a program can all be reduced to two main temporal concepts: invariance and eventuality. In [Lamport 1980], Leslie Lamport further refines these concepts as safety and liveness, respectively. Safety, for example, covers the concepts of partial correctness (something bad never happens) for sequential programs, mutual exclusion (two processes are not in their critical sections at the same time), and deadlock-freedom (the program does not reach a deadlocked state) for concurrent programs. Liveness covers the concepts of total correctness in addition to the generalization of correct behaviors for programs that contain loops. For

example, termination (the program eventually does terminate) and starvation-freedom (a process eventually serves) are liveness properties.[1]

In addition to supporting correctness properties of programs, temporal logic allows for hierarchical specification and reasoning. From a developer's standpoint, natural languages are very expressive yet very imprecise. Contrarily, formal languages are not very expressive, but they are precise. Temporal logic thus bridges the expression and precision gap by providing a single logical system for describing the program at any level of abstraction, from the highest-level specification to the programming-language implementation. A statement about the program at one level is a meaningful statement about any lower level. Thus, hierarchical design methods are supported directly, with no extra mechanism needed to bridge the different levels of description. One such temporal logic is the branching-time logic CTL. Branching-time property verification requires reasoning about sets of *states* within a transition system that satisfy a particular temporal formula. Linear-time logic, the most common being LTL, requires reasoning about sets of *paths* that satisfy a formula. However, these logics have significantly reduced expressiveness as they can only reason about either states or paths, but not the junction of both. Contrarily, CTL*, a superset of both LTL and CTL, can facilitate the interplay between state-based and path-based reasoning.

Proof systems for the verification of temporal logic, first introduced by [Emerson and Halpern 1986; Lamport 1980], have been well-studied. It is well-known that CTL* model checking for infinite-state systems generalizes termination and co-termination and is undecidable. A decision procedure exploring the structure of finite-state ω-automata was first introduced to determine the satisfaction of a CTL* formula over binary relations in [Emerson and Sistla 1984], and later extended in [Emerson and Jutla 1999]. A complete and sound axiomatization of propositional CTL* then followed in [Reynolds 2001], which inspired the first sound and relatively complete deductive proof system for the verification of CTL* properties over possibly infinite-state systems [Kesten and Pnueli 2005]. Proof rules for verifying CTL* properties of infinite-state systems were implemented in STeP [Bjørner et al. 2000]. However, the STeP system is only semi-automated, as it still requires users to construct auxiliary assertions and participate in the search for a proof.

Until now, no fully automatic CTL* proving methods for the undecidable general class of infinite-state systems have been known. Fully automated implementations of these proof systems have proven to be difficult over the years. Furthermore, despite the existence of automated verification tools for CTL and LTL for general integer manipulating programs [Beyene et al. 2013; Cook et al. 2007; Cook and Koskinen 2011; 2013; Cook et al. 2014; 2015], these tools still do not allow for the verification of CTL*. A key problem is that CTL* formulae cannot merely be partitioned into isolated CTL and LTL sub-formulae, as such a partition fails to treat the intricate dependence between state-based and path-based reasoning. Finding a way that allows us to symbolically move between representations of sets of states for branching-time, and sets of paths for linear-time in a way that is conducive to automatic analysis has thus been an outstanding problem in automatic program verification. This restriction on the interplay between linear-time and branching-time operators causes various crucial properties to be inexpressible. Consider a property involving the assertion "along *some* future an event occurs *infinitely often*". This property cannot be expressed in either LTL or CTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in an infinite-state system. However, this property is expressible in CTL*.

[1]Note that the first formal definition of both safety and liveness properties was not introduced until [Alpern and Schneider 1986]. Alpern and Schneider represent a finite prefix of an execution as the set of all possible continuations from that point on. This contrasts to the notions in [Lamport 1980], which only consider infinite executions. This leads to a slightly more general notion of safety and liveness properties, with liveness properties containing at least one continuation for every finite prefix.

In section 1.2, we further demonstrate how CTL* is capable of expressing CTL, LTL, and properties necessitating their interplay.

In this paper we address the application of CTL* properties in the (integer) infinite-state setting, in addition to introducing a solution that admits the arbitrary nesting of state-based reasoning within path-based reasoning, and vice versa. Our strategy allows us to symbolically move between representations of sets of states and sets of paths, thus leading to the first known fully-automatic method capable of proving CTL* properties of infinite-state programs.

1.2. Expressiveness of Temporal Logics and Their Applications to Programs

In order to discern the difficulty of partitioning the subtle interplay between the arbitrary nesting of path and state formulae, we must first delve into the differing temporal sub-logics and their relation to each other. Note that for finite-state systems, the distinctions between branching-time and linear-time logics are less crucial from an automation standpoint, as verifying these logics is decidable. However, when considering the undecidable general class of infinite-state systems (e.g., systems with unbounded arithmetic), the distinction is a key issue. CTL and LTL are the most well-studied temporal logics given that they each only require a homogenous approach to reason about computational systems. As previously mentioned, branching-time logic requires reasoning about sets of *states* while linear-time logic requires reasoning about sets of *paths*. The two interpretations correspond to two different ways of viewing time: as a continually branching set of possibilities, or as a single linear sequence of actual events, respectively. In the branching time approach, all of the possible futures are equally real and must be considered. Thus, when considering nondeterministic transition systems (as we do), the present does not determine a unique future, but rather a possibly infinite set of possible futures given that nondeterminism translates to many possible computations. In the linear time approach, at each instance of time there is only one future that will actually occur. Given that all assertions must be interpreted over one real future, path quantifiers are thus not required for linear-time logics.

Despite their discrepancies, the expressiveness of CTL and LTL are incomparable given that they simply provide differing interpretations of time. However, the restriction these two logics impose on the interplay between linear-time operators and path quantifiers disallows a great deal of properties vital to appropriately expressing the correctness of a system. For example, although CTL can express a system's interaction with inputs and nondeterminism, a capability in which linear-time temporal logics (LTL) is inadequate to express, it cannot model trace-based assumptions about the environment in sequential and concurrent settings (e.g. schedulers) that LTL can express. Some of these deficiencies can be mitigated by considering fairness for branching-time logic (CTL + Fair), as it allows for some interaction between linear-time and branching-time reasoning, but only in specifying fairness assumptions pertaining to a system's environment. CTL + Fair thus cannot be generalized to model all trace-based properties. In recent work, we have considered practical applications of CTL + Fair model-checking [Cook et al. 2015]. In the sections below, we provide further examples of properties exclusive to CTL* in addition to an analysis of the crucial application of CTL* properties in the infinite-state setting.

1.2.1. Expressiveness of CTL.* First, we briefly give an informal description of CTL* syntax to allow the reader to more intuitively understand the provided examples. Formal definitions are provided in Section 2. CTL* formulae are made up of path quantifiers and temporal operators. There exists two types of path quantifiers: *All*, written as $\mathsf{A}\psi$, indicates that ψ has to hold on all paths starting from a state. *Exists*, written as $\mathsf{E}\psi$ indicates that there exists at least one path starting from a state where ψ holds. For both A and E, ψ denotes a temporal formula, however in CTL*, not every temporal operator has to be preceded by a path quantifier. Temporal operators include:

— *Next* or $\mathsf{X}\psi : \psi$ has to hold starting from the next position in a path.
— *Globally* or $\mathsf{G}\psi : \psi$ has to hold starting from all the positions along a path.
— *Eventually* or $\mathsf{F}\psi : \psi$ eventually has to hold.
— *Strong Until* or $\psi_1\ \mathsf{U}\ \psi_2 : \psi_1$ has to hold starting from all positions until at some position ψ_2 starts to hold. ψ_2 must be verified in the future.
— *Weak Until* or $\psi_1\ \mathsf{W}\ \psi_2 : \psi_1$ has to hold starting from all positions until at some position ψ_2 starts to hold. Unlike the strong until, ψ_2 does not have to be verified and, if such is the case, then ψ_1 has to hold forever.

The linear-time logic LTL is a fragment of CTL* that only allows formulae of the form $\mathsf{A}\psi$, where A is the only occurrence of a path quantifier within ψ. When taking LTL as a subset of CTL*, LTL formulae are implicitly prefixed with the universal path quantifier A. For example, the LTL formula $\mathsf{FG}\ x$ asserts that for every trace of the system, variable x will eventually become true and stay true forever. The branching-time logic CTL is a restricted subset of CTL* in which a temporal operator is always directly preceded by a path quantifier. Thus, CTL sub-formulae are always composed of pairs containing a path quantifier and a temporal operator. For example, the CTL formula $\mathsf{EF}\ x$ is true in states from which there exists a path where eventually there is a state in which x holds. Recall that CTL* allows the unrestricted nesting of path quantifiers and temporal operators.

CTL* thus allows us to express properties involving existential system stabilization, stating that an event can eventually become true and stay true from every reachable state. Additionally, it can express "possibility" properties, such as the viability of a system, stating that every reachable state can spawn a fair computation. Below are properties that can only be afforded by the extra expressive power of CTL*. These liveness properties are often imperative to verifying systems such as Windows kernel APIs that acquire resources and APIs that release resources, as later shown by our experiments.

The property $\mathsf{EGF}(x)$ asserts that there *exists* some path such that x holds *infinitely often* along the path. This property is not expressible in CTL nor in LTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in a system. The CTL approximation $\mathsf{EGAF}\ x$ differs subtly in that it requires that there exists a path such that from all states along the path, x will *eventually* be reached for *all futures*. In LTL one can try to approximate a solution by trying to *disprove* $\mathsf{FG}\ \neg x$. However, this approach only goes so far, *e.g.* we cannot nest the property within another path quantifier, further stressing the expressive deficiency of LTL.

The property $\mathsf{EFG}(\neg x \wedge (\mathsf{EGF}\ x))$, results from nesting the property $\mathsf{EGF}(x)$ inside a larger formula, and conveys the divergence of paths. That is, there is a path in which a system stabilizes to $\neg x$, but every point on said path has a diverging path in which x holds infinitely often. This property is expressible neither in CTL nor in LTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in a system. In CTL, one can only examine sets of states, disallowing us to convey properties regarding paths. The CTL approximation $\mathsf{EFAG}(\neg x \wedge (\mathsf{EGAF}\ x))$ differs in that it requires that there is a state such that from *all* states along the path, a system stabilizes to $\neg x$, yet from every point on said path, *all* states have a diverging path in which x holds infinitely often, thus inducing a property that is unsatisfiable. The slightly weaker $\mathsf{EFEG}(\neg x \wedge (\mathsf{EGAF}\ x))$ is also unsatisfiable. The CTL under approximation $\mathsf{EFEG}(\neg x \wedge (\mathsf{EGEF}\ x))$ does indeed entail that there is a path in which a system stabilizes to $\neg x$, yet from every point on said path there exists a state in which x holds at least once. This under approximation is thus not sufficient given that it cannot satisfy that x must hold infinitely often in the diverging path. In LTL, one cannot approximate a solution by trying to *disprove* either $\mathsf{FG}\ \neg x$ or $\mathsf{GF}\ x$, as one cannot characterize these proofs within a path quantifier.

Another CTL* property $\mathsf{AG}\big[(\mathsf{EG}\ \neg x) \vee (\mathsf{EFG}\ y)\big]$ dictates that from every state of a program, there exists either a computation in which x never holds or a computation in which

y eventually always holds. The linear time property $\mathsf{G}(\mathsf{F}x \rightarrow \mathsf{FG}\ y)$ is significantly stricter as it requires that on every computation either the first disjunct or the second disjunct hold. Finally, the property $\mathsf{EFG}\big[(x \vee (\mathsf{AF}\ \neg y))\big]$ asserts that there exists a computation in which whenever x does not hold, all possible futures of a system lead to the falsification of y. This assertion is impossible to express in LTL. In Section 8, we further demonstrate the use of the aforementioned properties on I/O subsystems of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server.

1.2.2. Extending CTL* to Support Linear Past. In the philosophical context of which they were developed [Kamp 1968; Prior 1957], temporal logics have always provided temporal connectives that refer to both the past and the future. Yet in the context of system verification, past connectives have been often cast aside for the sake of minimality since they add no expressive power to linear temporal logics given that a computation always has a definite starting time and a unique past [Gabbay et al. 1980]. However, specifying temporal formulae can often become overly convoluted when specifying a system's correctness properties, thus rendering the intuitiveness and preciseness of temporal logic obsolete. Extending CTL* to admit past-time connectives thus allows for exponentially more succinct temporal formulae [Kupferman et al. 2012]. Furthermore, past-time connectives are known to make the formulation of specifications more intuitive [Lichtenstein et al. 1985].

As with future sub-logics, there exist two interpretations corresponding to two possible views regarding the nature of the past. In branching-time past (CTL^*_{bp}), past is branching and each moment in time may have several possible futures and several possible pasts. In linear-time past (CTL^*_{lp}), past is linear and each moment in time may have several possible futures and a unique past. Both views assume that past is finite. Extensions to branching-past connectives have been extensively studied [Kupferman et al. 2012]. The effect of adding such connectives on the expressiveness and computational complexity of CTL* differs from the linear-past results. Branching-past adds expressive power to CTL* (CTL^*_{bp}), while model-checking finite-state systems for CTL^*_{bp} is in PSPACE, and its satisfiability is in 2EXPTIME [Kupferman et al. 2012]. These are the same known complexities as of CTL*'s. We have not found CTL^*_{bp} to have beneficial expressiveness to the specific properties we wish to verify, thus we do not address the extension CTL^*_{bp} in this paper. Instead, we consider the linear-past extension CTL^*_{lp} for infinite-state systems in which the past is linear and each moment in time has a unique past. Specifically, we consider a fragment of CTL^*_{lp} in which the addition of linear-past connectives to CTL* (CTL^*_{lp}) does not increase the complexity of the satisfiability result for finite-state systems [Kupferman et al. 2012]. Yet supporting linear-past connectives is still sufficiently beneficial given that it enriches temporal logics with more intuitive and succinct specifications. Automata-theoretic algorithms for the verification of CTL^*_{lp} properties over finite-state systems have been introduced in [Bozzelli 2008; Kupferman et al. 2012], however we are not aware of implementations of these techniques. Additionally, we are not aware of any tools that considers past temporal operators in model checking for infinite-state programs.

An example of how CTL^*_{lp} can allow us to succinctly and intuitively express properties concerns the verification of a Windows device driver taken from [Ball et al. 2006], where a property requires that drivers mark an I/O request packet as pending (using `IoMarkIrpPending`) before queuing it, that is:

$$\mathsf{AG}(\texttt{Queue(Irp)} \Rightarrow \mathsf{X}^{-1}\ (\neg\texttt{Queue(Irp)}\ \mathsf{U}^{-1}\ \texttt{IoMarkIrpPending(Irp)}))$$

However, the property written solely in future-connectives would be:

$$\neg\texttt{Queue(Irp)}\ \mathsf{W}\ \texttt{IoMarkIrpPending(Irp)} \wedge$$
$$\mathsf{G}(\texttt{Queue(Irp)} \rightarrow \mathsf{X}(\neg\texttt{Queue(Irp)}\ \mathsf{W}\ \texttt{IoMarkIrpPending(Irp)}))$$

Note that we would be required to keep track of every queuing action, denoted by `Queue(Irp)`, and ensure a queuing call cannot be made until a call to `IoMarkIrpPending(Irp)` has been made. If a queuing call has indeed been made, then we must ensure that future queuing calls cannot be additionally made until additional calls to `IoMarkIrpPending(Irp)` have been made. A future-connective formulation is thus less intuitive and succinct when compared to its past-connective alternative. Using past connectives, we simply ensure that if we encounter a queuing call, then the I/O request packet has been previously marked as pending, with no other queuing calls in between.

Further on, we discuss the support of a fragment of CTL^*_{lp} using history variables, and provide further examples of its usage. The fragment we tackle merely restricts that the newly introduced past formulae be immediately followed by either an additional past formula, or a state formula. That is, we disallow referring to the future of a path within a past formula. Note that this does not affect the nesting of state formulae or the further nesting of past formulae within a CTL^*_{lp} property. We discuss the reasons for such a limitation in further sections.

We have noted earlier that adding linear-past connectives to CTL^* adds no additional expressive power given that a computation always has a definite starting time and a unique past [Gabbay et al. 1980; Kupferman et al. 2012]. One may speculate that an automated translation from CTL^*_{lp} to CTL^* is a more suitable strategy than embedding additional history variables, however, we believe this not to be the case. For CTL^*_{lp}, and more specifically the fragment in which we tackle, the translation itself is non-elementary, and a translation algorithm may induce combinatorial explosions, even with limited temporal height [Laroussinie and Schnoebelen 1995; Kupferman et al. 2012]. The known lower bounds for conversion from temporal logic with past to temporal logic without past are expressible in the fragment we consider [Laroussinie and Schnoebelen 1995]. It follows that already supporting this fragment offers succinctness of expression. Additionally, the conversion of linear-past connectives to future connectives would likely produce sub-formulae resembling the history variables we introduce. Hence, a translation strategy with an accompanying combinatorial formulae explosion would not be beneficial in practice. We are not aware of any implementations of the (non-elementary) translation from LTL with past, to LTL (and clearly none for CTL^*_{lp}).

1.3. Approach and Contribution

Our main contribution is an automated model-checking method that allows for the arbitrary nesting of state-based reasoning within path-based reasoning, and vice versa. Our strategy is to recursively partition a CTL^* formula, and for each nested sub-formula synthesize a precondition that ensures its satisfaction. The nested sub-formula would then be substituted with its new-found precondition, and the process would be repeated for the next outer sub-formula. The essence of our algorithm thus lies within acquiring sufficient preconditions for path formulae that admit a sound interaction with state formulae. Towards this purpose we recursively deconstruct a CTL^* formula in a way that allows us to determine where the subtle interplay between the arbitrary nesting of path and state formulae occurs. To reason about the path sub-formulae, we find a sufficient set of branching nondeterministic decisions within a program's transition relation. We then devise a method of *temporarily* substituting said nondeterministic decisions with a *symbolically partially-determinized* form. That is, nondeterministic decisions regarding which paths are taken are determined by variables that summarize some decisions regarding the future of the program execution. When interchanging between path and state formulae, these determinized relations must then be collapsed to incorporate path quantifiers. Preconditions for the given CTL^* property can then be acquired via existing CTL model checkers.

Furthermore, we extend our CTL^* algorithm to support part of CTL^*_{lp} via the instrumentation of a unique history variable per past connective present within a CTL^*_{lp} formula. The history variable tracks the state of the consequent nested temporal formula. If the consequent nested formula within a past connective is a state sub-formula, the history variable is satisfied based on the state of the sub-formula's synthesized preconditions. However if the nested formula is another past connective, the history variable is satisfied based on the state of an additional history variable associated with the sub-formula. The satisfaction of a history variable is clearly dependent on which past connective is being verified. Additionally, the history variables are analyzed within a larger context of a future formula, that is, they are instrumented when verifying a future CTL^* sub-formula which incorporates their corresponding past sub-formulae. For the sake of clarity, we will first demonstrate our technical contributions of exclusively verifying CTL^*, followed by further sections demonstrating how we can extend our algorithm to support this part of CTL^*_{lp}.

Based on our approach, we have developed a tool capable of automatically proving properties of programs that no tool could previously fully automate. The paper closes with a description of our experimental results using the developed tool on various programs drawn from industrial examples. Our tool is available under the MIT open-source license at https://github.com/hkhlaaf/T2/.

Limitations. Our tool does not support programs with heap, nor do we support recursion or concurrency. The heap-based programs we consider during our experimental evaluation have been abstracted using an over-approximation technique introduced by [Magill et al. 2007]. Effective techniques for proving temporal properties of programs with heap remains an open research question. Our technique relies on the availability of CTL model checking and non-termination procedures. It is, in principle, applicable to every class of infinite-state systems for which such procedures are available (provided that integer variables are allowed). Additionally, our procedure is not complete as we use a series of techniques for safety [McMillan 2006], termination [Podelski and Rybalchenko 2004; Cook et al. 2013], nontermination [Gupta et al. 2008], and CTL [Beyene et al. 2013; Cook et al. 2014] that are not complete. Furthermore, our determinization procedure is not complete. We will address this issue in later sections.

2. PRELIMINARIES

2.1. Defining Programs and Transition Systems

As is standard [Manna and Pnueli 1995], we treat programs as control-flow graphs, where edges are annotated by the updates they perform to variables. A program is a triple $P = (\mathcal{L}, E, \mathsf{Vars})$, where $\mathcal{L}$ is a set of locations, E is a set of edges/transitions, and Vars is a set of variables ranging over domain Vals. Each edge $\tau = (\ell, \rho, \ell')$ in E, where $\ell, \ell' \in \mathcal{L}$ and ρ is a condition, specifies possible transitions in the program. The condition ρ is an assertion in terms of Vars and Vars', a primed copy of Vars, where constants range over Vals. In general, Vars refers to the values of variables before an update and Vars' refers to the values of variables after an update. We use a similar notation for conditions, *i.e.*, if a is a condition over Vars, a' is the same condition where every reference to a variable v is replaced by reference to v'. For example, if a is $x = 5$ then a' would be $x' = 5$. Note that we do not give exact details pertaining an assertion language as the technique (per-se) is not limited to a specific one. Concretely, our implementation which depends on existing model-checker technology, uses linear constraints over variables of the program. For example, $x' = x + 1$ or $x' > 0$ are possible assertions that would be handled by our model-checker while more complicated assertions, such as $x' = x^2$, would not be.

The set of locations $\mathcal{L}$ includes the first location ℓ_I, which has no incoming transitions from other program locations. That is, for every $\tau = (\ell, \rho, \ell') \in E$ we have $\ell' \neq \ell_I$. Transitions exiting ℓ_I have their conditions expressed in terms of Vars'. Locations with incoming

transitions from ℓ_I are *initial locations*. This allows us to encode more complex initial conditions. In figures, we omit ℓ_I and merely display the edges to locations with incoming transitions from ℓ_I.

A program gives rise to a transition system $T = (S, R)$, where S is the set of program states of the form $S = (\mathcal{L} - \{\ell_I\}) \times (\mathsf{Vars} \to \mathsf{Vals})$ and $R \subseteq S \times S$. A program state s is a pair (ℓ, f) where $\ell \neq \ell_I$ and f is a valuation, i.e., a function from program variables to values. That is, a state of a program is described by a certain location in the control-flow graph and its corresponding variable valuation. Assertions, as above, represent both sets of program states and updates to program variables. Assertions can relate variables to their values and can refer to locations from the control-flow graph (but not necessarily). Assertions that refer to locations treat locations as boolean propositions. For example, $\ell_1 \wedge x = 5$ is an assertion describing all states in which the program is in ℓ_1 and the value of variable x is 5. The assertion $x' = x + 1$ describes the update of the value of x by incrementing it by 1. With regards to locations, a primed location indicates that it is the target transition. Later on, we use the term *preconditions* for assertions that describe sets of states of the program that satisfy a certain property, and thus probably include reference to locations.

A program can transition from (ℓ, f_1) to (ℓ', f_2) if there exists a transition $(\ell, \rho, \ell') \in E$ such that $(f_1, f_2) \models \rho$. The valuation (f_1, f_2) is a function from $\mathsf{Vars} \cup \mathsf{Vars}'$ to Vals such that for every $v \in \mathsf{Vars}$, $(f_1, f_2)(v) = f_1(v)$ and $(f_1, f_2)(v') = f_2(v)$. A state (ℓ, f) is considered initial if there is a transition (ℓ_I, ρ, ℓ) such that $(f_{-1}, f) \models \rho$, where f_{-1} is some arbitrary valuation. Notice that in this case ρ is expressed in terms of Vars' and hence the valuation f_{-1} does not affect the satisfaction of ρ.

Given $V \subseteq \mathsf{Vars}$, the valuation obtained from f by restricting the valuation to variables in V is denoted by $f{\Downarrow_V}$. The restriction of states of the form (ℓ, f) and paths in the program is defined similarly, e.g., $\pi{\Downarrow_V}$.

Paths. A *path* or a *trace* π in P is an infinite sequence of states $(\ell_0, f_0), (\ell_1, f_1), \ldots$, where for every $i \geq 0$, there exists some $(\ell_i, \rho_i, \ell_{i+1}) \in E$ where $(f_i, f_{i+1}) \models \rho_i$. We say that π is an (ℓ, f)-path if $\ell_0 = \ell$ and $f_0 = f$. Given a program P, a location ℓ, and a valuation f, we denote the set of (ℓ, f)-paths in P by $\mathsf{Path}(P, \ell, f)$. We say that π is a computation in P if (ℓ, f) is initial. Note that we restrict our attention to infinite paths and computations. In practice, we modify programs, transition systems, and temporal logic formulae to ensure that all paths are infinite, as is done, e.g., in [Cook et al. 2015].

2.2. CTL* Syntax and Semantics

We are interested in verifying full computation tree logic (CTL*) [Lamport 1980; Emerson and Halpern 1986]. The syntax of CTL* (written in negation normal form) includes state formulae φ, that are interpreted over states, and path formulae ψ, that are interpreted over paths. We assume that atomic propositions (ranged over by α) are expressed in some underlying theory over variables and constants (*e.g.* $\mathsf{x} < \mathsf{y}$). State formulae (φ) and path formulae (ψ) are co-defined:

$$\begin{array}{rcl} \varphi & ::= & \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid \mathsf{E}\psi \\ \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{G}\psi \mid \mathsf{F}\psi \mid \mathsf{X}\psi \mid [\psi \mathsf{W} \psi] \mid [\psi \mathsf{U} \psi] \end{array}$$

For a program P and a CTL* state formula φ, we say that φ holds at a state s in P, denoted by $P, s \models \varphi$ if:

— If $\varphi = \alpha$, then $P, s \models \alpha$ iff $s \models \alpha$
— If $\varphi = \neg\alpha$, then $P, s \models \neg\alpha$ iff $s \not\models \alpha$
— If $\varphi = \varphi_1 \vee \varphi_2$, then $P, s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$
— If $\varphi = \varphi_1 \wedge \varphi_2$, then $P, s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$
— If $\varphi = \mathsf{A}\psi$, then $P, s \models \mathsf{A}\psi$ iff $\forall \pi = (s, \ldots).\ P, \pi \models \psi$
— If $\varphi = \mathsf{E}\psi$, then $P, s \models \mathsf{E}\psi$ iff $\exists \pi = (s, \ldots).\ P, \pi \models \psi$

Path formulae are interpreted over paths. For a program P and a CTL^* path formula ψ, we say that ψ holds on a path $\pi = (s_0, s_1, \ldots)$ in P for location i, denoted by $P, \pi, i \models \psi$ if:
— If $\psi = \varphi$ is a state formula, then $P, \pi, i \models \varphi$ iff $P, s_i \models \varphi$.
— If $\psi = \psi_1 \vee \psi_2$, then $P, \pi, i \models \psi_1 \vee \psi_2$ iff $P, \pi, i \models \psi_1$ or $P, \pi, i \models \psi_2$
— If $\psi = \psi_1 \wedge \psi_2$, then $P, \pi, i \models \psi_1 \wedge \psi_2$ iff $P, \pi, i \models \psi_1$ and $P, \pi, i \models \psi_2$
— If $\psi = \mathsf{X}\psi_1$, then $P, \pi, i \models \mathsf{X}\psi_1$ iff $P, \pi, i+1 \models \psi_1$
— If $\psi = \mathsf{F}\psi_1$, then $P, \pi, i \models \mathsf{F}\psi_1$ iff $\exists j \geq i.\ P, \pi, j \models \psi_1$
— If $\psi = \mathsf{G}\psi_1$, then $P, \pi, i \models \mathsf{G}\psi_1$ iff $\forall j \geq i.\ P, \pi, j \models \psi_1$
— If $\psi = \psi_1 \mathsf{W} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{W} \psi_2$ iff either $\exists k \geq i.\ P, \pi, k \models \psi_2$ and $\forall i \leq j < k.\ P, \pi, j \models \psi_1$ or $\forall j \geq i.\ P, \pi, j \models \psi_1$
— If $\psi = \psi_1 \mathsf{U} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{U} \psi_2$ iff $\exists k \geq i.\ P, \pi, k \models \psi_2$ and $\forall i \leq j < k.\ P, \pi, j \models \psi_1$

A path formula ψ holds in a path π, denoted by $P, \pi \models \psi$, if $P, \pi, 0 \models \psi$. For a state formula φ, φ holds on P, denoted by $P \models \varphi$, if for every initial state s we have $P, s \models \varphi$. When the program P is clear from the context, we may write $s \models \varphi$ for a state formula φ or $\pi, i \models \psi$ for a path formula ψ.

The branching-time logic CTL is a restricted subset of CTL^* in which temporal operators cannot be nested. That is, the only path formulae allowed are $\mathsf{G}\varphi_1$, $\mathsf{F}\varphi_1$, $\mathsf{X}\varphi_1$, $\varphi_1 \mathsf{U} \varphi_2$, and $\varphi_1 \mathsf{W} \varphi_2$ for state formulae φ_1 and φ_2. The linear-time logic LTL is a fragment of CTL^* that only allows formulae of the form $\mathsf{A}\psi$, where A is the only occurrence of a path quantifier within ψ. When taking LTL as subset of CTL^*, LTL formulae are implicitly prefixed with the universal path quantifier A.

2.3. Utilizing Strongly Connected Subgraphs

Identifying a program's strongly-connected subgraphs allows us to find the set of *branching-relations* that characterize instances of branching nondeterministic decisions within a program's transition relation. These branching-relations are distinguished if there exists two nondeterministic transitions stemming from the same location and yet are not part of the same strongly-connected subgraph. We provide some notation regarding strongly-connected subgraphs followed by the definition of *branching-relations* below.

For a program P and $n \geq 1$, we denote an ordered sequence of locations $\ell_0, ..., \ell_n$ as a cycle c if $\ell_n = \ell_0$ and for every $i \geq 0$ there exists some $(\ell_i, \rho_i, \ell_{i+1}) \in E$. Let C be the set of program locations such that $\ell \in \mathcal{L}$ appears in some cycle c. That is, $C = \{\ell \mid \exists c.\ \ell \in c\}$. For a program P and the set of locations C, we identify $\mathrm{SCS}(P, C)$ as some maximal set of non-trivial strongly-connected subgraphs (SCSs) of P such that every two subgraphs $G_1, G_2 \in \mathrm{SCS}(P, C)$ are either disjoint or one is contained in the other and for every $\ell \in C$, there exists at least one $G \in \mathrm{SCS}(P, C)$ such that $\ell \in G$. The details regarding the identification of C and $\mathrm{SCS}(P, C)$ are standard and thus omitted here (see, e.g., [Cormen et al. 2001]). We denote the minimal SCS in $\mathrm{SCS}(P, C)$ that contains a location $\ell \in \mathcal{L}$ by $\textsc{MinSCS}(P, C, \ell)$. This is well defined as every two SCSs in $\mathrm{SCS}(P, C)$ are either disjoint or on is contained in the other. For example, consider the control-flow graph in Figure 1. One possible partition is $\{\ell_1, \ell_2\}$, $\{\ell_1, \ell_2, \ell_3\}$, and $\{\ell_1, \ldots, \ell_4\}$. The minimal SCS containing ℓ_1 is $\{\ell_1, \ell_2\}$ and the minimal SCS containing ℓ_3 is $\{\ell_1, \ell_2, \ell_3\}$. An alternative partition is $\{\ell_1, \ell_4\}$, $\{\ell_2, \ell_3\}$, and $\{\ell_1, \ldots, \ell_4\}$. According to this partition the minimal SCS containing ℓ_3 is $\{\ell_2, \ell_3\}$.

Branching-relations are pairs (ρ_1, ρ_2) such that for some location ℓ, (ℓ, ρ_1, ℓ') and (ℓ, ρ_2, ℓ'') are transitions of P and $\ell' \in \textsc{MinSCS}(P, C, \ell)$ and $\ell'' \notin \textsc{MinSCS}(P, C, \ell)$. That is, ρ_1 is the condition for remaining in the (minimal) SCS of ℓ and ρ_2 is the condition for leaving the (minimal) SCS of ℓ. Consider Figure 1 again, the pair (ρ_1, ρ_7) is a branching-relation over ℓ_1 when considering the first aforementioned partition. Indeed, ρ_1 stays within the $\textsc{MinSCS}(P, C, \ell_1) = \{\ell_1, \ell_2\}$ but ρ_7 leaves it. The pair (ρ_4, ρ_8) is a branching-relation over ℓ_4 when considering the second partition, but not the first. We note that one pair is sufficient

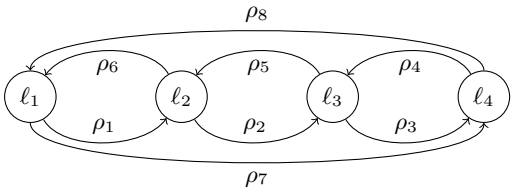


Fig. 1: A control-flow graph with multiple possible partitions of SCSs.

evidence that *some* transitions are leaving the SCS. In the case that there are multiple transitions leaving an SCS (or staying in the SCS), then multiple branching-relations can identify the same location.

3. APPROACH OVERVIEW AND EXAMPLE

3.1. Overview

In this section, we present a quick overview of our CTL* verification procedure ProveCTL*, presented in Alg. 4 and Alg. 3 with an in-depth explanation provided later in Section 4. The procedure is designed to recurse over the structure of a given CTL* formula, and for each sub-formula θ we produce a precondition a that ensures its satisfaction. That is, a is an assertion over program variables and locations characterizing the states of the program that satisfy θ. We start by finding the precondition of the innermost sub-formula, followed by searching for the preconditions of the outer sub-formulae dependent on it.

A given CTL* formula is deconstructed to differentiate between state and path sub-formulae, as the crux of verifying CTL* formulae lies within identifying the interplay between the arbitrary nesting of path and state formulae. Preconditions for branching-time logic state formulae can be acquired via existing CTL model checking techniques that return an assertion characterizing the states in which a sub-formula holds. The essence of our algorithm is thus within how we acquire sufficient preconditions for path formulae that admit a sound interaction with state formulae. The algorithm is based on the procedures below, which are defined in later sections of the paper:

Approximate is a procedure that performs a syntactic conversion from a path formula to its corresponding over-approximated universal CTL formula (ACTL)[2]. The over-approximated formula can then be checked by an existing CTL model checker over a symbolically partially-determinized form of the program to reduce path formula verification to state formula verification.

Determinize allows us to reason about path characterization through state characterization, as the satisfaction of an ACTL over-approximated formula implies the satisfaction of the path formula. However, the inverse does not hold. The procedure thus constructs a form of a partially-determinized program over the symbolic representations of all characterized instances of branching nondeterminism (i.e. *branching-relations*), stemming from the same program location ℓ. That is, nondeterministic decisions regarding which paths are taken would be determined by *prophecy variables*, which determine future outcomes of the program execution, and their values [Abadi and Lamport 1991]. Recall that branching-relations are distinguished if they are not part of the same strongly connected subgraph.

QuantElim acquires the proper set of states that satisfy a formula that has been verified over a determinized program. This allows for the path quantification present within a CTL* formula, that is, whether all paths (or some paths) starting from a state satisfy a path

[2]ACTL is the universal subset of CTL where one can only address all possible paths with the universal quantifier A (e.g. AG or AF), but not the existence of some paths with E (e.g. EG or EF).

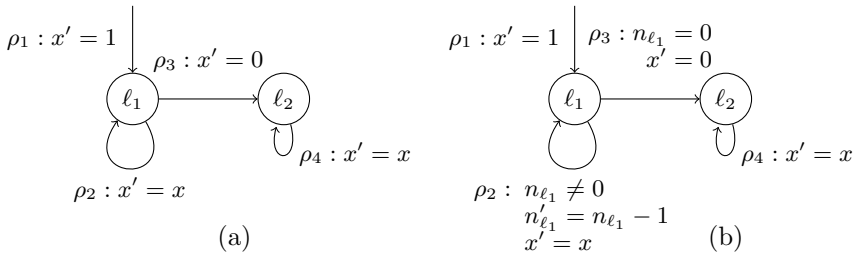


Fig. 2: (a) The control-flow graph of a program for which we wish to prove the CTL^* property $\mathsf{EFG}\ x = 1$. (b) The control-flow graph after calling DETERMINIZE, it includes the prophecy variable n_{ℓ_1} corresponding to the nondeterministic branching-relation (ρ_2, ρ_3).

formula. When a CTL^* formula of the form $\theta ::= \mathsf{A}\psi \mid \mathsf{E}\psi$ is reached after acquiring a set of states satisfying ψ, θ is verified on the same determinized program used for ψ. We then must use quantifier elimination to acquire the proper set of states that satisfy θ, thus quantifying the assertions over the values of the prophecy variables. If the formula is of the form $\mathsf{A}\psi$, we universally quantify the prophecy variables appearing in the set of states that satisfy $\mathsf{A}\psi$. If the formula is of the form $\mathsf{E}\psi$, we existentially quantify the prophecy variables.

3.2. Example

Consider the program in Fig. 2(a) and the property $\mathsf{EFG}\ x = 1$ stating that there exists a possible future where $x = 1$ will eventually become true and stay true. This is a system stabilization property, which can only be expressed in CTL^*. The property clearly holds for the program as evidenced by the path $(\ell_1, \langle x \mapsto 1\rangle), (\ell_1, \langle x \mapsto 1\rangle), \ldots$, which remains in ℓ_1 forever. In order to check this property we recursively handle its sub-formulae. We begin by identifying that $\mathsf{G}\ x = 1$ is a path formula, and thus use APPROXIMATE to return the over-approximated state formula $\mathsf{AG}\ x = 1$. We then initiate a CTL model checking task where we seek a set of states a_{G} such that $\mathsf{EF}a_{\mathsf{G}}$ holds, and for every state s such that $s \models a_{\mathsf{G}}$ we have $s \models \mathsf{AG}\ x = 1$.

Our formula would now only be valid if we can find a set of states that are eventually reached in a possible future from the program's initial states such that $\mathsf{AG}\ x = 1$ holds. However, no such set of states exists as the nondeterministic choice from ℓ_1 to ρ_2 and ρ_3 does not allow us to determine if we will eventually leave the loop or not. That is, there exists no set of states that can exemplify the infinite branching possibilities of leaving ρ_2 to possibly reaching ρ_3 or remaining in ρ_2 forever. In order to reason about the original sub-formula $\mathsf{G}\ x = 1$, we must be observing sets of paths, not states. Given that we over-approximated our formula in a way that allows us to only reason about states, we thus symbolically determinize the program to simultaneously simulate all possible related paths through the control flow graph and try to separate them to originate from distinct states in the program.

Our procedure DETERMINIZE would then return a new symbolically partially-determinized program in which a newly introduced prophecy variable, named n_{ℓ_1} in Fig. 2(b), is associated with the branching-relation (ρ_2, ρ_3), and is used to make predictions about the occurrences of relations ρ_2 and ρ_3. Recall that branching-relations are pairs of nondeterministic transitions, one remaining in a SCS and the other leaving the same SCS. In this case, ρ_3 is indeed disjoint from the strongly connected subgraph of ℓ_1.

Given that we initialize n_{ℓ_1} to a nondeterministic value, for every path in the program, a positive concrete number chosen at the nondeterministic assignment predicts the number of instances that transition ρ_2 is visited before transitioning to ρ_3. That is, we remain in

ρ_2 until $n_{\ell_1} = 0$, with n_{ℓ_1} being decremented at each passage through the loop. Once we terminate the loop, the prophecy variable is nondeterministically reset (for the case that we return to the same loop again). A negative assignment to n_{ℓ_1} denotes remaining in ρ_2 forever, or non-termination. We note that this modification does not change the set of traces of the program.

We can now utilize an existing CTL model-checker to return an assertion characterizing the states in which $\mathsf{G}\ x = 1$ holds by verifying the determinized program, denoted by P_D, using the over-approximated CTL formula $\mathsf{AG}\ x = 1$. The assertion $a_{\mathsf{G}} = (\ell_1 \wedge n_{\ell_1} < 0)$ is returned. Indeed, from states where the program is in ℓ_1 and when $n_{\ell_1} < 0$ the program remains in ℓ_1 forever. We proceed by replacing the sub-formula with its assertion in the original CTL* formula, resulting in $\mathsf{EF}a_{\mathsf{G}}$. To verify the outermost CTL* formula, EF, note that syntactically this is a readily acceptable CTL formula. However, we cannot simply use a CTL model checker as the path quantifier E exists within a larger relation context reasoning about paths given the inner formula FG. We thus must use the CTL model-checker to verify $\mathsf{EF}a_{\mathsf{G}}$ over the same determinized program previously generated.

Our procedure returns with the same precondition $(\ell_1 \wedge n_{\ell_1} < 0)$. Indeed, the set of states that eventually reach ℓ_1 with $n_{\ell_1} < 0$ are those that start in ℓ_1 with $n_{\ell_1} < 0$. We then use quantifier elimination to existentially quantify out all introduced prophecy variables. The existential quantification corresponds to searching for some path (or paths) that satisfy the path formula. Thus, if there is a state s in the original program, and some value of the prophecy variables v such that *all* paths from the combined state $(s, n_{\ell_1} = v)$ in P_D satisfy the path formula then clearly, these paths give us a sufficient proof to conclude that $\mathsf{EFG}\ x = 1$ holds from s in P. In our case, this indeed happens and the program, as mentioned, satisfies the formula.

4. CHECKING CTL* FORMULAE

In this section, we describe the details of our CTL* model checking procedure ProveCTL*. We first define the procedures utilized by ProveCTL*, namely Determinize and Approximate, followed by our model checking procedure and its utilization of QuantElim.

4.1. Determinization

The procedure Determinize constructs a form of symbolically partially-determinized program by considering branching-relations that characterize instances of branching nondeterminism. Note that a partial determinization denotes that a program will still include non-determinism following the transformation. We present our procedure in Alg. 1, where a program P is given and a partially-determinized program P_D, contingent upon nondeterministic branching-relations, is returned. Ultimately, Determinize is designed to allow proof tools for branching-time logic state formulae to be used to reason about path formulae.

We begin by finding a sufficient set of branching-relations to symbolically determinize the program to one which has the same set of paths as the original. These relations are distinguished if there exist at least two nondeterministic relations stemming from the same location and yet are not part of the same strongly-connected subgraph. Our procedure thus begins by iterating over the set of a program's edges, $(\ell, \rho, \ell') \in E$ on line 6. We identify whether or not $\ell \in C$ given that $G = \text{MinSCS}(P, C, \ell)$ and $G \neq \emptyset$ on lines 7 and 8. If from some location ℓ, where $G = \text{MinSCS}(P, C, \ell)$, there is an edge to ℓ' such that $\text{MinSCS}(P, C, \ell')$ is not equivalent to G, we can conclude that the transition from ℓ to ℓ' leaves the SCS of ℓ. We only desire that ℓ and ℓ' be elements of the most minimal SCS as such an edge eludes to the nondeterministic decision point where a transition diverted from remaining within an SCS. This nondeterministic point is key to the identification of where determinization must occur to facilitate the application of state-based reasoning to path-based reasoning for a given program P.

ALGORITHM 1: DETERMINIZE identifies branching-relations and constructs a symbolically determinized program over them.

```
1  Let DETERMINIZE(P) : program =
2      P_D = P
3      SYNTH = [ ]
4      (L_D, E_D, Vars_D) = P_D
5      C = CYCLEPOINTS(P)
6      foreach (ℓ, ρ, ℓ') ∈ E_D do
7          G = MINSCS(P, C, ℓ) ∈ SCS(P, C)
8          if G ≠ ∅ ∧ MINSCS(P, C, ℓ') ≠ G then
9              SYNTH = ℓ :: SYNTH
10     foreach (ℓ, ρ, ℓ') ∈ E_D do
11         if ℓ ∈ SYNTH then
12             Vars_D = Vars_D ∪ n_ℓ ∈ ℤ
13             if ℓ' ∈ MINSCS(P, C, ℓ) then
14                 ρ = ρ ∧ (n_ℓ ≠ 0) ∧ (n'_ℓ = n_ℓ − 1)
15             else
16                 ρ = ρ ∧ (n_ℓ = 0)
17     return P_D
```

If the strongly connected subgraphs of ℓ and ℓ' do differ, we add ℓ to SYNTH, a list that tracks locations with nondeterministic branching points. For every such location, we identify branching-relations corresponding to the decision of either remaining in the same SCS, or leaving it. After finding all possible elements of SYNTH, on line 11 we iterate over the program edges, and for the branching-relations encountered we introduce a new prophecy variable to predict the future outcome of the decision. Recall that there may exist multiple transitions leaving (or staying in) a strongly connected subgraph, as multiple branching-relations can identify the same location. In such a case, only one prophecy variable is produced for each location, and is utilized across these transitions. Indeed, our motivation is to identify nondeterministic points so we can symbolically simulate all possible branching paths through a program, yet decisions regarding which paths are taken are determined by prophecy variables and their values. Information regarding different paths is now stored in the state of the modified program. This allows for a correspondence such that the verification path formulae can be reduced to the verification of ACTL formulae.

When an edge $(\ell, \rho, \ell') \in E$ is reached containing $\ell \in$ SYNTH, a prophecy variable $n_\ell \in \mathbb{Z}$ is added to the set of program variables Vars at line 13. If ℓ' is contained within MINSCS(P, C, ℓ), we constrain ρ by requiring that $n_\ell > 0$, and then decrement n_ℓ. If ℓ' is not contained within MINSCS(P, C, ℓ), we constrain ρ by $n_\ell = 0$, and n'_ℓ remains unconstrained, entailing a reset to a nondeterministic integer value. The nondeterministic decision of the number of times a cycle is passed through is thus now determined by the prophecy variable n_ℓ. In the case that $n_\ell < 0$, this rule corresponds to behaviors where every visit to ℓ is followed by a successor in the same SCS (i.e., the computation always remains in the SCS of ℓ). The nondeterminism within a transition relation is thus either determined at initialization by the initial choice of values for n_ℓ or else later in a path by choosing new nondeterministic values for n_ℓ.

We show that the determinization maintains the set of paths in the original program and the prophecy variables introduced merely trade nondeterminism in the transition relation for a larger, nondeterministic state space.

THEOREM 4.1. *For every path π in P there is a path π' in P_D such that $\pi' \Downarrow_{\mathsf{Vars}} = \pi$. Furthermore, for every path π' in P_D it holds that $\pi' \Downarrow_{\mathsf{Vars}}$ is a path in P.*

PROOF.

— Consider a path π in P where $\pi = (\ell_0, f_0), (\ell_1, f_1), \ldots$. Consider a location ℓ_j, an SCS G_j such that $G_j = \text{MINSCS}(P, C, \ell_j)$, and the variable n_{l_j}. We can annotate each pair (ℓ_i, f_i) in π by the number of expected future visits to G_j. We call a transition (ℓ, ρ, ℓ') a *reset transition* for n_{l_j} if $\ell \in G_j$ and $\ell' \notin G_j$ or if $\ell = \ell_I$. Notice that in P_D, a reset transition $(\ell, \rho\, \ell')$ is conjuncted to $n_{l_j} = 0$. This leaves the value of n'_{l_j} unconstrained, assigning it an arbitrary value once such a transition is taken. We call a transition (ℓ, ρ, ℓ') an *internal transition* for n_{l_j} if $\ell \in G_j$, $\ell' \in G_j$ and there is some $\ell'' \notin G_j$ and a transition (ℓ, ρ', ℓ''). Notice that in P_D the transition (ℓ, ρ, ℓ') is conjuncted to $n'_{l_j} = n_{l_j} - 1$. Also, in P_D every transition that is neither reset nor internal for n_{l_j} is conjuncted (implicitly) to $n'_{l_j} = n_{l_j}$. It follows that for every $i \geq 0$ the number of internal transitions for n_{l_j} that appear until a reset transition is well-defined (and may be infinity). Clearly, this annotation also matches the transition in P_D. It follows that by adding an appropriate annotation for every n_l that is added to P_D, we get a path in P_D whose projection on Vars is exactly that of path π.

— Consider an infinite path π' in P_D. Now consider a pair of states $((\ell, (f, v)), (\ell', (f', v'))$ appearing in π', where v and v' are the assignments to the prophecy variables appearing in P_D. By definition, there is a transition (ℓ, ρ', ℓ') in P_D such that $((\ell, (f, v)), (\ell', (f', v')) \models \rho'$. However, $\rho' = \rho \wedge \xi$, where ρ is an assertion over Vars and ξ is the assertion over the prophecy variables. It then must be the case that $(f, f') \models \rho$. It follows that $\pi = \pi' \Downarrow_{\mathsf{Vars}}$ is a path in P.

□

4.2. Approximation

In Alg. 2, we present a syntactic conversion from pure linear-time formulae in CTL^*, that is LTL, to a corresponding over-approximation in ACTL. Our procedure is given a path formula ψ and two atomic preconditions, $a_{\theta'_1}$ and $a_{\theta'_2}$, corresponding to the satisfaction of the nested CTL^* formulae which appear within ψ. Recall that a precondition is an assertion over program variables and locations, characterizing the states of a program that satisfy a certain temporal formula. The precondition $a_{\theta'_2}$ is a conditional parameter utilized only when LTL formulae requiring two properties (e.g. W, U, $\wedge$, $\vee$) are given. Due to the recursive nature of PROVECTL*, presented in the next section, these preconditions would have already been priorly generated.

On lines 3 – 7, we instrument a universal path quantifier A preceding the appropriate temporal operators. Not only so, but the sub-formulae θ'_1 and θ'_2 are replaced with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively. This aligns with how PROVECTL* will recursively iterate over each inner sub-formula followed by search for the preconditions of the outer sub-formulae dependent on it. Replacing a path formula by its CTL approximation indeed is sound in the sense that if the modified formula holds then the original holds as well. Note that in the context of a deterministic program, approximation is both sound and *complete*. That is, both path formula and corresponding state formula have the same truth value. This follows from every state having at most one possible future.

In the following Theorem, for notational convenience, we assume that every path operator has an arity of two and refer to its operands. In case the second operand (or both) do not exist then they are not important and can be ignored.

ALGORITHM 2: APPROXIMATE produces a syntactic conversion from a path formula to its corresponding over-approximation in ACTL.

```
Let APPROXIMATE(ψ, a_{θ'_1}, a_{θ'_2}) : φ =
    match (ψ) with
        Fθ'_1 → return AF a_{θ'_1}
        Gθ'_1 → return AG a_{θ'_1}
        Xθ'_1 → return AX a_{θ'_1}
        θ'_1 W θ'_2 → return A a_{θ'_1} W a_{θ'_2}
        θ'_1 U θ'_2 → return A a_{θ'_1} U a_{θ'_2}
        θ'_1 ∧ θ'_2 → return a_{θ'_1} ∧ a_{θ'_2}
        θ'_1 ∨ θ'_2 → return a_{θ'_1} ∨ a_{θ'_2}
```

THEOREM 4.2. *Consider a program P and a path formula ψ, where θ_1 and θ_2 are the direct sub-formulae of ψ. Let a_{θ_i} be an approximation of θ_i such that for every state s we have $P, s \models a_{\theta_i}$ implies $P, s \models A\theta_i$. Then, for every state s, we have $P, s \models$ APPROXIMATE(ψ) then $P, s \models A\psi$.*

PROOF. For propositions and Boolean combinations of simpler formulae, the proof is immediate.

— Suppose that $\psi = \mathsf{G}\theta_1$. Then, APPROXIMATE$(\psi, a_{\theta_1}, a_{\theta_2})$ is $\mathsf{AG}(a_{\theta_1})$. Suppose that $s \models$ APPROXIMATE$(\psi, a_{\theta_1}, a_{\theta_2})$ but $s \not\models \mathsf{A}\psi$. Then, there is a path π starting in s such that π does not satisfy $\mathsf{G}\theta_1$. It follows that there is a suffix π' of π that does not satisfy θ_1. Let s' be the first state in π'. However, by assumption, $s' \models a_{\theta_1}$. This contradicts the assumption about a_{θ_1}.
— Suppose that $\psi = \mathsf{F}\theta_1$. Then, APPROXIMATE$(\psi, a_{\theta_1}, a_{\theta_2})$ is $\mathsf{AF}(a_{\theta_1})$. Suppose that $s \models$ APPROXIMATE$(\psi, a_{\theta_1}, a_{\theta_2})$ but $s \not\models \mathsf{A}\psi$. Then, there is a path π starting in *state* such that π does not satisfy $\mathsf{F}\theta_1$. However, by $s \models$ APPROXIMATE$(\psi, a_{\theta_1}, a_{\theta_2})$, there is a suffix π' of π such that the first state s' in π' satisfies a_{θ_1}. It follows that π' satisfies θ_1 and that π satisfies $\mathsf{AF}\theta_1$.
— The proofs for until and weak until are similar but take further corner cases into account.

□

Theorem 4.2 does not consider existential path quantification. In order to conclude that the CTL* formula $P, s \models \mathsf{E}\psi$ for some path formula ψ, we require that there is some value v of the prophecy variables such that $P_D, (s, v) \models \mathsf{A}\psi$. This means that when restricting attention to a certain set of paths that start in a state s (those that match the valuation v for prophecy variables), *all* paths in the set satisfy the formula ψ. Clearly, this satisfies the requirement that there is some path that satisfies the formula.

4.3. CTL* Verification Procedure

In this section, we present our main CTL* verification procedure, PROVECTL*. Alg. 3 depicts VERIFY, which wraps the main procedure PROVECTL*, shown in Alg. 4. We generate a determinized copy of the program, P_D, using the aforementioned procedure DETERMINIZE. This program is then passed into PROVECTL* along with the original program P and a CTL* property θ. PROVECTL* then returns an assertion a, characterizing the states in which θ holds. The second argument returned is disregarded, indicated by "_", as it is only used within the recursive calls of PROVECTL*. When PROVECTL* returns to VERIFY,

ALGORITHM 3: VERIFY wraps PROVECTL* and then checks all initial states.

```
1 Let VERIFY(θ, P) : bool =
2 |  (L, E, Vars) = P
3 |  P_D = DETERMINIZE(P)
4 |  (a, _) = PROVECTL*(θ, P, P_D)
5 |  return ∀(ℓ_I, ρ, ℓ) ∈ E ∀f : Vars → Vals . (f_{-1}, f) ⊨ ρ implies (ℓ, f) ⊨ a
```

it is only necessary to check if the precondition a is satisfied by the initial states of the program.

We now turn to the main procedure PROVECTL* in Alg. 4. In order to synthesize a precondition for a CTL* property θ, a given CTL* formula is first deconstructed to differentiate between state and path sub-formulae, as the crux of verifying CTL* formulae lies within identifying the interplay between the arbitrary nesting of path and state formulae. On line 3, if θ can be identified as a state formula φ, we carry out the set of actions on lines 4 – 20. If θ is identified as a path formula ψ, we then carry out the set of actions on lines 21 – 31. Note that in our algorithm, we denote any temporal operator (i.e., $\mathsf{F}, \mathsf{G}, \mathsf{X}, \mathsf{W}$ and U) by $\circ$. For both the state and path formulae cases, we recursively accumulate the preconditions generated when considering the sub-formulae of θ at lines 7, 8, 10, 13, 24, 25, and 27. That is, for each sub-formula θ, we produce a precondition a_θ that ensures its satisfaction. We note that the precondition of an atomic proposition α is the proposition itself, as shown on line 13. The precondition is then utilized in the remaining actions of the algorithm.

4.3.1. Verifying Path Formulae. When a path formula ψ is reached, we begin by over-approximating the path formula by syntactically converting it to the universal subset of branching-time logic (ACTL) using the procedure APPROXIMATE. Recall that the preconditions generated when considering the sub-formula(e) of ψ at lines 24, 25, and 27 will be utilized by APPROXIMATE to replace θ'_1 and θ'_2 with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively. On line 29, APPROXIMATE would then return a corresponding state formula ψ' where a universal path quantifier precedes the temporal operator within ψ.

A precondition for the newly attained ACTL formula ψ' can now be acquired via existing CTL model checkers which return an assertion characterizing the states in which ψ' holds. Existing tools which support this functionality include [Beyene et al. 2013] and [Cook et al. 2014]. In our tool prototype, we build upon the latter. Recall that a precondition for a path formula requires more than a precondition for the corresponding state formula, as ψ' is merely an over-approximation. We thus must utilize the provided determinized program P_D when employing a CTL model checker rather than the original program P, as shown on line 30. The assertion a_θ is then returned characterizing the sets of states in which θ holds.

Recall that P_D leads to better correspondence between ψ and ψ'. That is, we find a sufficient set of branching-relations, which determinize the program to one which has the same set of paths as the original, yet decisions regarding which paths are taken are determined by introduced prophecy variables and their values, allowing us to reduce path-based reasoning to state-based reasoning. The assertion a_θ that is returned thus may be defined over the introduced prophecy variables in P_D.

Finally, on line 31, we set the boolean flag PATH to true. This flag is the second argument to be returned by PROVECTL*. It indicates to the caller that the result a_θ returned by the recursive call is approximated. The value of PATH is used for deciding whether to use a_θ as is or modify it (in the case that the verified sub-formula is a state or a path formula, respectively), admitting a sound interaction between state and path formulae. Below, we further demonstrate this point.

ALGORITHM 4: Our recursive CTL* verification procedure employs an existing CTL model checker and uses our procedures Approximate and QuantElim. It expects a CTL* property θ, a program P, and its determinized version P_D as parameters. An assertion characterizing the states in which θ holds is returned along with a boolean value indicating whether the formula checked was a path formula (and hence approximated).

```
1  Let rec ProveCTL* (θ, P, P_D) : (formula, bool) =
2      (L, E, Vars) = P
3      match (θ) with
4          φ : stateformula →
5              match (φ) with
6                  θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 | Aθ'_1 ∘ θ'_2 | Eθ'_1 ∘ θ'_2 →
7                      (a_θ'1, Path_1) = ProveCTL*(θ'_1, P, P_D)
8                      (a_θ'2, Path_2) = ProveCTL*(θ'_2, P, P_D)
9                  A∘θ' | E∘θ' →
10                     (a_θ'1, Path_1) = ProveCTL*(θ', P, P_D)
11                     (a_θ'2, Path_2) = (False, False)
12             match (φ) with
13                 α → a_θ = α;
14                 _ →
15                     φ' = Replace(φ, a_θ'1, a_θ'2)
16                     if Path_1 ∨ Path_2 then
17                         a_θ = QuantElim(CTL(P_D, φ'), φ)
18                     else
19                         a_θ = CTL(P, φ')
20             Path = False
21         ψ : pathformula →
22             match (ψ) with
23                 θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 | θ'_1 ∘ θ'_2 →
24                     (a_θ'1, −) = ProveCTL*(θ'_1, P, P_D)
25                     (a_θ'2, −) = ProveCTL*(θ'_2, P, P_D)
26                 ∘θ' →
27                     (a_θ'1, −) = ProveCTL*(θ', P, P_D)
28                     a_θ'2 = False
29             ψ' = Approximate(ψ, a_θ'1, a_θ'2)
30             a_θ = CTL(P_D, ψ')
31             Path = True
32     return (a_θ, Path)
```

4.3.2. Verifying State Formulae. In the case that a state formula φ is reached, we partition the state sub-formulae by the syntax of CTL as shown on lines 6 and 9. Recall that temporal operators are denoted by $\circ$. This allows us to not only utilize existing CTL model checkers, but to also eliminate the redundant verification of a temporal operator, when it is already preceded by a path quantifier. As a side effect of partitioning φ in such a way, a path formula ψ will always be in the form of a pure linear-time path formula, that is, LTL. This particular deconstruction of a CTL* formula is what allows us to identify the intricate interplay between path and state formulae.

We begin by recursively generating preconditions when considering the sub-formula(e) of φ at lines 7, 8, and 10. Recall that the precondition of an atomic proposition α is the proposition itself, we thus return the atomic sub-formula on line 13, where no further work is necessary. Otherwise, the recursively acquired preconditions will then be utilized by the procedure REPLACE on line 15. REPLACE substitutes θ'_1 and θ'_2 with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively, and returns a new state formula φ'. Preconditions for branching-time logic state formulae can be acquired via existing CTL model checkers. However, in order to allow for the path quantification present within a CTL* formula to range over path formulae, we must consider whether all or some paths starting from a particular state satisfy a path formula. This is required in the case that the immediate inner sub-formula is a pure linear-time path formula, which is identified by the aforementioned boolean flag PATH given the partitioning of θ. The role of PATH is to track if a sub-formula of the current formula is a path formula. That is, PATH indicates that the path quantifier exists within the context of verifying a path formula, and not a branching-time state formula. Thus, it must be verified using P_D, yet the set of states of P_D that characterize it actually represents a set of paths. This set of paths must be collapsed later to a characterization of the set of states of P where the (state) formula holds. This is the key to allowing the interplay between state and path formulae, as we further demonstrate below.

ALGORITHM 5: QUANTELIM applies quantifier elimination in order to convert path characterization to state characterization restricting attention to states from which an infinite path exists.

```
1 Let QuantElim(a, φ) : AP =
2 |  a_EG = CTL(P_D, EG True)
3 |  match (φ) with
4 |  |  Eψ → return QE(∃n_{ℓ∈L}.a_EG ∧ a)
5 |  |  − → return QE(∀n_{ℓ∈L}.a_EG → a)
```

4.3.3. Quantifier Elimination for Satisfying Preconditions. The procedure QUANTELIM, presented in Alg. 5, which converts path characterization to state characterization, is thus executed at line 17 of PROVECTL*. QUANTELIM takes in the assertion a returned from calling a CTL model checker on the determinized program P_D and the partitioned CTL formula φ', as well as the original formula φ. We then quantify the assertions over the values of the prophecy variables. If φ is a universal CTL formula, we universally quantify the prophecy variables appearing in the set of states that satisfy φ on line 5 in Alg. 5. If φ is an existential CTL formula, we existentially quantify the prophecy variables on line 4. Predictions of the prophecy variables may lead to finite paths to appear in the program, thus quantification must be restricted to states for which there does exist a prophecy value leading to infinite paths. For example, consider Fig. 2(b), and a path in which the loop has not yet terminated, yet the prophecy variable n_{ℓ_1} can no longer be decreased given that it has reached a value of 0. We thus cannot take another loop transition given that we can no longer decrease n_{ℓ_1}, nor can we leave the loop given that it has not terminated. Hence, on line 2 we acquire the precondition a_{EG} satisfying the CTL formula entailing nontermination, that is EG TRUE for P_D. The precondition a_{EG} is then conjuncted with a to ensure that the quantification of prophecy variables does not include finite paths generated due to invalid predictions of the prophecy variables. This is done according to the polarity of the quantification (universal or existential). The assertion a_θ is then returned by QUANTELIM characterizing the set of states in which θ holds.

In the case that PATH is false, the most immediate inner sub-formula would then be a state formula. This indicates that we can indeed use a CTL model checker using φ' and

the original program P, as demonstrated on line 19. Upon the return of ProveCTL* to its caller Verify, a_θ will contain the precondition for the most outer temporal property of the original CTL* formula θ. Now it is only necessary to check if the precondition a_θ is satisfied by the initial states of the program to complete the verification of our CTL* formula. Finally, Path is set to false, in order to carry out the above procedure again when necessary.

Theorem 4.3. *If* Verify(θ, P) *returns true then* $P \models \theta$.

Proof. We show by induction on the number of path quantifiers in the CTL* formula θ that the set of states computed as satisfying θ in line 17 of ProveCTL* is sound. That is, if a state (ℓ, f) is such that $(\ell, f) \models a_\theta$ then (ℓ, f) satisfies θ.

— Consider a state formula $\mathsf{A}\psi$, where ψ does not include further path quantifications. The computation of a_θ uses recursive calls to Approximate() with preconditions for the subformulae of ψ. By induction on the structure of ψ and repeated use of Theorem 4.2 we can show that every precondition $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to Approximate() is sound. That is, if $P, s \models a_{\theta'_i}$ then $P, s \models \mathsf{A}\theta'_i$. It follows that once we get to the check of the state formula $\mathsf{A}\psi$ the precondition a_θ is obtained from universal quantification of a sound approximation of $\mathsf{A}\psi$. It follows that in P_D for every possible valuation v for the prophecy variables either $(\ell, (f, v))$ has no infinite paths starting from it or $(\ell, (f, v))$ satisfies $\mathsf{A}\psi$ in P_D.
Consider a path π that starts in (ℓ, f) in P. We note that if (ℓ, f) is reachable from some initial state, i.e., there is a computation $\sigma \cdot \pi$ for which π is a suffix, then by Theorem 4.1 there exists a computation $\sigma' \cdot \pi'$ of P_D such that $\sigma' \cdots \pi' \Downarrow_{\mathsf{Vars}} = \sigma \cdot \pi$. In particular, π' satisfies ψ as required and some state $(\ell, (f, v))$ for some assignment to prophecy variables v is reachable in P_D.

— Consider a state formula $\mathsf{E}\psi$, where ψ does not include further path quantifications. The computation of a_θ uses recursive calls to Approximate() with preconditions for the subformulae of ψ. By induction on the structure of ψ and repeated use of Theorem 4.2 we can show that every precondition $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to Approximate() is sound. That is, if $P, s \models a_{\theta'_1}$ then $P, s \models \mathsf{A}\theta'_i$. It follows that once we get to the check of the state formula $\mathsf{E}\psi$ the precondition a_θ is obtained from existential quantification of a sound approximation of $\mathsf{A}\psi$. It follows that in P_D for some possible valuation v for the prophecy variables we have that $(\ell, (f, v))$ has an infinite path starting from it and $(\ell, (f, v))$ satisfies $\mathsf{A}\psi$ in P_D.
Similar to the case of $\mathsf{A}\psi$, if an infinite path π' of P_D that starts in (ℓ, f) is the suffix of a computation of P_D then $(\ell, (f, v))$ is reachable in P_D.

— In the case of a state formula θ that includes nesting of path quantifiers the proof proceeds as before. This part relies on the structure of θ being in negation normal form and the soundness of previous approximations $a_{\theta'}$ for every state sub-formula θ' of θ.

□

We note that the implication in Theorem 4.3 is only in one direction. That is, failing to prove that a property holds does not implicate that its negation holds (though this might be proved by negating the formula, converting it to negation normal form, and running our procedure on it). This incompleteness stems from the over-approximation of path formulae by a corresponding ACTL formulae, as although this over-approximation is checked over P_D, P_D does not determinize all paths. In the next section we discuss the incompleteness of this determinization scheme.

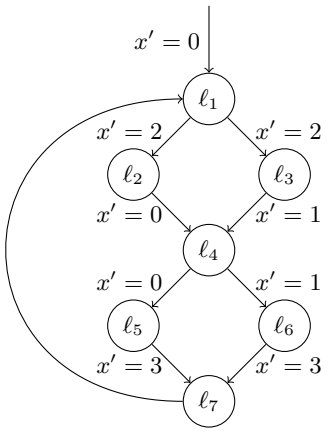


Fig. 3: Program for which determinization is insufficient.

5. (IN)COMPLETENESS OF DETERMINIZATION

Given that our determinization technique has been adopted and repurposed from a similar symbolic determinization technique introduced in [Cook and Koskinen 2011] for the verification of LTL formulae, we have thus inherited the limitations found within their technique. In this section we discuss the aforementioned limitations.

We begin with a contrived illustrative example in Fig. 3 that serves as a theoretical exercise on the completeness of determinization. First, we characterize all properties that represent the different paths which can be taken inside the loop:

$$\begin{aligned}
\varphi_1 &:= x{=}2 \rightarrow \mathsf{X}(x{=}0 \,\mathsf{U}\, x{=}3) \\
\varphi_2 &:= x{=}2 \rightarrow \mathsf{X}(x{=}0 \,\mathsf{U}\, (x{=}1 \,\mathsf{U}\, x{=}3)) \\
\varphi_3 &:= x{=}2 \rightarrow \mathsf{X}(x{=}1 \,\mathsf{U}\, (x{=}0 \,\mathsf{U}\, x{=}3)) \\
\varphi_4 &:= x{=}2 \rightarrow \mathsf{X}(x{=}1 \,\mathsf{U}\, x{=}3)
\end{aligned}$$

Now consider the property $\psi := \mathsf{EG}(\bigvee_{i=1}^{4} \varphi_1)$. This property holds given that there always exists a path in which a computation satisfies one of the four φ_i properties. That is, each property φ_i is representative of a possible passage through the loop. Unfortunately, our procedure would not be able to determine that the given program satisfies this property. Recall that our procedure will determinize the program by replacing nondeterministic decisions regarding which paths are taken using prophecy variables to determine future outcomes of the program execution. We then would attempt to verify the over-approximated ACTL variant of the properties introduced in φ_i. For example, φ_1's ACTL approximation would be $x{=}2 \rightarrow \mathsf{AX}(\mathsf{A}(x{=}0 \,\mathsf{U}\, x{=}3))$. In particular, the sub-property $\mathsf{A}(x{=}0 \,\mathsf{U}\, x{=}3)$ holds only at ℓ_5 and ℓ_7. It follows that there exists no set of states that satisfy $\mathsf{AX}(\mathsf{A}(x{=}0 \,\mathsf{U}\, x{=}3))$. Thus the set of states satisfying the property is characterized by the precondition False; and the ACTL approximation of φ_1 does not hold given that we would be applying QuantElim over the precondition False. Similarly, the remaining ACTL approximations of other subformulae do not hold. Our procedure will thus fail to verify that the property ψ is true for this program. The problem lies within the need to specify in advance one of uncountably many choices.

Consider again the program in Fig. 3. Let $\psi_0 := x{=}2 \wedge \mathsf{X}\, x = 0$ and $\psi_1 := x{=}2 \wedge \mathsf{X}\, x = 1$. That is, ψ_0 describes the choice $\ell_1 \mapsto \ell_2 \mapsto \ell_4$ and ψ_1 describes the choice $\ell_1 \mapsto \ell_3 \mapsto \ell_4$. We can construct LTL formulae that search for a path that toggles between the choice of ψ_0 and ψ_1. For example, $\mathsf{E}(x{\neq}2 \,\mathsf{U}\, (\psi_0 \wedge \mathsf{X}(x{\neq}\, 2 \,\mathsf{U}\, \psi_1)))$, requires to identify the paths that

first choose to go from ℓ_1 to ℓ_2 in the first run through the loop and go from ℓ_1 to ℓ_3 in the second run through the loop. Now consider a word $w \in \{\ell_2, \ell_3\}^*$, we can write the CTL^* formula $\mathsf{E}\varphi_w$ with the aforementioned pattern that corresponds to the existence of the path that takes the choices as written in w. It follows that in order to use our determinization strategy, we would have to include a choice for *all* possible future choices of whether to branch from ℓ_1 to ℓ_2 or ℓ_3.

A possible solution would be to strengthen our determinization strategy to include a larger number of choices encoded in one variable. For example, we could consider an arbitrary integer n_b (for *branch*) and whenever the value of n_b is even, we choose the first branch and whenever the value of n_b is odd we choose the second branch. Thus whenever n_b is used, it must be divided by two (integer division), and when n_b becomes 0 it is reset to a new arbitrary value. Thus, n_b would encode an arbitrarily large number of choices on how to branch in a certain point. Given that the branch appears in a loop that could be repeated forever, this suggested improvement still does not completely determinize the program. Indeed, the computations that remain in the loop include new branching points whenever the value of n_b is reset. From the branching point at ℓ_4, it is possible to create a formula that will search for a path that creates the pattern w^ω for a word $w \in \{\ell_2, \ell_3\}^*$. Thus, predictions of arbitrarily many choices is not sufficient, as we would need to consider the predictions in $\{\ell_2, \ell_3\}^\omega$. Unfortunately, there are uncountably many different words in $\{\ell_2, \ell_3\}^\omega$. Thus, in order to fully determinize a program we would have to allow nondeterministic variables ranging over the Reals (with infinite precision) and use a trick similar to the even/odd choice with division by 2. Thus our determinization approach is limited and, in general, it is impossible to completely determinize a program.

This is clearly a theoretical exercise in completeness of determinization, and we stress that, in practice, we have found that our determinization procedure handles programs and properties that we wish to verify quite well. The automata theoretic approach to LTL model checking [Vardi and Wolper 1986] can be viewed as determinization that is tailored for the formula to be verified. We are not aware of implementations that use the automata theoretic approach for handling LTL sub-formulae within CTL^* formulae for infinite-state programs. However, in the future we wish to eliminate the limitations of our determinization procedure, given that countable non-determinism in the context of nested nondeterministic branching leads to incompleteness. A technique introduced in [Cook et al. 2015] allows for some interaction between linear-time and branching-time over fairness assumptions pertaining to a system's environment. We suspect that building upon this technique may make way for a more complete procedure supporting CTL^* verification when reasoning about path formulae. We hope to further examine both the viability and practicality of such an extension.

6. CTL^*_{lp}– ADDING PAST TO CTL^*

In this section, we consider an extension to CTL^* that admits temporal operators that refer to the past. As perviously mentioned, we specifically consider a fragment of the linear-past logic CTL^*_{lp}. We thus redefine our semantics to incorporate past-connectives below. In addition we extend our recursive CTL^* verification procedure to support the incorporation of the past. We include an example that demonstrates the different stages of the algorithm in Section 7. We additionally discuss the challenges of extending to full CTL^*_{lp} in Section 6.3, and further exhibit the usefulness of our CTL^*_{lp} extension via a case study provided in Section 8.1.

6.1. CTL^*_{lp} Syntax and Semantics

Below, we define the fragment of CTL^*_{lp} that we support in the paper. To avoid introducing further names, we henceforth use CTL^*_{lp} to refer to this fragment. When necessary we stress

that we are referring to *full* CTL^*_{lp}. As with CTL^*, the syntax of CTL^*_{lp} includes state formulae φ and path formulae ψ. Here, path formulae are partitioned to pure-past formulae τ, and general path formulae ψ. The inclusion of the past modifies the semantics of formulae to distinguish between distinct occurrences of the same state with differing histories. Hence, the models of state formulae become histories of computations that end in a certain state, and not just a state itself. We thus define histories as a non-empty finite sequence of states $s_0, s_1, \ldots, s_n$ such that for every $i < n$, we have $(s_i, s_{i+1}) \in R$ and s_0 is initial. For a history σ we denote by $|\sigma|$ the length of σ, i.e., the number of states in σ. Given that histories are prefixes of computations, for a path $\pi = (s_0, s_1, \ldots)$, we let $\pi_{|i}$ denote the ith prefix of π, that is, the history $s_0 \ldots s_i$ for $i \geq 0$. Similarly, for $\sigma = s_0, \ldots, s_n$ we denote by $\sigma_{|i}$ the ith prefix of σ for $i \leq n$. We use σ to denote histories and write $\Pi(\sigma)$ to denote the set of all computations starting with σ. A history $\sigma = s_0 \ldots s_n$ thus represents a current state, s_n, of a computation still in progress, with the additional information that the past has been $\sigma_{|n-1}$. State formulae (φ), past formulae (τ), and path formulae (ψ) are co-defined as follows:

$$\begin{array}{lcl} \varphi & ::= & \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid \mathsf{E}\psi \\ \tau & ::= & \varphi \mid \tau \wedge \tau \mid \tau \vee \tau \mid \mathsf{G}^{-1}\tau \mid \mathsf{F}^{-1}\tau \mid \mathsf{X}^{-1}\tau \mid [\tau \mathsf{W}^{-1} \tau] \mid [\tau \mathsf{U}^{-1} \tau] \\ \psi & ::= & \tau \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{G}\psi \mid \mathsf{F}\psi \mid \mathsf{X}\psi \mid [\psi \mathsf{W} \psi] \mid [\psi \mathsf{U} \psi] \end{array}$$

When discussing temporal operators, we denote future connectives by $\circ$ and past connectives by $\circ^{-1}$. When addressing both future and past connectives we utilize $\circ^{\pm}$. Note that other literature may use the notation Y (yesterday) for X^{-1}, P (past) for F^{-1}, H (historically) for G^{-1}, S (since) for U^{-1}, and B (before) for W^{-1}.

For a program P and a CTL^*_{lp} state formula φ, we say that φ holds at a history $\sigma = s_0 \ldots s_n$ in P, denoted by $P, \sigma \models \varphi$ if:
— If $\varphi = \alpha$, then $P, \sigma \models \alpha$ iff $s_n \models \alpha$
— If $\varphi = \neg\alpha$, then $P, \sigma \models \neg\alpha$ iff $s_n \not\models \alpha$
— If $\varphi = \varphi_1 \vee \varphi_2$, then $P, \sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$
— If $\varphi = \varphi_1 \wedge \varphi_2$, then $P, \sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
— If $\varphi = \mathsf{A}\psi$, then $P, \sigma \models \mathsf{A}\psi$ iff $\forall \pi \in \Pi(\sigma).\ P, \pi, |\sigma| - 1 \models \psi$
— If $\varphi = \mathsf{E}\psi$, then $P, \sigma \models \mathsf{E}\psi$ iff $\exists \pi \in \Pi(\sigma).\ P, \pi, |\sigma| - 1 \models \psi$

Path formulae (τ and ψ) are interpreted over computations. Assume the inclusion of future connectives as specified in Section 2.2. For a program P and a CTL^* path formula ψ, we say that ψ holds on a computation π in P for location i, denoted by $P, \pi, i \models \psi$ if:
— If $\psi = \mathsf{X}^{-1}\psi_1$, then $P, \pi, i \models \mathsf{X}^{-1}\psi_1$ iff $i > 0$ and $P, \pi, i - 1 \models \psi_1$
— If $\psi = \mathsf{F}^{-1}\psi_1$, then $P, \pi, i \models \mathsf{F}^{-1}\psi_1$ iff $\exists j \leq i.\ P, \pi, j \models \psi_1$
— If $\psi = \mathsf{G}^{-1}\psi_1$, then $P, \pi, i \models \mathsf{G}^{-1}\psi_1$ iff $\forall j \leq i.\ P, \pi, j \models \psi_1$
— If $\psi = \psi_1 \mathsf{W}^{-1} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{W}^{-1} \psi_2$ iff either $\exists k \leq i.\ P, \pi, k \models \psi_2$ and $\forall k < j \leq i.\ P, \pi, j \models \psi_1$ or $\forall j \leq i.\ P, \pi, j \models \psi_1$
— If $\psi = \psi_1 \mathsf{U}^{-1} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{U}^{-1} \psi_2$ iff $\exists k \leq i.\ P, \pi, k \models \psi_2$ and $\forall k < j \leq i.\ P, \pi, j \models \psi_1$

A path formula ψ holds in a computation π, denoted by $P, \pi \models \psi$, if $P, \pi, 0 \models \psi$. For a state formula φ, φ holds on P, denoted by $P \models \varphi$, if for every initial state s we have $P, s \models \varphi$, where s stands for the history with one state in it. When the program P is clear from the context, we may write $\sigma \models \varphi$ for a state formula φ or $\pi, i \models \psi$ for a path formula ψ.

6.2. Checking CTL^*_{lp} Formulae

In this section, we describe the details of our CTL^*_{lp} model-checking procedure ProveCTL$^*_{lp}$ presented in Alg. 6. First, recall that a translation from CTL^*_{lp} to CTL^* as a solution to verifying CTL^*_{lp} could be non-elementary, and a translation algorithm may induce combinatorial

ALGORITHM 6: Extending our recursive CTL^* verification procedure to support CTL^*_{lp}.

```
1  Let rec ProveCTL*_lp (θ, P, P_D) : (formula, bool, program, program) =
2      (L, E, Vars) = P
3      match (θ) with
4          φ : stateformula →
5              match (φ) with
6                  θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 | Aθ'_1 ∘^± θ'_2 | Eθ'_1 ∘^± θ'_2 →
7                      (a_θ'_1, Path_1, P, P_D) = ProveCTL*_lp(θ'_1, P, P_D)
8                      (a_θ'_2, Path_2, P, P_D) = ProveCTL*_lp(θ'_2, P, P_D)
9                  A∘^±θ' | E∘^±θ' →
10                     (a_θ'_1, Path_1, P, P_D) = ProveCTL*_lp(θ', P, P_D)
11                     (a_θ'_2, Path_2) = (False, False)
12             match (φ) with
13                 α → a_θ = α;
14                 θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 | A∘θ' | E∘θ' | Aθ'_1 ∘ θ'_2 | Eθ'_1 ∘ θ'_2 →
15                     φ' = Replace(φ, a_θ'_1, a_θ'_2)
16                     if Path_1 ∨ Path_2 then
17                         a_θ = QuantElim(CTL(P_D, φ'), φ)
18                     else
19                         a_θ = CTL(P, φ')
20                 A∘^-1θ' | E∘^-1θ' | Aθ'_1 ∘^-1 θ'_2 | Eθ'_1 ∘^-1 θ'_2 →
21                     (a_θ, P, P_D) = AddHistory(φ, ∘^-1, a_θ'_1, a_θ'_2, P, P_D)
22             Path = False
23         ψ : pathformula →
24             match (ψ) with
25                 θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 | θ'_1 ∘^± θ'_2 →
26                     (a_θ'_1, Path_1, P, P_D) = ProveCTL*_lp(θ'_1, P, P_D)
27                     (a_θ'_2, Path_2, P, P_D) = ProveCTL*_lp(θ'_2, P, P_D)
28                 ∘^±θ' →
29                     (a_θ'_1, Path_1, P, P_D) = ProveCTL*_lp(θ', P, P_D)
30                     (a_θ'_2, Path_2) = (False, False)
31             match (ψ) with
32                 θ'_1 ∨ θ'_2 | θ'_1 ∧ θ'_2 →
33                     a_θ = Approximate(ψ, a_θ'_1, a_θ'_2)
34                     Path = Path_1 ∨ Path_2
35                 θ'_1 ∘ θ'_2 | ∘θ' →
36                     ψ' = Approximate(ψ, a_θ'_1, a_θ'_2)
37                     a_θ = CTL(P_D, ψ')
38                     Path = True
39                 θ'_1 ∘^-1 θ'_2 | ∘^-1θ' →
40                     (a_θ, P, P_D) = AddHistory(ψ, ∘^-1, a_θ'_1, a_θ'_2, P, P_D)
41                     Path = False
42     return (a_θ, Path, P, P_D)
```

ALGORITHM 7: AddHistory produces history variables corresponding to past-connectives in CTL^*_{lp}.

Let AddHistory$(\psi, \circ^{-1}, a_{\theta_1}, a_{\theta_2}, P, P_D)$ *: formula, past-operator, program, program* =
  $a_\theta = H_\psi$
  match $(\circ^{-1})$ **with**
    $\mathsf{F}^{-1} \rightarrow$
      $\iota = (H'_\psi = a'_{\theta_1})$; $\rho_h = (H'_\psi = H_\psi \vee a'_{\theta_1})$
    $\mathsf{G}^{-1} \rightarrow$
      $\iota = (H'_\psi = a'_{\theta_1})$; $\rho_h = (H'_\psi = H_\psi \wedge a'_{\theta_1})$
    $\mathsf{X}^{-1} \rightarrow$
      $\iota = (H'_\psi = \text{False})$; $\rho_h = (H'_\psi = a_{\theta_1})$
    $\mathsf{W}^{-1} \rightarrow$
      $\iota = (H'_\psi = a'_{\theta_1} \vee a'_{\theta_2})$; $\rho_h = (H'_\psi = (H_\psi \wedge a'_{\theta_1}) \vee a'_{\theta_2}))$
    $\mathsf{U}^{-1} \rightarrow$
      $\iota = (H'_\psi = a'_{\theta_2})$; $\rho_h = (H'_\psi = (H_\psi \wedge a'_{\theta_1}) \vee a'_{\theta_2})$
  $P =$ InstrumentHistory$(H_\psi, \iota, \rho_h, P)$
  $P_D =$ InstrumentHistory$(H_\psi, \iota, \rho_h, P_D)$
  return (a_θ, P, P_D)

ALGORITHM 8: InstrumentHistory embeds conditions over history variables within a transition system P.

Let InstrumentHistory(H, ι, ρ_h, P) *: program* =
  $(\mathcal{L}, E, \mathsf{Vars}) = P$
  foreach $(\ell, \rho, \ell') \in E$ **do**
    if $\ell = \ell_I$ **then**
      $\rho = \rho \wedge \iota$
    else
      $\rho = \rho \wedge \rho_h$
  $\mathsf{Vars} = \mathsf{Vars} \cup \{H\}$
  return P

explosions, even with limited temporal height [Laroussinie and Schnoebelen 1995; Kupferman et al. 2012]. We stress again that the lower bound in [Laroussinie and Schnoebelen 1995] can be expressed in our fragment. We thus introduce a procedure that extends ProveCTL* by introducing the sub-procedures AddHistory and InstrumentHistory, which serve to introduce history variables per past-connective present within a CTL^*_{lp} formula. We provide an overview of these sub-procedures below, followed by a more in-depth explanation regarding these extensions.

AddHistory & InstrumentHistory are procedures that produce a precondition for a past sub-formula by introducing history variables into the program. A history variable tracks the state of a consequent nested temporal formula within a program. AddHistory creates a history variable and its appropriate satisfying assertions tailored to the past-connective that is being verified. InstrumentHistory extends the transitions of P and P_D by instrumenting the assertions updating the truth values of the introduced history variable. The Boolean truth value of the history variable within a program's computation corresponds to the truth values of the past formula in a given history. The history variable produced can thus serve as the precondition for the past sub-formula at hand.

We now describe the details of these additional procedures and their use in ProveCTL$^*_{lp}$, followed by a detailed example that explores the usage of history variables. As with ProveCTL*, we generate a determinized copy of the program, P_D, using the procedure Determinize. This program is then passed to ProveCTL$^*_{lp}$ along with the original program P and a CTL$^*_{lp}$ property θ. Our extension can be observed on lines 20 – 21 and lines 39 – 41, where given a CTL$^*_{lp}$ formula, we not only deconstruct the formula to differentiate between state and path sub-formulae, but also between past and future sub-formulae. On line 20, if θ can be identified as a state formula with a past temporal operator, then we carry out the set of actions on line 21. If θ is identified as a past path formula on line 39, we carry out the set of actions on line 40.

Note that AddHistory indeed accepts the temporal operator $\circ^{-1}$ in addition to the sub-formula it operates on. This highlights a subtle difference between the treatment of state formulae in the algorithm. In the treatment of future-state formulae, an existing model checker for CTL is called. However, in the case of the past, a state formula characterizes a set of histories rather than a set of states. Thus, a model checker would have to return a characterization of the set of histories satisfying a given state formula rather a precondition characterizing a set of states. The approach we take is to add history variables to the program, hence allowing us to describe histories using preconditions that refer to the introduced history variables. Hence, we further partition a state-formula in AddHistory by treating the temporal operator as a path formula in order to instrument the correct history variable corresponding to the past-temporal operator $\circ^{-1}$. We now turn to the detailed description of AddHistory.

6.2.1. AddHistory & InstrumentHistory. In Alg. 7, we present a conversion from linear-past formulae to corresponding history variable conditions to be embedded into the programs P and P_D. That is, reasoning pertaining to a linear-past formula is reduced to conditions over a history variable that captures the truth value of the CTL$^*_{lp}$ formula. Our procedure is given a past-temporal operator $\circ^{-1}$, its corresponding linear-past formula ψ, and two preconditions, $a_{\theta_1}$and a_{θ_2}, corresponding to the satisfaction of the nested CTL$^*_{lp}$ formulae which appear within ψ. This aligns with how ProveCTL$^*_{lp}$ will recursively iterate over each inner sub-formula followed by search for the preconditions of the outer sub-formulae dependent on it, thus these preconditions would have already been priorly generated. Due to the structure of CTL$^*_{lp}$, note that a_{θ_1} and a_{θ_2} are either preconditions describing state formulae, or preconditions describing the histories satisfying past path formulae. It follows that both are expressed in terms of the variables of P alone and do not refer to the prophecy variables that form part of P_D. Recall that a_{θ_2} is a conditional parameter utilized only when a CTL$^*_{lp}$ formula requiring two properties (i.e., W^{-1}, U^{-1}) is given.

On line 2 we generate a unique history variable, H_ψ, corresponding to the past-temporal operator $\circ^{-1}$ to be analyzed. The history variable is indeed a Boolean variable that has the value *true* if the history of the computation so far satisfies the past property, and is *false* otherwise. We thus define conditions ι and ρ_h and assign them assertions that depend on $\circ^{-1}$. The condition ι is to be instrumented in the initial transitions of P and P_D, that being transitions leaving ℓ_I, while ρ_h is to be instrumented in the remaining transitions. If the aforementioned temporal operator is:

— F^{-1}, on line 5, H_ψ is assigned the truth valuation of the atomic precondition a'_{θ_1} in ι.[3] For the remaining transitions, ρ_h, H_ψ becomes true and stays true if a'_{θ_1} is satisfied at least once. These conditions reflect that sometime in the past, a_{θ_1} held.

[3]Recall that a'_{θ_1} refers to the value of a_{θ_1} after an update, that is, references to all variables would be replaced by references to primed versions.

— G^{-1}, on line 7, the dissatisfaction of a'_{θ_1} will cause H_ψ to become false and stay false in ρ_h. That is, in order for H_ψ to hold, then the valuation of a'_{θ_1} must always remain true, denoting that a_{θ_1} must have always held in the past.
— X^{-1}, on line 9, H_ψ is initially false in ι, given that there exists no previous state in P and P_D where a'_{θ_1} can be satisfied. As for the remaining transitions, if a_{θ_1} is satisfied, indicating the valuation before an update, then H_ψ is true. That is, H_ψ only holds if a_{θ_1} held in the immediate previous state.
— W^{-1}, on line 11, H_ψ is assigned the truth valuation of the disjunction of a'_{θ_1} or a'_{θ_2} in ι. For ρ_h, H_ψ is satisfied if a'_{θ_2} holds, otherwise the valuation of a'_{θ_1} must be true in addition to H_ψ being previously true. These conditions thus reflect that in the past, a_{θ_2} may hold before a_{θ_1} holds indefinitely.
— U^{-1}, on line 13, H_ψ is assigned the truth valuation of the atomic precondition a'_{θ_2} in ι. As for ρ_h, H_ψ is assigned the same valuation as in W^{-1}. It is the initialization state that enforces that a_{θ_2} held at some time in the past, and a_{θ_1} has been holding ever since. H_ψ can only ever become true in ι if a_{θ_2} holds, thus in ρ_h, $H_\psi \wedge a'_{\theta_1}$ will be falsified until a_{θ_2} becomes true at least once.

Once ι and ρ_h are appropriately assigned in AddHistory, InstrumentHistory is called on lines 14 and 15 with H_ψ, ι, and ρ_h. InstrumentHistory instruments the conditions over our history variable H_ψ within the programs P and P_D, respectively. Both ι and ρ_h are defined in terms of variables of P and can be instrumented into both P and P_D. As demonstrated in Alg. 8, a program is iterated on line 3. When an edge $(\ell, \rho, \ell') \in E$ is reached containing $\ell = \ell_I$, that is the initial location, then ι is conjuncted to the condition ρ. Otherwise, ρ_h, our condition over the history variable H, is conjuncted with ρ. The modified program with the history variable H is then returned on line 9.

Finally, on line 16 in Alg. 7, we return the transformed programs P and P_D alongside the history variable H_ψ as an atomic proposition, serving as the precondition of our linear-past formula. Given that we encode linear-past formulae within our programs, it is sufficient to return H_ψ as the precondition given that its truth valuation will be contingent upon the newly embedded transitions. We now show that AddHistory is deterministic, that is, the computations of the resulting program are the computations of the original program annotated by additional information.

Theorem 6.1. *Consider a sub-formula ψ in which the outermost operator is a past temporal operator. Given a program $\hat{P} = (\mathcal{L}, E, \mathsf{Vars})$ and conditions a_{θ_1} and a_{θ_2}, let $(\_, \hat{P}', \_) = \text{AddHistory}(\psi, a_{\theta_1}, a_{\theta_2}, \hat{P}, \_)$. Then, for every computation π of $\hat{P}$ there is a unique computation π' of $\hat{P}'$ such that $\pi' \Downarrow_{\mathsf{Vars}} = \pi$, and all computations of $\hat{P}'$ are of this form.*

Proof.

— We construct π' by extending π with assignment for the history variables. We do this by induction on the positions in a computation π of $\hat{P}$. For the initial location, a unique value for the history variable H'_ψ can be determined by a'_{θ_1} and a'_{θ_2}. This is carried out by going over the five options for ι in Alg. 7.
By induction given that the values of H_ψ are determined up to some location i, then it is the case that the value of H'_ψ is determined by the value of H_ψ and a'_{θ_1} and a'_{θ_2}. This is carried out by one of the five options for ρ_h in *Alg.* 7. It thus follows that π is a computation of $\hat{P}'$, as a'_{θ_1} and a'_{θ_2} are givens and thus H'_ψ can be determined over every computation π.
— In the other direction, consider a computation π' of $\hat{P}'$. For every transition $(\ell, \hat{\rho}, \ell')$ of $\hat{P}'$ we know that there is a transition (ℓ, ρ, ℓ') of P_1 such that $\hat{\rho} = \rho \wedge \alpha$, where α is a

condition produced by AddHistory (either ι or ρ_h in terms of Alg. 7). It follows that $\pi' \Downarrow_{\mathsf{Vars}}$ is a computation of $\hat{P}$, as the valuation obtained by restricting the valuation to variables in Vars remains the same.

□

Consider a past formula ψ. We now show that given sound preconditions for the sub-formulae nested within ψ, AddHistory soundly approximates the truth of ψ. This approximation is due to the value of preconditions for the sub-formulae themselves being an over-approximation.

Theorem 6.2. *Consider a past path formula ψ. Given a program $P = (\mathcal{L}, E, \mathsf{Vars})$ and the preconditions a_{θ_1} and a_{θ_2} computed for the sub-formulae of ψ. Suppose that for every history σ such that $\sigma \models a_{\theta_i}$ we have $P, \sigma \models \mathsf{A}\psi_i$. Let $(\_, P', \_) = \textsc{AddHistory}(\psi, a_{\theta_1}, a_{\theta_2}, P, \_)$. If $P', \sigma \models H_\psi$ then $P, \sigma \Downarrow_{\mathsf{Vars}} \models \mathsf{A}\psi$.*

Proof. By Theorem 6.1, the premises of the Theorem are well defined. Indeed, given a history σ of $\textsc{AddHistory}(\psi, a_{\theta_1}, a_{\theta_2}, P, \_)$ the history $\sigma \Downarrow_{\mathsf{Vars}}$ is well defined and for every computation σ' of P there is a history σ of $\textsc{AddHistory}(\psi, a_{\theta_1}, a_{\theta_2}, P, \_)$ such that $\sigma' = \sigma \Downarrow_{\mathsf{Vars}}$.

We consider the case of past path formulae.

— Suppose that $\psi = \theta_1 \mathsf{U}^{-1} \theta_2$. By assumption a_{θ_1} and a_{θ_2} are sound. We proceed by induction on the length of σ. If $|\sigma| = 1$ then the value of H_ψ soundly approximates the truth value of ψ as a_{θ_2} is sound and H'_ψ is initialized by ι to a'_{θ_2}. If $|\sigma| > 1$ then the value of H_ψ soundly approximates the truth value of ψ as a_{θ_1} and a_{θ_2} are sound and ρ_h updates H'_ψ to $(H_\psi \wedge a'_{\theta_1}) \vee a'_{\theta_2}$.

— The cases of $\psi = \theta_1 \mathsf{W}^{-1} \theta_2$, $\psi = \mathsf{G}^{-1}\theta_1$, and $\psi = \mathsf{F}^{-1}\theta_1$ are similar.

— Suppose that $\psi = \mathsf{X}^{-1}\theta_1$. By assumption a_{θ_1} is sound. We proceed by induction on the length of σ. If $|\sigma| = 1$ then the value of H_ψ is the truth value of ψ as H'_ψ is initialized by ι to false. If $|\sigma| > 1$ then the value of H_ψ soundly approximates the truth value of ψ as a_{θ_1} is sound and ρ_h updates H'_ψ to a_{θ_1}.

□

We note that pure-past formulae can include disjunctions and conjunctions. However, given that the precondition for $\alpha \wedge \beta$ will be the conjunction of the preconditions for α and β (and similarly for disjunction), the soundness of using Boolean connectives is immediate.

6.2.2. ProveCTL$^*_{lp}$. We return to our main algorithm in Alg. 6. The treatment of path formulae is somewhat different from our original ProveCTL*. Future temporal operators (lines 36 – 38) are treated just like the previous case. Past temporal operators (lines 39 – 41) are deterministically encoded as history variables and depend only on the variables of P. Thus, we set Path to False. Finally, Boolean connectives can be either pure-past formulae or include future temporal operators in them. In both cases, the precondition is set to the Boolean combination of the preconditions for the sub-formulae (as is masked by the call to CTL in the previous algorithm). However, the decision of whether the check should continue over P or P_D depends on the values of Path$_1$ and Path$_2$. Accordingly we set Path to their disjunction on line 34.

Theorem 6.3. *If $\textsc{Verify}(\theta, P)$ returns true for a program $P = (\mathcal{L}, E, \mathsf{Vars})$ then, $P \models \theta$.*

PROOF. We show by induction on the number of path quantifiers in a CTL^*_{lp} formula θ that the set of states computed as satisfying θ returned from PROVECTL^*_{lp} is sound. That is, for a program P returned from PROVECTL^*_{lp} and a history σ, if $P, \sigma \models a_\theta$ then $P, \sigma \Downarrow_{\mathsf{Vars}}$ satisfies θ.

— Consider a state formula $\mathsf{A}\psi$, where ψ does not include further path quantification. Suppose that $P, \sigma \models a_\psi$.
The computation of a_ψ uses recursive calls to ADDHISTORY for the past sub-formulae of ψ and calls to APPROXIMATE() with preconditions for the future sub-formulae of ψ. Every call to ADDHISTORY changes P and P_D by adding history variables to them. By induction on the structure of ψ and repeated use of of Theorem 6.2 we can show that every preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to ADDHISTORY are sound. Then, by repeated use of Theorem 4.2 we can show that every preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to APPROXIMATE() are also sound.
The precondition a_θ is obtained either in line 17, 19, or 21. If it is obtained in line 17, then it is obtained from universal quantification of a sound approximation of $\mathsf{A}\psi$ on the last version of P_D. If it is obtained in line 19, then it is obtained from a call to a (sound) CTL model checker. If it is obtained in line 21, then by Theorem 6.2 it is sound.
It follows that in P_D for every possible valuation v of the prophecy variables either σ has no infinite paths starting from it or σ satisfies $\mathsf{A}\hat{\psi}$ in P_D, where $\hat{\psi}$ is obtained from ψ by repeatedly replacing sub-formulae by their approximated versions as done by recursive calls of APPROXIMATE().
Consider a path π that starts in $\sigma \Downarrow V$ in P. By Theorems 4.1 and 6.1 the path π satisfies ψ.

— Consider a state formula $\mathsf{E}\psi$, where ψ does not include further path quantifications. Suppose that $P, \sigma \models a_\psi$.
As in the universal case, a_ψ is obtained by using recursive calls to ADDHISTORY for the past sub-formulae of ψ and calls to APPROXIMATE() with preconditions for the future sub-formulae of ψ. Every call to ADDHISTORY changes P and P_D by adding history variables to them. By induction on the structure of ψ and repeated use of Theorem 6.2 we can show that every preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to ADDHISTORY are sound. By induction on the structure of ψ and repeated use of Theorem 4.2 we can show that every preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$ appearing in the calls to APPROXIMATE() are sound.
The precondition a_θ is obtained either in line 17, 19, or 21. If it is obtained in line 17, then it is obtained from existential quantification of a sound approximation of $\mathsf{E}\psi$ on the last version of P_D. If it is obtained in line 19, then it is obtained from a call to a (sound) CTL model checker. If it is obtained in line 21, then by Theorem 6.2 it is sound.
If follows that in P_D for some possible valuation v of the prophecy variables σ has some computation starting from it and σ satisfies $\mathsf{E}\hat{\psi}$ in P_D, where $\hat{\psi}$ is obtained from ψ by repeatedly replacing sub-formulae by their approximated versions as done by recursive calls of APPROXIMATE(). It follows that there is a computation π in P_D that starts in σ such that $P_D, \pi, |\sigma| - 1 \models \psi$.
Consider a computation π that starts in σ in P_D and consider their projections $\pi' = \pi\Downarrow_{\mathsf{Vars}}$ and $\sigma' = \sigma\Downarrow_{\mathsf{Vars}}$ on the variables of P. By Theorem 4.1 π' is a computation of P. By Theorems 4.2 and 6.2 it follows that $P, \pi', |\sigma'| - 1 \models \psi$.

— In the case of a state formula θ that includes nesting of path quantifiers, the proof proceeds as before. This part relies on the structure of θ being in negation normal form and the soundness of previous approximations of $a_{\theta'}$ for every state sub-formula θ' of θ.

□

6.3. Interaction of Histories and Prophecies

In this section, we discuss what would be required in order to extend our algorithm to handle *full* CTL^*_{lp}. The fraction of CTL^*_{lp} that we consider ensures that there are no references to the future (*i.e.*, prophecy variables) appearing inside references to the past (*i.e.*, history variables). Indeed, the definition of past formulae τ ensures that the direct sub-formlae of a past operator are either state formulae, past formulae, or Boolean operators that nest them.

Consider the removal of this restriction and an attempt to use our algorithm in the case of a future path formula immediately nested within a past formula. Due to the determinization arising from verifying a path formula, the preconditions that describe the approximation of states that satisfy a future path formula could refer to the values of prophecy variables. Such preconditions are relevant only with respect to P_D as P does not include the prophecy variables. Now consider a past sub-formula that refers to such a precondition. The history variable instrumented in the program would describe the truth value of the past formula. The assertions that govern the truth value of such a history variable – ι and ρ_H, as described in AddHistory – would thus include a reference to prophecy variables. It follows that we would be able to add these history variables only to P_D and not to P. This would be sound for P_D and would produce correct approximations for P_D. However, as in our CTL^* algorithm, at some point the algorithm reaches the path quantification within which these path formulae are nested. We would need to "collapse" the preconditions that now refer to both prophecy variables and history variables to be relevant to P. Preconditions containing prophecy variables would be handled by the appropriate quantification as is done now in QuantElim. However, the conversion of path characterization back to state characterization over preconditions alone is not sufficient in the case of CTL^*_{lp}. We must quantify not only over the preconditions, but the transitions of P_D. Just as in QuantElim, we seek to acquire the proper set of states that satisfy formulae, which have been instrumented into the program as assertions over history variables, given that these assertions may depend on prophecy variables that have been produced by previous calls to ProveCTL$^*_{lp}$.

Now consider if our path quantification was indeed universal, then universally quantifying the assertions ι and ρ_H would be sufficient to translate the truth of the history variables to P. However, such is not the case with existential path quantification. It is not clear how one can embed history variables into P if they do reason about prophecy variables, and require existential quantification. We have chosen not to include the universal quantification option formally here as it would lead to the definition of a very complex fragment of CTL^*_{lp}, where once future is used within past, it can be used only within universal path quantification (and this remains the case also for the state formulae that contain this part).

We demonstrate this limitation further with a counterexample. First, consider the program in Fig. 2(a) and the property $\mathsf{EX}^{-1}\mathsf{X}^{-1}x = 0$. Clearly, the property holds in ℓ_2 from the second iteration and onwards of ℓ_2. It follows that the locations and variables of the program do not provide sufficient information to express the truth value of this property, thus some information must be added to the program in order to be able to express the truth value of the formula [Lichtenstein et al. 1985]. This is the role of the history variables – instrument information to the program that enables us to distinguish between histories that end in the same state of the original program. Now consider the formulae $\varphi_0 = \mathsf{F}^{-1}\mathsf{FG}\ x = 0$, $\varphi_1 = \mathsf{F}^{-1}\mathsf{FG}\ x = 1$, $\mathsf{E}\varphi_0$, $\mathsf{E}\varphi_1$, and $\mathsf{E}\varphi_1 \wedge \varphi_2$. This would required the usage of the determinized program in Fig. 2(b) for our analysis. The precondition for $\mathsf{AG}\ x = 0$ is ℓ_2. The precondition for $\mathsf{AF}\ \ell_2$ is $a_0 = \ell_2 \vee (\ell_1 \wedge n_{\ell_1} \geq 0)$. The precondition for $\mathsf{AG}\ x = 1$ is $\ell_1 \wedge n_{\ell_1} < 0$. The precondition for $\mathsf{AF}(\ell_1 \wedge n_{\ell_1})$ is $a_1 = \ell_1 \wedge n_{\ell_1} < 0$. Now, adding a history variable for $\mathsf{F}^{-1}a_0$ would add the condition $H'_0 = n'_{\ell_1} \geq 0$ to the initial transition, $H'_0 = H_0$ to the transition looping on ℓ_1 and $H'_0 = \textsc{True}$ to all transitions entering ℓ_2. Adding a history variable for $\mathsf{F}^{-1}a_1$ would add the condition $H'_1 = n'_{\ell_1} < 0$ to the initial transition, $H'_1 = H_1$ to all other transitions.

If we attempt to introduce these history variables in P once quantification is reached, we arrive at a problem. Indeed, $\mathsf{E}\varphi_0$ should be true for every state of P and $\mathsf{E}\varphi_1$ should be true for ℓ_1. If we indeed were to existentially quantify the introduced prophecy variables over the transitions in P_D as necessitated by E, $\mathsf{E}(\varphi_0 \wedge \varphi_1)$ would result in being true on all transitions, however, this is not sound as it is indeed false everywhere. It follows that existential quantification over transitions is not sufficient to transform history variables instrumented in P_D into sufficient conditions over P.

7. DEMONSTRATING CTL^*_{lp}

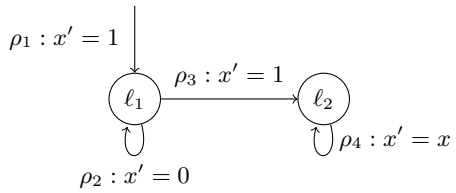
In this section, we provide a CTL^*_{lp} example that demonstrates the usage of history variables to provide a comprehensive view of how the CTL^* algorithm extends to verifying CTL^*_{lp}. Consider the program in Fig. 4(a) and the property $\mathsf{EGF}\mathsf{G}^{-1}\ x = 1$ stating that there exists some path such that infinitely often there is a state in which $x = 1$ has always held in the past. The property clearly holds for the program as evidenced by the path $(\ell_1, \langle x \mapsto 1\rangle)((\ell_2, \langle x \mapsto 1\rangle))^\omega$, which never enters the loop in ℓ_1 and continues to remain in the loop at ℓ_2 forever. Importantly (for the example), the path does pass through ℓ_1, where the property $\mathsf{AF}\mathsf{G}^{-1}\ x = 1$ does not hold.

As discussed, in order to check this property we recursively handle its sub-formulae. The most inner sub-formula is a past sub-formula, namely $\mathsf{G}^{-1}\ x = 1$. We call the function AddHistory with the precondition $x = 1$, given that the precondition of an atomic proposition is the atomic proposition itself. AddHistory produces a history variable corresponding to the past-connective G^{-1} and calls upon InstrumentHistory to add conjuncts to the transitions of P and P_D that update the value of this new history variable. In Fig. 4(b), we show the history variable $H_{\mathsf{G}^{-1}}$ introduced in P, and in Fig. 4(c) in P_D. In the initial state, $H'_{\mathsf{G}^{-1}}$ is set to the Boolean valuation of $x' = 1$. For the remaining transitions, if $x' = 1$ is satisfied and $H_{\mathsf{G}^{-1}}$ is true, indicating the valuation before an update, then $H'_{\mathsf{G}^{-1}}$ is true after the update. The history variable $H_{\mathsf{G}^{-1}}$ is then returned by AddHistory as the precondition satisfying $\mathsf{G}^{-1}\ x = 1$. We now continue with the next inner sub-formula with $H_{\mathsf{G}^{-1}}$ replacing $\mathsf{G}^{-1}\ x = 1$. Namely, $\mathsf{F}\ (H_{\mathsf{G}^{-1}})$.

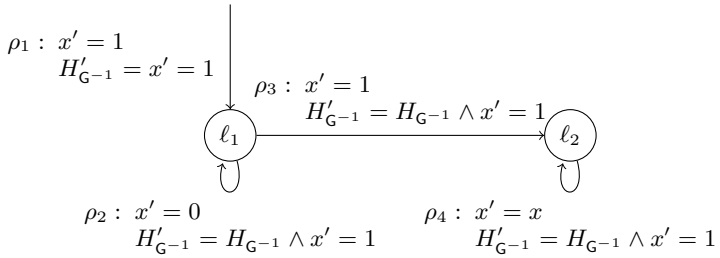
We identify that $\mathsf{F}\ H_{\mathsf{G}^{-1}}$ is a path sub-formula, and thus produce the over-approximated CTL formula $\mathsf{AF}(H_{\mathsf{G}^{-1}})$, which is returned from Approximate. The property $\mathsf{AF}(H_{\mathsf{G}^{-1}})$ does not hold on ℓ_1 in P. From ℓ_1, the nondeterministic choices to ρ_2 and ρ_3 mean that not all successors satisfy $H_{\mathsf{G}^{-1}}$. In order to reason about the original (path) sub-formula $\mathsf{F}\ H_{\mathsf{G}^{-1}}$, we must be observing sets of paths, not states. Recall that we over-approximated our formula in a way that allows us to only reason about states, we thus symbolically determinize the program to simultaneously simulate all possible related paths through the control flow graph and try to separate them to originate from distinct states in the program.

As before, Determinize returns a new symbolically partially-determinized program in which a newly introduced prophecy variable, namely n_{ℓ_1} in Fig. 4(c), is associated with the branching-relation (ρ_2, ρ_3), and is used to make predictions about the occurrences of relations ρ_2 and ρ_3. As can be seen in Fig. 4(c), prophecy variables are initialized to a nondeterministic value, are reset whenever exiting the minimal SCS associated with their location, and are decremented whenever staying inside the same minimal SCS. As in the case of CTL^*, we now utilize an existing CTL model-checker to return an assertion characterizing the states in which the $\mathsf{F}\ (H_{\mathsf{G}^{-1}})$ holds by verifying the determinized program, denoted by P_D, using the over-approximated CTL formula $\mathsf{AF}\ (H_{\mathsf{G}^{-1}})$. The result of this CTL model-checking task over P_D is $a_{\mathsf{F}} = (\ell_1 \wedge n_{\ell_1} = 0 \wedge H_{\mathsf{G}^{-1}}) \vee (\ell_2 \wedge H_{\mathsf{G}^{-1}})$.

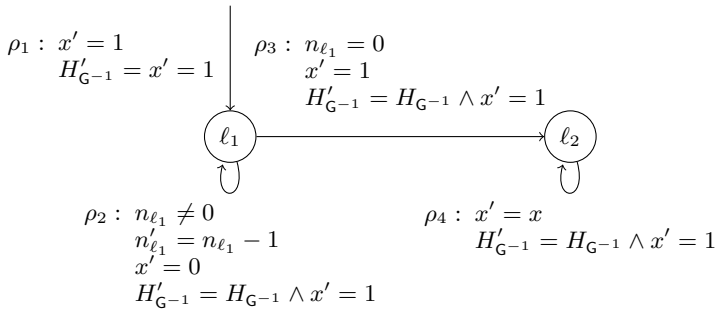
We then replace $\mathsf{F}\ H_{\mathsf{G}^{-1}}$ by a_{F} and finally arrive at our outermost CTL^*_{lp} formula $\mathsf{EG}\ a_{\mathsf{F}}$. As dictated by our ProveCTL$^*_{lp}$ algorithm, our final step is to verify $\mathsf{EG}\ a_{\mathsf{F}}$, a syntactically acceptable CTL formula. As discussed, we cannot simply use a CTL model checker as the path quantifier E exists within a larger relation context reasoning about paths given the



(a)



(b)



(c)

Fig. 4: (a) The control-flow graph of a program for which we wish to prove the CTL^*_{lp} property $\mathsf{EGFG}^{-1}\ x = 1$. (b) The control-flow graph after calling AddHistory to instrument the history variable necessary for reasoning about the past-connective G^{-1}. (c) The control-flow graph after calling Determinize, it includes the prophecy variable n_{ℓ_1}, corresponding to the nondeterministic branching-relation (ρ_2, ρ_3)

inner formula GF. We thus must use the CTL model-checker to verify $\mathsf{EG}a_{\mathsf{F}}$ over the same determinized program previously generated in Fig. 4(c). Our procedure then returns with the same precondition $(\ell_1 \wedge n_{\ell_1} = 0 \wedge H_{\mathsf{G}^{-1}}) \vee (\ell_2 \wedge H_{\mathsf{G}^{-1}})$. The set of states that satisfy the formula $\mathsf{EGFG}^{-1}\ x = 1$ are indeed those that start in ℓ_1 with $n_{\ell_1} = 0$.

Finally, we use quantifier elimination to existentially quantify out all introduced prophecy variables. Recall that if there is a state s in the original program, and some value of the prophecy variables v such that *all* paths from the combined state $(s, n_{\ell_1} = v)$ in P_D sat-

isfy the path formula then clearly, these paths give us a sufficient proof to conclude that $\mathsf{EGFG}^{-1}\ x = 1$ holds from s in P. In our case, the procedure QUANTELIM existentially quantifies our precondition given the path quantifier E, and produces the precondition $H_{\mathsf{G}^{-1}}$. History variables are instrumented in both P and P_D and the precondition can be evaluated over P. For this program, $H_{\mathsf{G}^{-1}}$ does indeed hold at the initial state. The program, as mentioned, does satisfy the formula.

8. CASE STUDY AND EVALUATION

In this section we discuss the results of our experiments with an implementation of the procedure from Alg. 6. Our implementation[4] is built as an extension to the open source project T2. T2 is an open-source framework that implements, combines, and extends techniques developed over the past decade aimed towards the verification of temporal properties of programs. T2 operates on an input format that can be automatically extracted from the LLVM compiler framework's intermediate representation, allowing T2 to analyze programs in a wide range of programming languages (*e.g.* C, C++, Objective C, ...). T2 allows users to (dis)prove temporal properties via a reduction to its *safety*, *termination* and *nontermination* analysis techniques [Brockschmidt et al. 2016].

8.1. Case Study

First, we report on a case study that requires the application of our extended PROVECTL$^*_{lp}$ algorithm presented in Alg. 6. Our case study concerns I/O request packets (IRP) in Windows Device Drivers and the requirement that each IRP must have a *Cancel* routine that allows the cancellation of an I/O operation. In Fig. 5, we thus provide an example in which an IRP is queued in order to set and clear its *Cancel* routine. When setting the *Cancel* routine for the IRP, one must use a spin lock, as shown on line 1, to protect the IRP pointer and the queue. A spin lock is a lock that causes a thread trying to acquire it to simply wait in a loop while repeatedly checking if the lock is available. However note that before queuing an IRP, despite it being protected by a spin lock, it is a requirement that drivers must mark an IRP as pending (using `IoMarkIrpPending`) before queuing it. In our example, the driver does indeed mark the IRP as pending on line 3 after acquiring a spin lock, and then continues to call the method `InsertTailList` on line 4, which queues the IRP in order to set and clear its *Cancel* routine. The *Cancel* routine is then set and cleared on lines 9 – 22. Given this requirement, we wish to verify the property requiring that drivers mark an IRP as pending using `IoMarkIrpPending` before queuing it, that is:

$$\mathsf{AG}(\texttt{InsertTailList()} \Rightarrow \mathsf{X}^{-1}\ (\neg\texttt{InsertTailList()}\ \mathsf{U}^{-1}\ \texttt{IoMarkIrpPending()}))$$

We thus call PROVECTL$^*_{lp}$ with the property above, the program in Fig. 5, which we will denote as P, and a determinized variation (P_D) attained from the DETERMINIZE algorithm previously discussed in Alg. 1. Supplementary information regarding how we interpret and parse a program's commands to attain P can be found in [Brockschmidt et al. 2016]. Given that we recursively partition our CTL^*_{lp} formula, we begin with the sub-formula $\neg$`InsertTailList()` U^{-1} `IoMarkIrpPending()` and identify it as a path formula containing a past-connective. We thus refer to our ADDHISTORY algorithm in Alg. 7. Given that `InsertTailList()` and `IoMarkIrpPending()` are call sites, they serve as the atomic propositions a_{θ_1} and a_{θ_2}, respectively. Our sub-formula is then matched on line 12 in Alg. 7 with U^{-1} in which ι corresponds to the condition to be instrumented at the initial state of P and P_D, that being ℓ_I, while ρ denotes the condition to be instrumented in the remaining transitions. That is, $\iota = (H'_{\mathsf{U}^{-1}} = a'_{\theta_2})$ and $\rho = (H'_{\mathsf{U}^{-1}} = (H_{\mathsf{U}^{-1}} \wedge a'_{\theta_1}) \vee a'_{\theta_2})$ are to to be instru-

[4]The source-code of our implementation and our benchmarks are available under the MIT open-source license at https://github.com/hkhlaaf/T2/.

```
1   KeAcquireSpinLock(&deviceContext->irpQueueSpinLock, &oldIrql);
2
3   IoMarkIrpPending(Irp);
4   InsertTailList(&deviceContext->irpQueue, &Irp->Tail.Overlay.ListEntry);
5
6   oldCancelRoutine = IoSetCancelRoutine(Irp, IrpCancelRoutine);
7   ASSERT(oldCancelRoutine == NULL);
8
9   if (Irp->Cancel) {
10
11      oldCancelRoutine = IoSetCancelRoutine(Irp, NULL);
12      if (oldCancelRoutine) {
13
14          RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
15
16          KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);
17          Irp->IoStatus.Status = STATUS_CANCELLED;
18          Irp->IoStatus.Information = 0;
19          IoCompleteRequest(Irp, IO_NO_INCREMENT);
20          return STATUS_PENDING;
21
22      } else {
23
24      }
25  }
26
27  KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);
28  return STATUS_PENDING;
```

Fig. 5: A Windows Device Driver driver setting a *Cancel* routine for an I/O request packet.

mented into P and P_D. Our uniquely synthesized history variable $H_{\mathsf{U}^{-1}}$ then serves as our precondition for our sub-formula. That is, a true valuation over $H_{\mathsf{U}^{-1}}$ would satisfy the sub-formula $\neg$`InsertTailList()` U^{-1} `IoMarkIrpPending()`. We then substitute $H_{\mathsf{U}^{-1}}$ in the original sub-formula to attain the CTL^*_{lp} formula $\mathsf{AG}($`InsertTailList()` $\Rightarrow \mathsf{X}^{-1}(H_{\mathsf{U}^{-1}}))$.

Our next inner sub-formula happens to be another path formula containing a past-connective. AddHistory would thus be called upon again with $a_{\theta_1} = H_{\mathsf{U}^{-1}}$ given that we substituted our previous linear-past sub-formula with its corresponding history variable. The initial transition ι is then assigned False while ρ is assigned $(H'_{\mathsf{X}^{-1}} = H_{\mathsf{U}^{-1}})$. As with our previous sub-formula, ι and ρ are also instrumented into the transition systems P and P_D, with $H_{\mathsf{X}^{-1}}$ serving as the precondition of $\mathsf{X}^{-1}(H_{\mathsf{U}^{-1}})$. We substitute our linear-past sub-formula once more with its associated history variable, and thus finally arrive to our outer-most CTL^*_{lp} formula $\mathsf{AG}($`InsertTailList()` $\Rightarrow H_{\mathsf{X}^{-1}})$.

Given the above transformations, every transition within P and P_D has now been embedded with history variable conditions corresponding to our inner linear-past sub-formulae. Our outer-most formula can now simply be treated as a CTL formula where a precondition can be acquired via existing CTL model checkers which return an assertion characterizing the states in which (`InsertTailList()` $\Rightarrow H_{\mathsf{X}^{-1}}$) holds. Recall that existing tools that support this functionality include [Beyene et al. 2013] and [Cook et al. 2014]. For this particular example, we will be utilizing our CTL model checker on P, given that all of our nested sub-formulae are not future path formulae, hence determinization is not required. In this

case, the model-checker does not return any counterexamples, deeming our precondition to be TRUE. We have thus proved that our property holds for Fig. 5.

8.2. Benchmarks

Program	LoC	Property	Time(s)	Res.
Cancel I/O	35	$\mathsf{AG}(\texttt{InsertTailList()} \Rightarrow \mathsf{X}^{-1}\ (\neg\texttt{InsertTailList()}\ \mathsf{U}^{-1}\ \texttt{IoMarkIrpPending()})$	1.0	✓
Cancel I/O	35	$\mathsf{AG}(\texttt{InsertTailList()} \Rightarrow\ (\mathsf{F}^{-1}\ \texttt{KeAcquireSpinLock()} \wedge \mathsf{AF}\ \texttt{KeReleaseSpinLock()}))$	0.1	✓
OS frag. 1	393	$\mathsf{AG}((\mathsf{EG}(\texttt{phi\_io\_compl} \leq 0)) \vee (\mathsf{EFG}(\texttt{phi\_nSUC\_ret} > 0))))$	17.4	×
OS frag. 1	393	$\mathsf{EF}((\mathsf{AF}(\texttt{phi\_io\_compl} > 0)) \wedge (\mathsf{AGF}(\texttt{phi\_nSUC\_ret} \leq 0))))$	23.8	✓
OS frag. 2	380	$\mathsf{EFG}((\texttt{keA} \leq 0 \wedge (\mathsf{AG}\ \texttt{keR} = 0)))$	13.7	✓
OS frag. 2	380	$\mathsf{EFG}((\texttt{keA} \leq 0 \vee (\mathsf{EF}\ \texttt{keR} = 1)))$	3.5	✓
OS frag. 3	50	$\mathsf{EF}(\texttt{PPBlockInits} > 0 \wedge (((\mathsf{EFG}\ \texttt{IoCreateDevice} = 0) \vee (\mathsf{AGF}\ \texttt{status} = 1)) \wedge (\mathsf{EG}\ \texttt{PPBunlockInits} \leq 0)))$	10.4	✓
PgSQL arch 1	106	$\mathsf{EFG}(\texttt{tt} > 0 \vee (\mathsf{AF}\ \texttt{wakend} = 0))$	1.5	×
PgSQL arch 1	106	$\mathsf{AGF}(\texttt{tt} \leq 0 \wedge (\mathsf{EG}\ \texttt{wakend} \neq 0))$	3.8	✓
PgSQL arch 1	106	$\mathsf{EFG}(\texttt{wakend} = 1 \wedge (\mathsf{EGF}\ \texttt{wakend} = 0))$	18.3	✓
PgSQL arch 1	106	$\mathsf{EGF}(\mathsf{AG}\ \texttt{wakend} = 1)$	10.3	✓
PgSQL arch 1	106	$\mathsf{AFG}(\mathsf{EF}\ \texttt{wakend} = 0)$	1.5	×
PgSQL arch 2	100	$\mathsf{AGF}\ \texttt{wakend} = 1$	1.4	✓
PgSQL arch 2	100	$\mathsf{EFG}\ \texttt{wakend} = 0$	0.5	×
Bench 1	12	$\mathsf{EFG}(\texttt{x} = 1 \wedge (\mathsf{EG}\ \texttt{y} = 0))$	0.2	✓
Bench 2	12	$\mathsf{EGF}\ \texttt{x} > 0$	0.1	✓
Bench 3	12	$\mathsf{AFG}\ \texttt{x} = 1$	0.1	✓
Bench 4	10	$\mathsf{AG}((\mathsf{EFG}\ \texttt{y} = 1) \wedge (\mathsf{EF}\ \texttt{x} \geq \texttt{t}))$	0.5	×
Bench 5	10	$\mathsf{AG}(\texttt{x} = 0\ \mathsf{U}\ \texttt{b} = 0)$	T/O	–
Bench 6	8	$\mathsf{AG}((\mathsf{EFG}\ \texttt{x} = 0) \wedge (\mathsf{EF}\ \texttt{x} = 20))$	0.1	×
Bench 7	6	$(\mathsf{EFGx} = 0) \wedge (\mathsf{EFGy} = 1)$	0.4	×
Bench 8	6	$\mathsf{AG}((\mathsf{AFG}\ \texttt{x} = 0) \vee (\mathsf{AFGx} = 1))$	0.5	✓

Fig. 6: Experimental evaluations of infinite-state programs drawn from the Windows OS, PgSQL, and 8 toy examples. There are no competing tools available for comparison.

We have drawn out a set of CTL* problems from industrial code bases. Examples were taken from the I/O subsystems of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. The tool was executed on an Intel x64-based 2.8 GHz single-core processor. CTL* allows us to express "possibility" properties, such as the viability of a system, stating that any reachable state can spawn a fair computation. Additionally, we demonstrate that we can now verify properties involving existential system stabilization, stating that an event can eventually become true and stay true from any reachable state. For example, "OS frag. 1", "OS frag. 3", "PgSQL arch 1", and "Bench 2" are verified using said properties, described in detail in Section 1.2.1. Our case study's results, demonstrating our CTL^*_{lp} extension, are also included under "Cancel I/O". We also include a few toy examples to further demonstrate further expressiveness of CTL* and its usefulness in verifying programs.

Given that our benchmarks tackle infinite-state programs, the only existing automated tool for verifying CTL* in the finite-state setting [Griffault and Vincent 2004] is not ap-

plicable. In Figure 6 we display the results of our benchmarks. For each program and its corresponding CTL* property to be verified, we display the number of lines of code (LoC), and report the time it took to verify a CTL* property (Time column) in seconds. We provide a **"Res."** column which indicates the results of our tool. A ✓ indicates that the tool was able to verify the property. Likewise, an × indicates that the tool failed to prove the property. The symbol "–" in the result column indicates that a result was not determined due to a timeout. A timeout or memory exception is indicated by T/O. A timeout is triggered if verification of an experiment exceeds 3000 seconds. Note that in various cases, we verify the same program using a CTL* property and its negation. Our tool thus allows us to prove each of the properties as well as disprove each of their negations.

Our experiments demonstrate the practical viability of our approach. Our runtimes show that our tool runs well within the range of performance previously exhibited by specialized tools such as as [Cook et al. 2007; Cook and Koskinen 2011; 2013; Beyene et al. 2013; Cook et al. 2014], which can only verify significantly less expressive properties over infinite-state programs. Our tool has successfully both verified and invalidated CTL* properties corresponding to their expected results for all but one of the benchmarks. This is due to the aforementioned limitation, that is, our countable nondeterministic determinization technique is not complete.

9. RELATED WORK

There are various algorithms for model checking CTL* for finite-state programs and other decidable settings. The approach of Emerson et al. [Emerson and Lei 1987] reduces a CTL* formula to μ-calculus using a system of fixed-point equations on relations with first-order quantifiers and equalities. This approach has been implemented in [Griffault and Vincent 2004], where a μ-calculus model checker is invoked after the translation. The approach described in [Clarke et al. 1999] calls for repeated calls to a (global) linear-time model checker. The linear-time model checker computes the set of states that satisfies a given path formula. This set of states can be used as a precondition that replace state sub-formulae of super-formulae that include the said path formulae.

Contrarily, we seek to verify the undecidable general class of infinite-state programs supporting both control-sensitive and integer properties. Given that μ-calculus model checking is polynomial-time equivalent to the solution of parity games [Emerson and Jutla 1999], one can conceive that the approach in [Beyene et al. 2014b] could potentially solve CTL* model checking if the latter were reduced to solving parity games by combining [Griffault and Vincent 2004] and [Emerson and Jutla 1999]. However, we note that the resulting infinite-state game would integrate the (first-order μ-calculus) property within the program making it difficult to extract invariants pertaining the program. For this reason, it is often the case that such a series of reductions inhibits tool performance. Furthermore, [Beyene et al. 2014b] requires a manual instantiation of the structure of assertions, characterizing subsets of the infinite-state game, that are to be found by their tool. One can think of our approach as an implementation of the technique described in [Clarke et al. 1999], but over infinite-state programs. We generalize an approach introduced by [Cook and Koskinen 2013], which reduces linear-time model checking to branching-time model-checking. We extend this approach to global model-checking instead of local model-checking by incorporating preconditions and existential path quantifications, in addition to various improvements to their technique.

Existing automated tools for verification of infinite-state programs support *either* branching-time only *or* linear-time only reasoning, e.g., [Bodden 2004; Cook et al. 2007; Cook and Koskinen 2011; 2013; Beyene et al. 2013; Cook et al. 2014; Song and Touili 2012]. The important distinction however is that these tools do not allow for the interaction between linear-time and branching-time formulae.

Finally, as we have previously discussed, we have adopted and repurposed a similar symbolic determinization technique introduced in [Cook and Koskinen 2011] for the verification

of LTL formulae in the infinite-state setting. Their symbolic determinization is based on the counterexample-guided refinement of generated tree counterexamples, or counterexamples with branching paths. That is, [Cook and Koskinen 2013] produce a semantics-preserving transformation that encodes the structure of the nested CTL formulae within the state space, allowing for the generation of tree counterexamples. This causes precondition generation for syntactically partitioned formulae to be no longer possible, limiting the interplay between linear-time operators and path quantifiers allowed by our strategy.

10. CONCLUDING REMARKS

We have introduced the first-known fully automatic method capable of proving CTL^* properties for infinite-state (integer) programs. This allows us, for the first time ever, to automatically verify properties of programs that mix branching-time and linear-time temporal operators. We have developed an implementation capable of automatically proving properties of programs that no tool could previously prove. The method underlying our tool is one that uses a symbolic representation capable of facilitating reasoning about the interaction between sets of states and sets of paths. In addition, we provide a novel methodology which extends our CTL^* procedure to the verification of a fragment of CTL^*_{lp}, providing users with an exponentially more succinct logic to reason about linear-past. We have introduced a transformation which embeds history variables corresponding to nested past-connective formulae within the transition system. That is, the history variables track the truth valuation of a past CTL^*_{lp} formula along a computation.

In future work, we hope to eliminate the limitations of our determinization procedure by potentially utilizing the technique introduced in [Cook et al. 2015] which allows for *some* interaction between linear-time and branching-time over fairness assumptions pertaining to a system's environment. Additionally, when specifying the correct behavior of systems, relating data at various stages of a computation is often crucial, as expressing program correctness often requires relating program data throughout different branches of an execution. However, CTL^*_{lp} alone cannot express this without the support of first-order quantification. There does exist one automated model-checking tool that supports first-order CTL [Beyene et al. 2014a]. The first-order quantification is encoded as a set of constraints within an existing CTL model-checker to obtain an automatic verifier for first-order CTL. However, it is unclear if a similar strategy can be integrated with our CTL^* model-checker, as the constraints reason about quantification over sets of states, and not paths. We thus hope to further investigate the aforementioned approach to extend the support of CTL^*_{lp} to include first-order logic.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees of both CAV and JACM for their work and their comments. In particular, the comments of one of the referees of JACM provided many helpful comments on how to improve the clarity of the paper and found an issue with an earlier version of one of our algorithms.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82 (1991), 253–284.

Bowen Alpern and Fred B. Schneider. 1986. *Recognizing Safety and Liveness*. Technical Report. Ithaca, NY, USA.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough static analysis of device drivers. *SIGOPS Operating Systems Review* 40 (April 2006), 73–85. Issue 4. DOI:http://dx.doi.org/10.1145/1218063.1217943

Tewodros A. Beyene, Marc Brockschmidt, and Andrey Rybalchenko. 2014a. CTL+FO Verification As Constraint Solving. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software (SPIN 2014)*. ACM, New York, NY, USA, 101–104.

Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014b. A Constraint-based Approach to Solving Games on Infinite Graphs. In *POPL '14*. ACM, 221–233.

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *CAV'13*. Springer, 869–882.

Nikolaj S. Bjørner, Anca Browne, Michael A. Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. 2000. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Form. Methods Syst. Des.* 16, 3 (2000), 227–270.

Eric Bodden. 2004. A Lightweight LTL Runtime Verification Tool for Java. In *OOPSLA '04*. ACM, 306–307.

Laura Bozzelli. 2008. The Complexity of CTL* + Linear Past. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg.

Marc Brockschmidt, , Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *TACAS '16 (To Appear)*.

E.M. Clarke, O. Grumberg, and D.A. Peled. 1999. *Model checking*.

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving that programs eventually do something good. In *POPL'07*. ACM, 265–276.

Byron Cook, Heidy Khlaaf, and Nir Piterman. 2014. Faster Temporal Reasoning for Infinite-State Programs. In *FMCAD '14*. FMCAD Inc, 16:75–16:82.

Byron Cook, Heidy Khlaaf, and Nir Piterman. 2015. Fairness for Infinite-State Systems. In *TACAS '15*. Springer Berlin Heidelberg, 384–398.

Byron Cook and Eric Koskinen. 2011. Making prophecies with decision predicates. In *POPL'11*. ACM, 399–410.

Byron Cook and Eric Koskinen. 2013. Reasoning about nondeterminism in programs. In *PLDI'13*. ACM, 219–230.

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. lexicographic termination proving. In *TACAS'13 (LNCS)*. Springer, 47–61.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

E. Allen Emerson and Joseph Y. Halpern. 1986. "Sometimes" and "Not Never"; Revisited: On Branching Versus Linear Time Temporal Logic. *J. ACM* 33, 1 (1986), 151–178.

E. Allen Emerson and Charanjit S. Jutla. 1999. The Complexity of Tree Automata and Logics of Programs. *SIAM J. Comput.* 29, 1 (1999), 132–158.

E. Allen Emerson and Chin-Laung Lei. 1987. Modalities for Model Checking: Branching Time Logic Strikes Back. *Sci. Comput. Program.* 8, 3 (1987), 275–306.

E. Allen Emerson and A. Prasad Sistla. 1984. Deciding Branching Time Logic. In *STOC '84*. ACM, 14–24.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. 1980. On the Temporal Analysis of Fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '80)*. ACM, New York, NY, USA, 163–173.

Alain Griffault and Aymeric Vincent. 2004. The Mec 5 model-checker. In *CAV'04 (LNCS)*. springer, 488–491.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. *SIGPLAN Not.* 43 (January 2008), 147–158. Issue 1.

Johan Kamp. 1968. Tense logic and the theory of linear order. (1968).

Yonit Kesten and Amir Pnueli. 2005. A compositional approach to CTL* verification. *Theor. Comput. Sci.* 331, 2-3 (2005), 397–428.

Orna Kupferman, Amir Pnueli, and Moshe Y. Vardi. 2012. Once and for All. *J. Comput. Syst. Sci.* 78 (2012), 981–996.

Leslie Lamport. 1980. "Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs. In *POPL '80*. ACM, 174–185.

F. Laroussinie and Ph. Schnoebelen. 1995. A Hierarchy of Temporal Logics with Past. In *Selected Papers of the Eleventh Symposium on Theoretical Aspects of Computer Science (STACS '94)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 303–324.

Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. 1985. The Glory of the Past. In *Proceedings of the Conference on Logic of Programs*. Springer-Verlag, London, UK, UK, 196–218.

Stephen Magill, Josh Berdine, Edmund M. Clarke, and Byron Cook. 2007. Arithmetic Strengthening for Shape Analysis. In *SAS'07*. Springer, 419–436.

Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems: safety*. Vol. 2. Springer.

Kenneth Lauchlin McMillan. 2006. Lazy Abstraction with Interpolants. In *CAV'06 (LNCS)*, Vol. 4144. Springer, 123–136.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, Washington, DC, USA, 46–57.

Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *LICS*. IEEE, Turku, Finland, 32–41.

Arthur Prior. 1957. Time and Modality. (1957).

Mark Reynolds. 2001. An Axiomatization of Full Computation Tree Logic. *The Journal of Symbolic Logic* 66, 3 (2001), pp. 1011–1057.

Fu Song and Tayssir Touili. 2012. Pushdown Model Checking for Malware Detection. In *TACAS'12*. ACM, 607–610.

Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*. 332–344.