University College London
Department of Computer Science

The Past, Present, and Future(s): Verifying Temporal Software Properties

Heidy Khlaaf

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science of University College London, April 2018

Declaration

I, Heidy Khlaaf confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Software systems are increasingly present in every aspect of our society, as their deployment can be witnessed from seemingly trivial applications of light switches, to critical control systems of nuclear facilities. In the context of critical systems, software faults and errors could potentially lead to detrimental consequences, thus more rigorous methodologies beyond the scope of testing need be applied to software systems. Formal verification, the concept of being able to mathematically prove the correctness of an algorithm with respect to a mathematical formal specification, can indeed help us prevent these failures.

A popular specification language for these formal specifications is temporal logic, due to its intuitive, yet precise expressions that can be utilized to *both* specify and verify fundamental properties pertaining to software systems. Temporal logic can express properties pertaining to safety, liveness, termination, non-termination, and more with regards to various systems such as Windows device drivers, kernel APIs, database servers, etc. This dissertation thus presents automated scalable techniques for verifying expressive temporal logic properties of software systems, specifically those beyond the scope of existing techniques. Furthermore, this work considers the temporal sub-logics fair-CTL, CTL*, and CTL$^*_{lp}$, as verifying these more expressive sub-logics has been an outstanding research problem.

We begin building our framework by introducing a novel scalable and high-performance CTL verification technique. Our CTL methodology is unique relative to existing techniques in that it facilitates reasoning about more expressive temporal logics. In particular, it allows us to further introduce various methodologies that allow us to verify fair-CTL, CTL*, and CTL$^*_{lp}$. We support the verification of fair-CTL through a reduction to our CTL model checking technique via the use of infinite non-deterministic branching to symbolically partition fair from unfair executions. For CTL*, we propose a method that uses an internal encoding which facilitates reasoning about the subtle interplay between the nesting of path and state temporal operators that occurs within CTL* proofs. A precondition synthesis strategy is then used over a program transformation which trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. Finally, we propose a linear-past extension to CTL*, that being CTL$^*_{lp}$, in which the past is linear and each moment in time has a unique past. We support this extension through the use of history variables over our CTL* technique.

We demonstrate the fully automated implementation of our techniques, and report our benchmarks carried out on code fragments from the PostgreSQL database server, Apache web server, Windows OS kernel, as well as smaller programs demonstrating the expressiveness of fair-CTL, CTL*, and CTL$^*_{lp}$ specifications. Together, these novel methodologies lead to a new class of fully automated tools capable of proving crucial properties that no tool could previously prove in the infinite-state setting.

Impact Statement

Outside Academia: Automated tools with enriched expressiveness capabilities are pivotal in the role of verifying software systems. The logics supported thus far within the verification community, that being CTL and LTL, 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". Existing temporal verifiers would thus not be sufficient in allowing a user to exhaustively verify various properties, as the supported logics simply are not able to express them. Contrarily, our automated verification techniques for fair-CTL, CTL*, CTL$^*_{lp}$ allow us to express such properties, in an even more succinct manner, including properties involving existential system stabilization and possibility properties. These novel expressive properties have been imperative in verifying systems such as Windows kernel APIs that acquire resources and APIs that release resources, as later shown by our experiments.

Our research thus shows potential for industrial applications, as support for increasingly expressive temporal logics is beneficial in industry given that it would pave way for a more exhaustive, succinct, and complete system analyses. Additionally, given that our techniques are modular and can be built upon existing CTL model-checkers, including those present within industry, the low effort of extending CTL verification to fair-CTL, CTL*, CTL$^*_{lp}$ verification would lead to a seamless and thus wide-use adoption of our techniques.

Inside Academia: We propose a framework utilizing a novel scalable and high-performance CTL verification technique to facilitate reasoning about more expressive temporal logics. We thus demonstrate that one can indeed reduce more expressive temporal properties, via history and prophecy variables that are to be annotated throughout a software's code, to existing well-supported logics. Our methods have been published in top-tier conferences and journals, and are accompanied by an open-source tool. Of course, such methods lend themselves to various limitations, however, this dissertation suggests research methods which could mitigate each of these limitations. We are thus advancing the dialogue of how increasingly expressive temporal logics can be better supported in the future, through our thorough discussion of our methodologies, limitations, and potential improvements. Furthermore, our open-source tool would allow for the adaptation and improvement of our research methodologies by the larger verification community.

Acknowledgements

I would like to express my gratitude to my official supervisor, Alexandra Silva, who has provided me with valuable advice regarding my career and work. Additionally, she has always supported me with regards to my capabilities and what I could accomplish, especially in critical times. I would also like to express my gratitude towards my previous supervisor, Byron Cook, for giving me tremendous opportunities in the field of verification, allowing me to study and work among the most talented and capable researchers in our community. Particularly, he had faith in my potential and abilities, despite doubts exhibited by others.

Most importantly, I would like to express how fortunate enough I am to have Nir Piterman as my primary supervisor. The lessons he taught me were numerous, on every level of my academic career. Intellectually, he offered invaluable insights and resources into the nature of our work. No question I asked was ever too silly or simple to answer, a stark difference from attitudes exhibited in the standard academic environment. He never failed to take the time to educate me on a multitude of theoretical concepts, and I am ultimately a better researcher because of his guidance. In research, his humbleness allowed for our regular productive research discussions, as he continuously entertained my ideas and solutions, always giving me the benefit of the doubt and the utmost respect. Finally, he always provided support and advice when I've faced dire emotional, political, and ethical hardships in the realm of academia. He consistently exhibited a great work ethic, and thus I am thankful that I had such a great role-model to follow in academia. This dissertation wouldn't have been made possible without his incredibly detailed feedback with regards to all aspects of this dissertation and our work together.

I met many friends and colleagues who made my PhD experience more meaningful. Not only did they provide me with their support and friendship, but also their thoughtful research insights whenever it was necessary: Tyler Sorensen, Klaus von Gleissenthall, Aws Albarghouthi, Zak Kincaid, Ruben Martins, Siddharth Krishna, Carsten Fuhs, Markus Rabe, Marc Brockschmidt, and many others.

To Ziyad Soobhan, my husband, life companion, and climbing partner, you have followed me through this journey on every step of the way, providing me with unwavering support despite the difficult circumstances faced. I distinctly recall my first paper deadline, where I was ruthlessly working at the university campus until 2 AM. There was not an accessible way for me to reach home, yet at the brink of dawn, you drove through central London and ensured I made it home safely, all despite having to work the next day. When I was at my lowest, and did not imagine it possible to see my whole PhD journey through, you lifted my spirits. You guided me in making the most difficult decisions I have had to make, and have always shown me that there was a way. Through thick and thin, we grew together in ways I could not imagine. Thank you for being in my life.

To Jaime Bue, the best friend one could ever ask for. Having been through all aspects of life together in our 13 years of friendship, this was no different. You were both a sympathetic and an empathetic support as we faced the challenges of graduate school together. You understood the challenges, frustrations, and mental battles I faced, and consistently comforted me in knowing that I certainly was not alone. I hope our friendship always continues to be as strong as it has been for the past decade.

Finally, thank you to all those I've met, from colleagues to acquaintances, who have provided me with any advice, no matter how little, that guided me through this journey.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

In recent years, we have witnessed a surge in the interest of formal verification for software systems, and not without reason. Formal verification, the concept of being able to mathematically prove the correctness of an algorithm with respect to a mathematical formal specification, can indeed help us prevent detrimental failures in safety-critical systems and avoid financially burdening bugs prevalent in software. As Edsger W. Dijkstra has famously claimed "*Program testing can be used to show the presence of bugs, but never to show their absence!*" Verification is thus crucial in that it is exhaustive in its measures to guarantee the absence of incorrect system behavior. Applications of verification techniques have ranged from aviation control, to financial systems, and is ever-expanding given this technological era. Yet the burden of formal proofs lies on the shoulders of those developing software systems, as many techniques call for the annotation of programs with correctness criteria (*e.g.* pre and post conditions) that are then to be formally proven. However, developers are often not versed in the formal methods of mathematics, thus undermining successful state-of-the-art research into numerous verification methodologies. Alas, there still exists limited techniques allowing for a uniform, fully automated scalable tool for proving intuitive specifications of software systems.

In this dissertation, we thus introduce the first known unifying, fully automated verification system culminating to the verification of a superset logic, known as CTL^*_{lp}, of the widely accepted specification language of temporal logic. Our efforts are established through introducing novel model checking methodologies for the verification of sub-logics such as CTL and fair-CTL that are conducive to a new class of fully-automated tools capable of proving crucial properties that no tool could previously prove. 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. Temporal logic thus allows for the specification of a crucial class of software properties, those that characterize the behavior of a system over time. Hence, a number of semi-automated systems have been previously proposed

to exclusively reason about either Computation-Tree Logic (CTL) or Linear Temporal Logic (LTL) in the infinite-state setting. However, corresponding verification techniques are neither scalable nor fully automated, and disallow extensions conducive to the expression of many practical properties, such as CTL^*_{lp} existential stabilization, *i.e.*, "along some future an event occurs infinitely often". Furthermore, until now there have not existed semi-automated, let alone, fully automated systems that allow for the verification of such expressive CTL^*_{lp} properties over infinite-state systems. This leaves a significant gap in the realm of formal verification, as CTL^*_{lp}, the superset of both CTL and LTL, can indeed facilitate the interplay between path-based and state-based reasoning. CTL^*_{lp} thus exclusively allows for the expressiveness of properties involving existential system stabilization and "possibility" properties.

This dissertation proposes methods capable of such a task through building novel CTL model checking techniques capable of supporting more expressive logics such as fair-CTL and CTL^*_{lp}, thus introducing the first known fully automated tool for symbolically proving CTL^*_{lp} properties of (infinite-state) software systems. Additionally, this is the first automated verification system for software systems to consider a linear-past extension to temporal logic in which the past is linear and each moment in time has a unique past. We initially propose a novel high-performance and fully automated CTL verification technique, utilizing counterexample-guided precondition synthesis strategy. This methodology is unique to competing strategies beyond its scalability in that it allows us to implement an internal encoding that admits reasoning about the subtle interplay between the nesting of temporal operators and path quantifiers, conducive to the verification of more expressive logics such as CTL^*_{lp} and fair-CTL. A program transformation introducing prophecy variables to predict the future outcomes, in addition to history variables preserving past outcomes, is then employed to synthesize and quantify preconditions over the transformed program that represent program states that satisfy a CTL^*_{lp} formula. We further demonstrate 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 in the infinite-state setting. Furthermore, we have implemented our approach and report our benchmarks carried out on case studies ranging from smaller programs to demonstrate the expressiveness of CTL^*_{lp} specifications, to larger code bases drawn from device drivers and various industrial examples.

In this chapter, we begin by describing the context under which temporal logic exists within the realm of formal verification. We then provide a detailed analysis of the expressiveness of various temporal logics and their applications to software programs, as this is paramount in discerning the subtle difficulties in verifying each of these logics, and the outstanding research questions associated with them. Finally, we provide the outline and technical contributions of this dissertation.

1.1 Context and Motivation

In [Pnu77], 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.

The context in which the usage of temporal logic first arose for the analysis of programs was towards the trend of unifying the basic notions and approaches of program verification for both sequential and parallel programs. 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 [Lam80], 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. Note that the first formal definition of both safety and liveness properties was not introduced until [AS86]. 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 [Lam80], 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 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.

1.1.1 Expressiveness of Temporal Logics and Their Applications to Programs

In order to discern the subtle difficulty in verifying various temporal logics, 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 given the unparalleled difficulty of verifying linear-time logics over branching-time logics.

Expressiveness of CTL and LTL

CTL and LTL are the most well-studied temporal logics given that they each only require a homogenous approach to reason about computational systems. CTL is a branching-time logic, which requires reasoning about sets of *states*, while LTL is a linear-time logic, which 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. Branching-time logic, the most common being CTL, thus requires reasoning about sets of *states* within a transition system that satisfy a particular temporal formula. Such reasoning is crucial to applications including planning, games, security analysis, disproving, environment synthesis, and many others [GT00, PMT02]. 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. Linear-time logic, the most common being LTL, thus requires reasoning about sets of *paths* that satisfy a formula. In order to demonstrate the further nuances between the various temporal logics to be discussed, we provide an informal description of the temporal operators shared by all temporal logics:

- *Next* or $\mathsf{X}\theta : \theta$ has to hold starting from the next position in a path.
- *Globally* or $\mathsf{G}\theta : \theta$ has to hold starting from all the positions along a path.
- *Eventually* or $\mathsf{F}\theta : \theta$ eventually has to hold.
- *Strong Until* or $\theta_1 \ \mathsf{U} \ \theta_2 : \theta_1$ has to hold starting from all positions until at some position θ_2 starts to hold. θ_2 must be verified in the future.

- *Weak Until* or θ_1 W $\theta_2 : \theta_1$ has to hold starting from all positions until at some position θ_2 starts to hold. Unlike the strong until, θ_2 does not have to be verified and, if such is the case, then θ_1 has to hold forever.

We now consider various CTL properties crucial to the verification of software programs, that being infinite-state systems:

- AG GOOD: Safety, entailing that GOOD will be true for all states on all paths, that is, something bad will never happen.
- AF GOOD: Liveness or termination, entailing that GOOD will eventually become true for some state on every path, that is, something good will eventually happen. Subsequently, AF can also express termination.
- EG GOOD: Nontermination, entailing that there exists an execution that will never halt in which GOOD will forever continue to hold for.
- AGEF RESET: From every state, it is possible to get to the reset state along some path.

We further discuss the applications and expressiveness of LTL in Sections 1.1.2 and 1.1.3 below. 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.

1.1.2 Expressiveness of Fair-CTL

Some of these deficiencies can be mitigated by considering fairness for branching-time logic (fair-CTL). Additionally, despite LTL being more intuitive than CTL, there exists significantly more tools capable of verifying CTL instead of LTL. This is due to the fact that LTL properties are computationally harder to prove than CTL properties in both the finite-state and infinite-state settings. Fair-CTL thus allows us to specify some interaction of linear-time reasoning within branching-time reasoning, but only in specifying fairness assumptions pertaining to a system's environment. These properties are often imperative to verifying the liveness of systems such as Windows kernel APIs that acquire resources and APIs that release resources. Below

are properties which can be expressed in fair-CTL, but neither CTL nor LTL. We write these properties in CTL*, the superset of both CTL and LTL to be discussed in the next section. For brevity, we write Ω for the LTL property $\mathsf{GF}p \Rightarrow \mathsf{GF}q$, where p and q are subsets of program states, constituting our fairness requirement (infinitely often p implies infinitely often q). We note that we define and discuss fairness further in Chapter 4.

The property $\mathsf{E}[\Omega \wedge \mathsf{G}\varphi]$ generalizes fair non-termination, that is, there exists an infinite fair computation all of whose states satisfy the property φ. The property $\mathsf{A}\big[\Omega \Rightarrow \mathsf{G}[\varphi_1 \Rightarrow \mathsf{A}(\Omega \Rightarrow \mathsf{F}\varphi_2)]\big]$ indicates that on every fair path, every φ_1 state is later followed by a φ_2 state. We will later verify this property for a Windows Device Driver, indicating that a lock will always eventually be released in the case that a call to a lock occurs, provided that whenever we continue to call a Windows API repeatedly, it will eventually return a desired value (fairness). Similarly, $\mathsf{A}\big[\Omega \Rightarrow \mathsf{G}[\varphi_1 \Rightarrow \mathsf{A}(\Omega \Rightarrow \mathsf{FE}(\Omega \wedge \mathsf{G}\varphi_2))]\big]$ dictates that on every fair path whenever a φ_1 state is reached, on all possible futures there is a state from which there is a possible fair future and φ_2 is always satisfied. For example, one may wish to verify that there will be a possible active fair continuation of a server, and that it will continue to effectively serve if sockets are successfully opened. Note that later on our definition of fair-CTL considers finite paths. Thus, all path quantifications above range over finite paths as well.

Fairness is also crucial to the verification of concurrent programs, as well-established techniques such as [GCPV09] reduce concurrent liveness verification to a sequential verification task. Thread-modular reductions of concurrent to sequential programs often require a concept of fairness when the resulting sequential proof obligation is a progress property such as wait-freedom, lock-freedom, or obstruction-freedom. Moreover, obstruction freedom cannot be expressed in LTL without additional assumptions. With our technique we can build tools for automatically proving these sequential reductions using fair-CTL model checking.

Unfortunately, fair-CTL still cannot be generalized to model all trace-based properties in LTL. It still does not fully mitigate the significantly reduced expressiveness of CTL and LTL, as recall that they only reason about either states or paths, but not the junction of both. Contrarily, CTL*, a superset of LTL, CTL, and fair-CTL can facilitate the interplay between state-based and path-based reasoning.

1.1.3 Expressiveness of CTL*

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, CTL, or fair-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*. In

the section 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.

First, we briefly give an informal description of CTL* syntax to allow the reader to more intuitively understand the provided examples. CTL* formulae are made up of path quantifiers and temporal operators. Two types of path quantifiers exist, as previously demonstrated: *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.

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. Note 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 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.

1.1.4 Expressiveness of CTL^*_{lp}

In the philosophical context of which they were developed [Kam68, Pri57], 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 [GPSS80]. 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 [KPV12]. Furthermore, past-time connectives are known to make the formulation of specifications more intuitive [LPZ85].

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 [KPV12]. 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 [KPV12]. These are the same known complexities as of CTL^*'s. Unfortunately, the logic CTL^*_{bp} is beyond the scope of this dissertation, thus 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 [KPV12]. 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 [Boz08, KPV12], however we are not aware of implementations of these techniques. Additionally, we are not aware of any tools that consider 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 [BBC+06], where a property requires that drivers mark an I/O request packet as pending (using `IoMarkIrpPending`) before queuing it, that is:

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

Where the past-connective X^{-1} indicates an event that occurs in the *previous* state, while the past-connective $\theta_1 \mathsf{U}^{-1} \theta_2$ indicates that θ_1 has occurred *since* θ_2. 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.

1.2 Objectives and Contributions

We are interested in the static analysis technique of automated model checking, which aims to determine whether or not a formula is true in a given model. That is, given a model of a system, we exhaustively and automatically check whether this model meets a given specification. *More concisely, we seek to automatically verify temporal specifications for the undecidable general class of infinite-state programs supporting both control-sensitive and integer properties.* Fully automated verification of temporal specifications have generally proven to be difficult over the years.

Despite the existence of semi-automated verification tools for CTL and LTL for general integer manipulating programs [BPR13, CGP+07, CK11, CK13], these tools are not conducive for the verification of CTL*. Beyond these tools' stagnation in both high-performance and automation, an additional 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. Moreover, existing techniques for model checking CTL properties cannot even provide support for the verification of the fair-CTL subset, further excluding a large set of branching-time liveness properties necessitating fairness.

Furthermore, no unifying, fully automatic CTL* system, encompassing CTL^*_{lp}, CTL, LTL, and fair-CTL, for verifying the undecidable general class of infinite-state systems has been known. 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 [ES84], and later extended in [EJ99]. Manual proof systems for the verification of temporal logic, first introduced by [EH86, Lam80], have been well-studied. A complete and sound axiomatization of propositional CTL* then followed in [Rey01], which inspired the first sound and relatively complete deductive proof system for the verification of CTL* properties over possibly infinite-state systems [KP05]. Proof rules for verifying CTL* properties of infinite-state systems were implemented in STeP [BBC+00]. 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. In this dissertation, we thus introduce the first known unifying temporal logic verification system capable of automatically verifying CTL, LTL, fair-CTL, CTL*, and CTL^*_{lp} formulae for software, or infinite-state transition systems. We provide preliminaries, notation, and further technical background utilized throughout the dissertation in Chapter 2. Our verification system builds upon a novel scalable and high-performance CTL verification technique introduced in Chapter 3. Our CTL methodology is unique beyond its scalability, in

that it is conducive in allowing us to verify fair-CTL in Chapter 4, and further on CTL* and CTL$^*_{lp}$ in Chapters 5 and 6. We thus define and outline our contributions as follows:

- Chapter 3: We introduce our aforementioned novel CTL verification technique which all other Chapters are to be built on. Given a CTL property φ and a program P, we recursively compute quantifier-free preconditions $\wp$ for all sub-formulae of φ. We then proceed to verify the formulae obtained from φ by replacing the sub-formulae with their corresponding preconditions. Preconditions for a property φ are computed using a counterexample-guided precondition synthesis strategy where several preconditions for each location can be computed simultaneously through the natural decomposition of the counterexample's state space. The main contributions are summarized in the table below:

Chapter 3: CTL Verification of Infinite-State Systems	
Reducing CTL to Safety and Liveness	Algorithm 3
Verifying CTL via Weakest-Precondition	Algorithm 2
Proof of Soundness	Proposition 3.2

  An experimental evaluation in Chapter 7 using examples from the benchmark suites of the competing tools (which are drawn from industrial benchmarks) demonstrates orders-of-magnitude performance improvements in many cases. This work was initially produced and published in [CKP14].

- Chapter 4: As discussed, although CTL can express a system's interaction with inputs and nondeterminism, it cannot model trace-based assumptions about the environment in sequential and concurrent settings (e.g. schedulers) that LTL can express. We thus introduce a novel methodology that reduces Fair-CTL to fairness-free CTL model checking in Chapter 3. Given a CTL property φ and fairness constraint $\Omega = (p, q)$, entailing that if p occurs infinitely often, then so must q, we modify each transition relation by adding prophecy variables [AL91] to encode a partition of fair from unfair paths. We thus proceed to verify our CTL properties on paths that are deemed as fair. This work was initially produced and published in [CKP15a], and the main contributions are summarized in the table below:

Chapter 4: Fair-CTL Verification of Infinite-State Systems	
Reduction of fair-CTL to CTL	Figure 4.1
Proof of Soundness of Figure 4.1	Theorem 4.1
Fair-CTL Model-Checking	Algorithm 8
Proof of Soundness and Relative Completeness	Theorem 4.2

- Chapter 5: We introduce the first fully automated tool for symbolically proving CTL* properties of infinite-state transition systems. We introduce a solution that admits the

arbitrary nesting of state-based reasoning within path-based reasoning, and vice versa. Given our strategic CTL technique, we formulate a strategy allowing 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. A precondition synthesis strategy is used with a program transformation that trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. The main contributions are summarized in the table below:

Chapter 5: CTL* Verification of Infinite-State Systems	
Verifying CTL* via Determinization and Approximation	Algorithm 13
Proof of Soundness	Theorem 5.3
Discussion on Incompleteness of Determinization	Section 5.4

We note that for properties expressible in Fair-CTL, our Fair-CTL technique is relatively complete (to our safety and termination sub-procedures), whereas our CTL* prover is incomplete. We will thus further emphasize that fair-CTL would additionally allow us to employ the automata-theoretic technique for LTL verification [VW94], thus allowing us to prove relatively completely trace-based properties. This work was initially produced and published in [CKP15b].

- Chapter 6: 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. We 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. An invited submission to the Journal of ACM includes the CTL^*_{lp} extension to our CTL* algorithm in [CKP17]. The main contributions are summarized in the table below:

Chapter 6: Extending CTL* to CTL^*_{lp} Verification	
Verifying CTL^*_{lp} via History Variables	Algorithm 15
Proof of Soundness	Theorem 6.3
Case Study	Section 6.3.1

- Chapter 7: We conclude this dissertation with a demonstration of our open-source tool T2, upon which all of our implementations have been made from Chapters 3–6. We demonstrate how T2 has been implemented to support the verification of CTL, LTL, fair-CTL, CTL*, and CTL^*_{lp} directly from C programs. We discuss T2's architecture, its underlying techniques, and conclude with an experimental illustration of its competitiveness and

directions for future extensions. The full demonstration and release of T2 was initially published in [BCI+16].

Chapter 2

Preliminaries and Background

We define all mathematical notation and semantics to henceforth be used in the remaining Chapters of this thesis. This chapter thus aims to guide the reader in understanding the technical contributions to be demonstrated in Chapters 3–6. In Section 2.1 we introduce basic mathematical notation. In Section 2.2 we define software programs and their representation as both control flow graphs and infinite-state transition systems. Given that all contributions in this thesis operate over infinite-state systems, we additionally provide a brief comparison between infinite-state and finite-state verification. Finally, in the remaining sections, we provide the semantics of all temporal logics we are to address and verify in this thesis.

2.1 Basic Notation

2.1.1 Sets

A set is a finite or infinite collection of objects in which order has no significance. Members of a set are known as elements and the notation $a \in A$ denotes that a is an element of a set A. There are various ways of describing members of a set. In the case of a finite set of elements, one often encloses the list of members in curly brackets as follows:

$$A = \{a, b, c\}$$

Similar notation is used for infinite sets, with an addition of ellipses to indicate infiniteness:

$$B = \{1, 2, 3, \ldots\}$$

A more general form of this is set-builder notation which describe sets that are defined by a predicate, rather than explicitly enumerated as previously demonstrated. For example:

$$\{x \in \mathbb{R}.\ x > 0\}$$

is the set of all strictly positive real numbers, which can be written in interval notation as $(0, \infty)$. In our notation, note that the period (".") means "such that". Further below we elaborate on the notation $\mathbb{R}$.

If every member of set A is also a member of set B, then A is said to be a subset of B, written $A \subseteq B$. You can additionally write this statement as $B \supseteq A$, read as B is a superset of A. Other symbols used to operate on sets include intersection $\bigcap$ ("and" or intersection of sets), and union $\bigcup$ ("or" or union of sets). The symbol $\emptyset$ is used to denote the set containing no elements, called the empty set. A Cartesian product is an operation that returns a set from multiple sets. That is, for sets A and B, the Cartesian product $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. Products can be specified using set-builder notation as follows:

$$A \times B = \{\, (a, b) \mid a \in A \ \wedge b \in B \,\}.$$

Finally we define some sets that hold significant mathematical importance, and have acquired specific naming conventions to identify them:

- $\mathbb{N}$, denoting the set of all natural numbers: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
- $\mathbb{Z}$, denoting the set of all integers (whether positive, negative or zero): $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbb{R}$, denoting the set of all real numbers. This set includes all rational numbers, together with all irrational numbers.

2.1.2 Functions

A function f from A to B is a subset of the Cartesian product $A \times B$ in which every element of A is the first component of one and only one ordered pair in the subset. That is, a function is a many-to-one or a one-to-one relation that uniquely associates every $a \in A$ with an object $f(a)$ in B. The set of values of which a function is defined over is called its domain, and the set of values that the function can produce is called its range, this is denoted by $f : \mathit{domain} \to \mathit{range}$. We denote a general function acting upon a particular domain, and returning a specific range using $\mapsto$, *e.g.* $\mathit{domain} \mapsto \mathit{range}$.

2.1.3 Propositional Logic and First-Order Logic

The syntax of propositional logic formulae are defined as follows:

$$\begin{aligned} formula &: \alpha \mid formula\ connective\ formula \mid \neg formula \\ \alpha &: p(term, \ldots, term) \\ connective &: \Rightarrow \mid \wedge \mid \vee \mid \Longleftrightarrow \end{aligned}$$

where p is an n-place predicate symbol (an operator which returns true or false) denoting an atomic condition, and $formula_1 \Rightarrow formula_2$ is defined as $formula_1 \vee \neg formula_2$. Note that finding solutions to propositional logic formulae is an NP-complete problem. We assume that the reader is familiar with propositional logic, and choose to provide more detail with regards to first-order logic(FOL), a generalization of propositional logic.

The basic components of first-order logic are called terms. A term is a variable, constant, or the result of acting on variables and constants by function symbols. The syntax for first-order logic and the set of terms and atoms belonging to it are defined as follows:

$$\begin{aligned} formula &: \alpha \mid formula\ connective\ formula \mid \\ &\quad quantifier\ Variable\ formula \mid \neg formula \\ \alpha &: p(term, \ldots, term) \\ term &: f(term, \ldots, term) \mid Constant \mid Variable \\ connective &: \Rightarrow \mid \wedge \mid \vee \mid \Longleftrightarrow \\ quantifier &: \forall \mid \exists \end{aligned}$$

where f is an n-place function symbol. Consider the formula $\forall x.\ F$ and $\exists x.\ F$, where F is a formula, the $\forall$ operator is the universal quantifier, *i.e., "for all"*, and $\exists$ is the existential quantifier, *i.e., "there exists"*. F denotes the scope of the associated quantifier, and when a variable x occurs in the scope of a quantifier, then it is bound by the closest $\forall x$ or $\exists x$. The variable x is considered free in the formula F if one of its occurrences in F is not bound by any quantifier within the scope of F.

2.1.4 Linear Arithmetic

As previously discussed, the contributions covered in this thesis are in principle applicable to every class of programs (see Section 2.2). However, our implementation (Chapter 7), which depends on existing model-checker technology, constrains our domain of programs to those of which can be directly translated to linear arithmetic. We thus define linear arithmetic as follows [KS08]:

$$\begin{aligned} formula &: formula \wedge formula \mid (formula) \mid atomic \\ atomic &: sum\ op\ sum \\ op &: = \mid \leq \mid < \\ sum &: term \mid sum + term \\ term &: f(term, \ldots, term) \mid Constant \mid Variable \\ Constant &: \ \mathbb{Z} \ \mid \ \mathbb{R} \end{aligned}$$

Note that the operators $\geq$ and $>$ are indeed supported, as they can be expressed by $\leq$ and $<$ if the coefficients are negated, or if the order of the formula is reversed. Similarly, the binary minus operator ("-") can be expressed by $x + -1y$.

Linear arithmetic only considers both the rational and integer number domains. For the former domain, solving for formulae is polynomial, while for the latter, the problem is NP-complete. The underlying technology utilized by our algorithm indeed only supports the integer domain, henceforth when using the term "linear arithmetic" throughout the thesis, we are indeed only referring to the fragment which only considers the integer domain. Additionally, all conditional formulae and assertions considered will henceforth be assumed to fall under the integer domain of linear arithmetic.

The following is an example of a linear arithmetic formula:

$$(3x_1 + 2x_2 \leq 5x_3) \wedge (2x_1 - 2x_2 = 0)$$

Satisfiability and Validity: A formula is satisfiable if there exists at least one interpretation that makes the formula true. A formula is said to be valid if all interpretations make the formula true. Conversely, a formula is unsatisfiable if there exists no interpretations that make the formula true, and invalid if some such interpretations makes the formula false. As an example, a satisfying assignment to the formula above would be $x_1 = 2, x_2 = x_1, x_3 = 2$. Not all solutions are satisfiable, hence this formula is not valid.

There exists a considerable amount of code that is expressible within the linear arithmetic framework, especially code written in C. In Chapter 7, we provide further details on how C

code can indeed be translated to linear arithmetic formulae.

Fourier-Motzkin Elimination: In the context of linear arithmetic, it is often necessary to find the satisfiability of a given conjunction of linear constraints over real or integer variables. One particular strategy to solve for these linear constraints is Fourier-Motzkin elimination, based on variable elimination. That is, the elimination of a set of variables Vars from a formula F of linear inequalities leads to another formula F' without the variables in Vars, such that both systems have the same solutions over the remaining variables. Fourier-Motzkin elimination is a widely used strategy towards deciding the satisfiability of a conjunction of linear constraints over the reals and integers [Mon10].

Now we define *Quantifier Elimination*, that is for every quantified formula F there exists a quantifier-free formula F' in such that F is equivalent to F'. That is, consider a formula F containing the variables x and y, with regards to $\exists x.\ F(x, y)$, Quantifier Elimination would thus result in a formula $F'(y)$ such that for every possible value y, $F'(y)$ is true if and only if $\exists x.\ F(x, y)$ is also true. Recall that a variable is called free in a given formula if at least one of its occurrences is not bound by any quantifier. We assume that F contains both free variables, and variables bound by quantifiers. Under quantifier elimination, it is always possible to eliminate the quantified variables and get an equivalent formula with only free variables. If all variables in F are bound by quantifiers, then for every quantified formula F there exists a quantifier-free formula F' in such that F' is valid if and only if F is valid. That is, every validity checking algorithm can reduce the formula to either true or false. Additionally, if all variables are eliminated from a system of linear inequalities, then the resulting system is one of exclusive constant inequalities. It thus becomes trivial to decide whether the resulting system is satisfiable or not. One can thus eliminate all variables within a system of linear inequalities to determine whether it has a solution(s) or not. For the purpose of our contributions, we note that quantifier elimination is indeed closed under linear-integer arithmetic, but not non-linear integer arithmetic.

In general, we require a quantifier elimination algorithm which transforms a quantified formula into an equivalent formula without quantifiers. Due to its nature, Fourier-Motzkin elimination thus consequentially performs quantifier elimination given its elimination of variable in Vars to which an existential quantifier refers. Thus let us consider the Fourier-Motzkin elimination method, where in order to eliminate a variable x_n from a formula with variables $x_1, \ldots, x_n$, for every two constraints of the form

$$\sum_{i=1}^{n-1} a'_i \cdot x_i < x_n < \sum_{i=1}^{n-1} a_i \cdot x_i$$

Where for $i \in \{1, \ldots, n-1\}$, a_i and a'_i are constants, we generate a new constraint

$$\sum_{i=1}^{n-1} a'_i \cdot x_i < \sum_{i=1}^{n-1} a_i \cdot x_i$$

After generating all such constraints for x_n, we remove all constraints that involve x_n from the formula. Fourier-Motzkin elimination is thus equivalent to performing existential quantifier elimination, one variable at a time. A universal quantification $\forall x_n.G(x)$ is equivalent to $\neg\exists x_n.\neg G(x)$.

2.2 Programs and Transition Systems

In this section, we demonstrate how programs are considered and formalized in the context of our verification framework.

2.2.1 Control Flow Graphs

As is standard [MP95], we treat programs as control flow graphs. A control flow graph (CFG) is a directed graph representing all paths that might be traversed through a program during its execution. In a CFG, each node in the graph represents a basic block of code without any jumps or jump targets; jump targets start a block, and jumps end a block. That is, a program is a triple $P = (\mathcal{L}, E, \mathsf{Vars})$, where

- $\mathcal{L}$: A finite set of vertices containing one location for each basic block in the program.
- Vars : A set of variables(identifiers) ranging over the domain Vals, that is all permissible values allowed by the program.
- E: A set of labeled edges/transitions where each edge (ℓ, ρ, ℓ') in E, and $\ell, \ell' \in \mathcal{L}$. The assertion ρ is a linear arithmetic expression over Vars and Vars' (a primed copy of Vars, where constants range over Vals), specifying possible transitions in the program.

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 $x' = 5$. Recall that we assume our conditions use linear constraints over variables in the program.

```
int nondet();
int main() {
    int y = 0;
    int k = nondet();
    int x = nondet();
    if (k > 0)
        while (x > 0)
            x = x - k;
    return 0; }
```

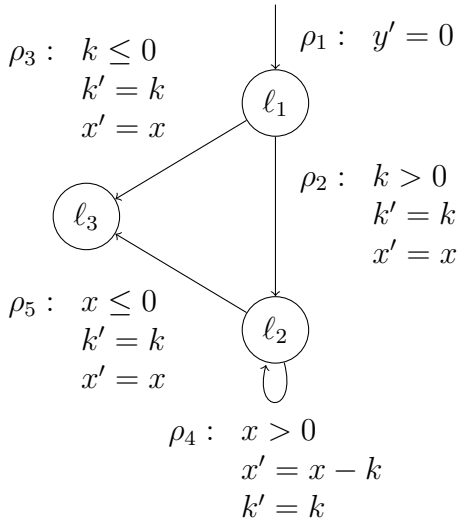


Figure 2.1: A C example program (left) with its corresponding CFG representation (right).

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 $(\ell, \rho, \ell') \in E$ 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. When demonstrating figures, we omit ℓ_I and merely display the edges to locations with incoming transitions from ℓ_I.

Nondeterminism: In a program, there may exist various alternatives for a program flow from a certain branching location. However, unlike an if-then statement, the method of choice between these alternatives is not directly specified by the programmer; the program at run time indeed decides between theses alternatives, *e.g.*, a loop condition that may depend upon user-input. That is, the output cannot be predicted at a branching location. The program thus offers alternate control flows, but must ultimately decide between them upon execution. CFGs naturally represent nondeterminism when two valid edges stemming from the same program location exist. With regards to a program's CFG P, nondeterministic assignments to variables are denoted by the absence of an update to said primed program variables. We demonstrate all concepts described above with an example below.

Consider Fig. 2.1, where a C program is represented with its corresponding CFG. In Chapter 7 we fully describe our procedure to translate C programs to CFGs, however, for the purpose of clarity, we provide an example to illustrate our concepts further. Note that since we statically verify programs, we choose to symbolically represent nondeterministic run-time behavior as *nondeterministic* decisions within a C program denoted by the function `nondet()`. In this program the integer variable y is initialized to 0, while x and k are given nondeterministic assignments. Note how in transition ρ_1 of the CFG the variables x and k are absent, indicating their nondeterministic assignments. The nondeterministic value of k thus determines if the "if" branch is indeed to be taken in the case that $k > 0$. Note the assignments $k' = k$ and $x' = x$ in

ρ_2 of the CFG, indicating that the values of these variables are remaining the same. If indeed the branching condition $k > 0$ is taken, we arrive at a loop with a condition relying on the yet another nondeterministic variable x. If the loop is entered, indicating that indeed $x > 0$, then x is decremented by the value of k until $x \leq 0$. Finally, note that ℓ_3 in the CFG is a *final* location, that is, a location without any outgoing edges. Recall that each node or location in the graph represents a basic block of code without any jumps or jump targets, thus we indeed did not specify a new node for each C program location. For example, ρ_4 contains both the program commands $x > 0$ and $x' = x - k$.

2.2.2 Infinite-State Transition Systems

Ultimately, P gives rise to an infinite-state transition system. Informally, a transition system is used to describe the potential behavior of discrete systems. It consists of states and transitions between states. If the reader is familiar with finite-state automata, transition systems differ from automata in several ways:

- The set of states is not necessarily finite, or even countable.
- The set of transitions is not necessarily finite, or even countable, and an infinite number of transitions could stem from a single state.
- There exists no singular initial state or final state.

Given that we constrain our programs over the linear arithmetic domain, our set of states are indeed countable, but infinite. We thus define a transition system $T = (S, S_0, R, L)$, where

- S is the set of program states of the form $S = (\mathcal{L} - \{\ell_I\}) \times (\mathsf{Vars} \to \mathsf{Vals})$.
- S_0 is the set of initial program states. We provide a more detailed definition of initial states in "Transitions" below.
- R is the set of program transitions of the form $R \subseteq S \times S$.
- $L : S \to 2^{AP}$ a labeling function associating a set of atomic predicates with every state $s \in S$. We note that transition systems arise from programs, where a program state is characterized by a program location and a valuation over the program variables (as discussed further below). Predicates expressed in linear arithmetic over program variables thus serve the role of atomic propositions, as their truth value is utilized to label the intended program states.

States

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 a variable valuation. 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. Recall that for a program's CFG P that the a transitions' assertion ρ indeed does not include locations. Thus we note that logical assertions can represent sets of program states by relating variables to their values through linear arithmetic expressions, in addition to dictating updates to program variables. Furthermore, assertions can treat locations as boolean values, and could refer to locations and hence, by extension, assertions can also represent sets of transitions.

With regards to locations, a primed location indicates that it is the target of a transition. Similarly, with regards to states, we may use a primed state to denote the next target state within a transition system.

Transitions

To define transitions, we first consider the relation between two states. We define a valuation (f_1, f_2) as 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 $(f_1, f_2)(v') = f_2(v)$. Now consider a program's transitions R, a program can transition from a state (ℓ, f_1) to state (ℓ', f_2) if there exists a transition $(\ell, \rho, \ell') \in E$ such that $(f_1, f_2) \models \rho$, where "$\models$" denotes semantic implication. For succinctness, we denote this as $(s = (\ell, f_1), s' = (\ell', f_2)) \in R$. An initial state $s_0 = (\ell, f)$ is considered initial if and only if there is a transition (ℓ_I, ρ, ℓ) such that $(f_{-1}, f) \models \rho$, where f_{-1} is some arbitrary valuation. Note 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 $f{\Downarrow_V}$.

Paths

A *trace* or a *path* π in P is either a finite or infinite sequence of states such that for a finite path $\pi = \langle s_0 = (\ell_0, f_0)\rangle, \langle s_1 = (\ell_1, f_1)\rangle, \ldots, \langle s_n = (\ell_n, f_n)\rangle$ where $\forall\ 0 \leq i \leq n.\ (s_i, s_{i+1}) \in R$ and $\forall\ 0 \leq i \leq n+1.\ (s_n, s_{n+1}) \notin R$, or for an infinite path $\pi = \langle s_0 = (\ell_0, f_0)\rangle, \langle s_1 = (\ell_1, f_1)\rangle, \ldots$, where $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$. That is, for every $i \geq 0$, there exists some $(\ell, \rho, \ell') \in E$ where $(f_i, f_{i+1}) \models \rho$. We denote the above path π as an (ℓ, f)-path. The set of infinite traces starting at $s \in S$, denoted by $\Pi_\infty(s)$, is the set of sequences $(s_0, s_1, \ldots)$ such that $s_0 = s$ and $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$. The set of finite traces starting at $s \in S$, denoted by $\Pi_f(s)$, is the set of sequences $(s_0, s_1, \ldots, s_j)$ such that $s_0 = s$, $j \geq 0$, $\forall i < j.\ (s_i, s_{i+1}) \in R$, and $\forall s \in S.\ (s_j, s) \notin R$.

Finally, the set of maximal traces starting at s, denoted by $\Pi_m(s)$, is the set $\Pi_\infty(s) \cup \Pi_f(s)$. For a path π, we denote the length of said path by $|\pi|$, which is ω in case that π is infinite. 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 for $\pi = \langle(\ell_0, f_0)\rangle, \langle(\ell_1, f_1)\rangle, \ldots, (\ell_0, f_0)$ is an initial state. A state s_i is reachable if (ℓ, f_i) appears in some computation. Similarly, a location ℓ is reachable if there exists some state s_i such that (ℓ, f_i) appears in some computation The restriction of states of the form (ℓ, f) and paths in the program is denoted by $\pi\Downarrow_V$.

2.3 CTL* Semantics

Given that CTL and Fair-CTL are both subsets of CTL*, we begin by formally defining full computation tree logic [Lam80, EH86], followed by said subsets. 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. Atomic predicates (ranged over by α) are expressed in some underlying theory, *e.g.* linear arithmetic, over variables and constants. State formulae (φ) and path formulae (ψ) are co-defined as follows:

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

Traditionally, CTL* is defined only over infinite paths, thus note that our semantics alternatively ranges over maximal paths, as defined in 2.2.2. 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 $\forall \pi = (s, ...) \in \Pi_m(s).\ P, \pi \models \psi$
- If $\varphi = \mathsf{E}\psi$, then $P, s \models \mathsf{E}\psi$ iff $\exists \pi = (s, ...) \in \Pi_m(s).\ P, \pi \models \psi$

Path formulae are interpreted over paths. For a program P and a CTL* path formula ψ, we say that ψ holds on a path $\pi = (s_0, s_1, \ldots) \in \Pi_m(s_0)$ in P for location $i \in [0, |\pi|)$, 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 $\exists j \in [i, |\pi|).\ P, \pi, j \models \psi_1$
- If $\psi = \mathsf{G}\psi_1$, then $P, \pi, i \models \mathsf{G}\psi_1$ iff $\forall j \in [i, |\pi|).\ P, \pi, j \models \psi_1$
- If $\psi = \psi_1 \mathsf{W} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{W} \psi_2$ iff either $\exists k \in [i, |\pi|).\ P, \pi, k \models \psi_2$ and $\forall \in [i, k).\ P, \pi, j \models \psi_1$ or $\forall j \in [i, |\pi|).\ P, \pi, j \models \psi_1$
- If $\psi = \psi_1 \mathsf{U} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{U} \psi_2$ iff $\exists k \in [i, |\pi|).\ P, \pi, k \models \psi_2$ and $\forall j \in [i, k).\ P, \pi, j \models \psi_1$

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.

To give some intuition to the semantics of CTL^*, $P, s \models \mathsf{EGF}\alpha$ asserts that for a program P, there exists some execution π starting from every initial state s such that α occurs along π infinitely often. The formula $\mathsf{EF}\varphi$ is an example of a "state" formula where $P, s \models \mathsf{EF}\varphi$ if there is a state s' reachable from s where φ holds. The formula $\mathsf{FG}\alpha$ is an example of a"Path" formula where $P, \pi \models \mathsf{FG}\alpha$ if on a path π, predicate α will eventually become true and stay true.

Although in our formalization negation can be applied only to atomic predicates, we use negation in higher level by using the de-Morgan rules as well as the equalities, where θ indicates either φ or ψ:

$$\begin{aligned} \neg\mathsf{E}\theta &\equiv \mathsf{A}\neg\theta \\ \neg\mathsf{F}\theta &\equiv \mathsf{G}\neg\theta \\ \neg[\theta \mathsf{U} \theta'] &\equiv [\neg\theta' \mathsf{W} (\neg\theta' \wedge \neg\theta)] \end{aligned}$$

2.3.1 CTL and LTL Semantics

The branching-time logic CTL [CE81] 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. CTL is thus of the form:

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

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

To give intuition behind the semantics of CTL, here $P, s \models \mathsf{AF}\varphi$ asserts that in program P and in all possible executions starting from s the property φ will eventually hold in some future state reachable from s, whereas $P, s \models \mathsf{EF}\varphi$ asserts that there is a state in which φ holds

and that it can be reached from s. The formula $\mathsf{AG}\varphi$ asserts that φ must hold throughout all possible executions, while $\mathsf{EG}\varphi$ asserts that there exists an execution such that φ would be true throughout. $\mathsf{A}\varphi_1\mathsf{W}\varphi_2$ asserts that for all executions, φ_1 has to hold until φ_2 holds, signifying that φ_2 does not necessarily have to hold as long as φ_1 holds. Contrarily, $\mathsf{E}\varphi_1\mathsf{U}\varphi_2$ asserts that there exists an execution in which φ_1 has to hold *at least until* at some position φ_2 holds. We denote a formula as an ACTL formula if the only path quantifiers used are universal, i.e., AX, AW, AF, AU, or AG.

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. The LTL subset is thus of the form:

$$\psi \quad ::= \quad \alpha \quad | \quad \psi \wedge \psi \quad | \quad \psi \vee \psi \quad | \quad \mathsf{G}\psi \quad | \quad \mathsf{F}\psi \quad | \quad \mathsf{X}\psi \quad | \quad [\psi\mathsf{W}\psi] \quad | \quad [\psi\mathsf{U}\psi]$$

Note that in this thesis we do not cover exclusive LTL verification methodologies as we do for CTL. Our CTL* techniques solely rely on CTL verification methods, and given that CTL* is a superset of LTL, our CTL* verification algorithm is thus indeed sufficient to verify LTL properties.

Counterexamples: In our setting counterexamples are produced by an underlying safety prover (discussed in more detail in Chapter 7). Generally speaking, a path that does not satisfy an LTL property is called a counterexample. Similarly, a path that contains a state which does not satisfy a CTL property is also a counterxample. Due to the recursive nature of our procedure discussed in Chapter 3, it is only necessary to handle counterexamples to CTL formulae with a single path quantifier. For example, $\mathsf{A}\varphi$, where φ is a path formula that includes no nesting of additional operators, or $\alpha_1 \vee \alpha_2$, where α_1 and α_2 are assertions. A counterexample for an atomic predicate α is a state in which α does not hold. A counterexample for a conjunction $\varphi_1 \wedge \varphi_2$ is a state where either φ_1 or φ_2 does not hold. A counter example for disjunction $\varphi_1 \vee \varphi_2$ is a state where both sub-formulae do not hold. A counterexample to an $\mathsf{AG}\varphi$ property is a path to a place where φ does not hold. A counterexample to an $\mathsf{AF}\varphi$ property is a "lasso": a stem path to a particular program location, then a (not necessarily simple) cycle which returns to the same program location, and the property φ does not hold along the stem and the cycle. Finally, a counterexample to $\mathsf{A}[\varphi_1\mathsf{W}\varphi_2]$ is a path to a place where there is a sub-counterexample to φ_1 as well as one to φ_2. A counterexample to $\mathsf{E}[\varphi_1\mathsf{U}\varphi_2]$ can be of the same form as that of $\mathsf{A}[\varphi_1\mathsf{W}\varphi_2]$, as well as one where φ_1 holds while φ_2 does not hold anywhere along the path.

Fair-CTL Semantics

To support fair-CTL, we modify our transition system P to include a fairness condition $\Omega = (p, q)$, where $p, q \subseteq S$. When fairness is part of the transition system we denote it as $P = (S, S_0, R, L, \Omega)$. That is, p and q are assertions, as recall that assertions can indeed correspond to sets of program states. We still include Ω as a separate component in transformations and algorithms for emphasis. An infinite path π is unfair under Ω if states from p occur infinitely often along π but states from q occur finitely often. Otherwise, π is fair. We elaborate on the concept of fairness further in Chapter 4. Equivalently, π is fair if it satisfies a path formula $\pi \models (\mathsf{GF}p \Rightarrow \mathsf{GF}q)$. For a transition system $P = (S, S_0, R, L, \Omega)$, an infinite path π, we denote $P, \pi \models \Omega$ if π is fair [EL86].

2.3.2 CTL^*_{lp} and CTL_{lp} Semantics

Below, we define the fragment of CTL^*_{lp} that we support in this thesis, where CTL^*_{lp} stands for CTL^* *with* linear-past. 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. Additionally, just as how CTL is a subset of CTL^*, CTL_{lp} is a subset of our fragment of CTL^*_{lp}. Note that when referring to CTL_{lp}, we are indeed referring to *full* CTL_{lp}, despite it being a subset of a fragment of CTL^*_{lp}. 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:

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

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 $\sigma = s_0 \dots s_n$ 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 $P, \sigma_m \models \mathsf{A}\psi$ iff $\forall \pi \in \Pi_m(\sigma).\ P, \pi, |\sigma| - 1 \models \psi$
- If $\varphi = \mathsf{E}\psi$, then $P, \sigma_m \models \mathsf{E}\psi$ iff $\exists \pi \in \Pi_m(\sigma).\ P, \pi, |\sigma| - 1 \models \psi$

Path formulae (τ and ψ) are interpreted over computations. Assume the inclusion of future-connectives as specified in Section 2.3. For a program P and a CTL^* path formula ψ, we say that ψ holds on a computation $\pi = (s_0, s_1, \dots) \in \Pi_m(s_0)$ in P for location $i \in [0, |\pi|)$, 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 $\forall k < j \leq i.\ P, \pi, j \models \psi_1$

As before, 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.

As with CTL, CTL_{lp} is a restricted subset of CTL^*_{lp} in which temporal operators cannot be nested within one another. CTL_{lp} is thus of the form:

$$
\begin{aligned}
\varphi ::= & \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
& \mid \mathsf{AG}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi] \mid \mathsf{A}[\varphi \mathsf{U} \varphi] \mid \mathsf{AG}^{-1}\varphi \mid \mathsf{AF}^{-1}\varphi \mid \mathsf{A}[\varphi \mathsf{W}^{-1} \varphi] \mid \mathsf{A}[\varphi \mathsf{U}^{-1} \varphi] \\
& \mid \mathsf{EF}\varphi \mid \mathsf{EG}\varphi \mid \mathsf{E}[\varphi \mathsf{U} \varphi] \mid \mathsf{E}[\varphi \mathsf{W} \varphi] \mid \mathsf{EF}^{-1}\varphi \mid \mathsf{EG}^{-1}\varphi \mid \mathsf{E}[\varphi \mathsf{U}^{-1} \varphi] \mid \mathsf{E}[\varphi \mathsf{W}^{-1} \varphi]
\end{aligned}
$$

2.4 Further Terminology

2.4.1 Ranking functions

Recall that temporal logic subsumes liveness properties, ensuring progress properties such as termination, deadlock-freedom, and freedom of starvation. It is thus required in our analysis of these logics to find ranking functions that demonstrate progress towards a bound during each transition of a system. Although a thorough discussion on finding ranking functions is beyond the scope of this thesis, our techniques indeed rely on an existing model-checker possessing a ranking function generator [CSZ13, BCF13]. For a state space S, we thus define a *ranking function* γ as a total map from S to a well-ordered set with ordering relation $\prec$. Ranking functions are used to measure progress on a potentially terminating process in a program. A linear ranking function is of the form $r_i x_i + r_{i+1} x_{i+1} + \ldots + r_n x_n$ where x_i denotes a program variable. Given the our programs are constrained to the domain of linear arithmetic, our linear ranking functions thus range over the well-ordered set of the natural numbers with the relation $\leq$. We thus substitute the ordering relation $\prec$ with $\leq$ Given a ranking function γ, we define its ranking relation as

$$T_\gamma = \{s, s'.\gamma(s) > \gamma(s') \wedge \gamma(s) \geq 0\}$$

That is, all pairs of states over which γ decreases and is bound by 0. Transitions in the ranking relation contribute to the progress of γ. Similarly, we define a ranking function's un-affecting relation as

$$U_\gamma = \{s, s'.\gamma(s) \geq \gamma(s')\}$$

That is, all pairs of states over which γ does not increase. Note that transitions in U_γ do not hinder the progress of γ. For a condition ρ over the state space S, we say that a ranking function γ is unaffected by ρ if $\rho \subseteq U_\gamma$. A relation $R \subseteq S \times S$ is considered *well-founded* if and only if there exists a ranking function γ such that $\forall (s, s') \in R.\ \gamma(s') < \gamma(s)$. We denote a finite set of ranking functions (or *measures*) as $\mathcal{M}$. Note that the existence of a non-empty set of ranking functions for a relation R is equivalent to containment of R^+ within a finite union of well-founded relations, where R^+ is the transitive closure of the transition relation R [PR04b]. That is, a set of ranking functions $\{\gamma_1, \ldots, \gamma_n\}$ denotes the disjunctively well-founded relation $\{(s, s').\gamma_1(s') < \gamma_1(s) \vee \ldots \vee \gamma_n(s') < \gamma_n(s)\}$.

2.4.2 Recurrence Sets

A transition system $T = (S, S_0, R, L)$ is considered non-terminating if and only if there exists an infinite transition sequence $\pi = \langle s_0 = (\ell_0, f_0)\rangle, \langle s_1 = (\ell_1, f_1)\rangle, \ldots$. Non-termination is char-

acterized over a transition relation R by the existence of a *recurrence set*, that is, a non-empty set of states $\mathcal{N}$ such that for each $s \in \mathcal{N}$ there exists a transition to some $s' \in \mathcal{N}$ [GHM+08]. A transition system T has a recurrence set of states $\mathcal{N}$ if and only if

$$\exists s.\ s \in \mathcal{N} \wedge s \in S_0$$

$$\forall s \exists s'.\ s \in \mathcal{N} \Rightarrow (s, s') \in R \wedge s' \in \mathcal{N}$$

A transition system T is thus only considered non-terminating if and only if it has a recurring set of states.

2.4.3 Calculating pre-images

In Chapter 4, we describe a symbolic model checking procedure for CTL verification of infinite-state systems that synthesizes preconditions asserting the satisfaction of CTL sub-formulae. We thus define *pre-images* over paths for the purpose of utilization by our CTL technique. Note that techniques in both Chapters 5 and 6 additionally depend on the aforementioned CTL procedure. Let $\pi = \langle s_0 = (\ell_0, f_0)\rangle, \langle s_1 = (\ell_1, f_1)\rangle, \ldots, \langle s_n = (\ell_n, f_n)\rangle$ be a finite path. We compute a pre-image for every possible suffix of π. We denote $pre_{n+1} = S$ and $pre_i = pre(s_i = (\ell_i, f_i), \ldots, s_n = (\ell_n, f_n))$ as the set of states such that for $s_i, s_{i+1} \in pre_{i+1}$ then $\forall\ 0 \leq i \leq n.\ (s_i, s_{i+1}) \in R$, that is, there exists some $(\ell, \rho, \ell') \in E$ where $(f_i, f_{i+1}) \models \rho$. Generally speaking, given an assertion α (in terms of Vars) representing pre_{i+1}, and an assertion ρ, we must compute an assertion representing pre_i. We thus consider $\exists\ \mathsf{Vars}'.\ \alpha' \wedge \rho$ where we practically utilize Fourier-Motzkin for performing quantifier elimination on such formulae.

2.4.4 Utilizing Strongly Connected Subgraphs

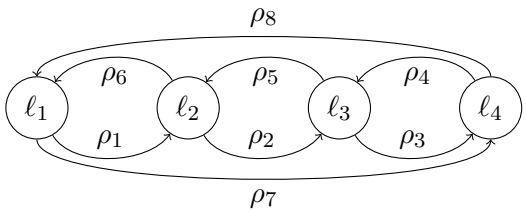


Figure 2.2: A control-flow graph with multiple possible partitions of SCSs.

We provide some notation regarding strongly-connected subgraphs 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., [CSRL01]). 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 one is contained in the other.

For example, consider the control-flow graph in Figure 2.2. One possible segmentation 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 segmentation is $\{\ell_1, \ell_4\}$, $\{\ell_2, \ell_3\}$, and $\{\ell_1, \ldots, \ell_4\}$. According to this segment, the minimal SCS containing ℓ_3 is $\{\ell_2, \ell_3\}$. Given the numerous ways in which an SCS can be segmented, one might anticipate that a particular choice may impact the behaviours and the results of the algorithms utilizing this strategy. Although this may be the case theoretically, we note that such is not the case in practice. Due to the nature of the procedural programs that we analyze, only one choice will ever be identified.

Chapter 3

Faster Temporal Reasoning for Infinite-State Programs

In this chapter, we describe a symbolic model checking procedure for the CTL verification of infinite-state programs. The introduced procedure exploits the natural decomposition of the state space given by the control flow graph in combination with the nesting of temporal operators to optimize reasoning performed during symbolic model checking.

3.1 Introduction

As previously emphasized, state-based temporal logics such as CTL allow us to reason about a system's interaction with inputs and nondeterminism in a way that path-based temporal logics such as LTL do not. Such reasoning is crucial to applications including planning, games, security analysis, disproving, environment synthesis, and many others [GT00, PMT02]. Unfortunately, the search for scalable and high-performance temporal-logic proof tools for infinite-state programs remains an open problem, as a shortcoming of existing tools is that performance is hindered by redundant reasoning performed in the presence of nested temporal operators. For example, tools supporting the state-based temporal logic CTL [CCG$^+$02, CCG$^+$05, CE81, CK13, BPR13] invariably recurse over the structure of an input property and redundantly reason over the same sets of system states with regards to each sub-formula. Thus in this chapter, we propose a symbolic CTL model-checker which gains orders-of-magnitude performance speed improvement over the verification of infinite-state programs. In later chapters, we additionally demonstrate how our CTL methodology can used as the basis for proving higher-performance logics such as CTL*.

We leverage the well-known sound and relatively complete deductive proof system for the

verification of CTL* proposed by [KP05] towards the automated CTL verification over infinite-state systems. That is, a CTL formula is recursively decomposed into its constituent sub-formulae, with each temporal sub-formula then being subsequently replaced by a precondition. The key insight to our approach is the exploitation of the natural decomposition of the state space given by the control flow graph. That is, using a counterexample-guided precondition synthesis strategy, we compute program location specific preconditions. Our model checker drastically improves performance by reducing the amount of redundant and irrelevant reasoning performed through information sharing extracted from reachability analysis. That is, several preconditions for each program location can be computed simultaneously. Take for example the fact that the set of states respecting a property such as $\mathsf{EF}y < z$ *before* a program command is very often the same or nearly the same as the set of states respecting $\mathsf{EF}y < z$ *after* the command. In comparison to existing tools (*e.g.* [CK13, BPR13]), we reduce the amount of reasoning performed as part of the procedure. We can infer whether a command is likely to affect the truth of $\mathsf{EF}y < z$. So, sequential locality implies that the precondition of a location is easily computed if the preconditions of its successors are known.

This approach gives way to compositional reasoning. For instance, given a program and a desired property, we can, in parallel, synthesize preconditions, desired environments, and plans of individual procedures of a program with the goal of composing the found preconditions into a proof of the whole program. The advantage to this approach is that the program verification tools never have to examine the program as a whole, but instead find a modular proof using compositional reasoning. Recent success in this style of reasoning can be found in areas such as proving correctness of non-blocking algorithms [GCPV09], and the analysis of biological models [CFKP11]. We also suggest a new method of treating existential path quantification in the infinite-state setting. Existential formulae are handled by considering their universal dual, allowing counterexamples of said duals to serve as a witness asserting the satisfaction of the existential CTL formula.

An experimental evaluation using examples from the benchmark suites of the competing tools (which are drawn from industrial benchmarks) demonstrates orders-of-magnitude performance improvements in many cases. This evaluation is discussed in Chapter 7.

3.1.1 Related work

In this work we are aiming to prove CTL properties with *all* syntactically valid nested combinations of existential and universal path quantifiers of programs that are expressible in the linear arithmetic framework. A number of CTL model checking tools for programs do not meet these criteria. For example, SMV (and in general BDD based tools) are restricted to finite-state programs [CCG+02]. Song & Touili [ST12] perform a coarse one-time abstraction that takes

programs and produces pushdown automata, however the abstraction produced is imprecise and leads to significant information loss. Gurfinkel *et al.* [GWC06] do not reliably support all possible CTL syntactic formulations of nested universal and existential path quantifiers.

Tools with a more relevant feature set to our setting include Cook & Koskinen [CK13] and Beyene *et al.* [BPR13]. Cook & Koskinen utilize an incremental reduction to other existing program analysis techniques, as we do. However, their symbolic determinization is based on the counterexample-guided refinement of generated tree counterexamples, or counterexamples with branching paths. That is, [CK13] 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 leads to an exponential state explosion, instigated by the recursive nesting of the verification of each CTL sub-formula within the program state-space. Contrary to our procedure, such a method causes precondition generation for a property's sub-formulae to be no longer possible. Finally, our novel approach to the treatment of existential path quantification based on dualization contrasts to that of Cook & Koskinen, which attempts to find a non-trivial restriction on the state-space such that AF can be used to reason about EF, or AG can be used to reason about EG. Note that this technique is only applicable to systems with *one* initial state, while our procedure is suitable for systems with every number of initial states.

Beyene *et al.* [BPR13] implement the the Kesten and Pnueli [KP05] deductive proof system using a reduction to Horn-clause reasoning. Our approach also contrasts to the tool of Beyene *et al.* [BPR13], as their tool requires a manual instantiation of the structure of assertions, characterizing CTL formulae, that are to be found by their tool. Neither Cook & Koskinen nor Beyene *et al.* make use of the locality in programs as we do. Effectively, these tools carry out unnecessary computation in their analysis.

We note that our CTL verification procedure leverages recent techniques for proving safety, termination, and nontermination of programs [GHM$^+$08, CSZ13, BCF13, McM06] to synthesize preconditions asserting the satisfaction of CTL sub-formulae of an input property. Counterexample-guided refinement is a very common technique utilized in the model-checking of both finite and infinite-state systems. However, we note that techniques in [CGJ$^+$00, HHP13] differ from our contribution as they seek to find the right abstraction of a *program* for a given correctness property. Contrarily, we seek to refine a *precondition*, entailing the satisfaction of a temporal formula.

3.1.2 Limitations

We do not support programs with heap, nor do we support recursion or concurrency. The heap-based programs we consider during our experimental evaluation have been abstracted using the

over-approximation from the technique of Magill *et al.* [MBCC07]. Note that this abstraction can lead to unsoundness when we use the existential subset of CTL. Our comparison to existing tools remains fair, as each of the previous tools uses the same abstraction. Effective techniques for proving temporal properties of programs with heap remains an open research question.

As our technique heavily relies on calculating pre-images, it is important that fragments of the underlying program logic are closed under pre-images, *e.g.,* integer linear arithmetic, a fragment of integer arithmetic. Thus we note that weakest preconditions are indeed closed under integer linear arithmetic, but not non-linear integer arithmetic. In general, our procedure is not complete as we use a series of incomplete subroutines.

3.2 Illustration and Example

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

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

Consider a universal CTL formula. Initially, we approximate its precondition as TRUE. We then search for counterexamples from every possible reachable program location. Produced counterexamples will result in the strengthening of the precondition through adding the negation of the pre-image of the discovered counterexample, as discussed in 2.4. We use the control flow graph of a counterexample to simultaneously synthesize preconditions of multiple locations. That is, a counterexample that consists of multiple program locations can be utilized to update the precondition of each contained program location. This is done by iterating along the counterexample path, and for each suffix computing a pre-image from a program location onwards. Each counterexample found further strengthens a precondition, we thus eliminate said counterexample and search for other proof failures for the given CTL property. Eventually, the precondition will imply the correctness of the sub-formula when no further counterexamples are returned.

Existential sub-formulae are handled by considering their universal dual. We thus seek a set of

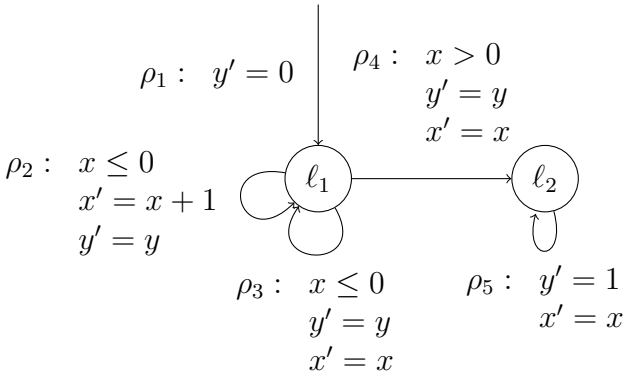


Figure 3.1: The control-flow graph of an example program for which we wish to prove the CTL property AGEF $y = 1$.

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

3.2.1 Example

Consider the program in Fig. 3.1 and the property $\varphi \equiv \mathsf{AGEF}\ y = 1$, which states that for all states, it is always possible that eventually $y = 1$. The approach followed by nearly all tools supporting CTL would be to find, in this instance, a set of states $\wp$ such that $\mathsf{AG}\wp$ holds, and such that $\wp \models \mathsf{EF}\ y = 1$ holds. In this chapter, we suggest a strategy based on precondition synthesis.

Consider the sub-formula $\varphi_{\mathsf{EF}} \equiv \mathsf{EF}\ y = 1$. For the predicate $y = 1$, for every program location ℓ_i we have $\wp\langle \ell_i, y = 1\rangle \triangleq y = 1$. In order to support existential path formulae (*e.g.* EF), we use a strategy that synthesizes preconditions through the negation of the existential property (*i.e.* a universal dual). That is, we prove $\wp \models \mathsf{EF}\ y = 1$ via negation: we attempt to prove $\wp \not\models \mathsf{AG}\ y \neq 1$. We now attempt to prove that $\wp \not\models \mathsf{AG}\ y \neq 1$ given that AG is EF's universal dual. We start with $\wp\langle \varphi_{\mathsf{EF}}\rangle \triangleq$ FALSE as only failures to proving $\mathsf{AG}\ y \neq 1$ can necessitate that there exists a witness such that $\mathsf{EF}\ y = 1$. Failures to the proof attempt will result in refinements to $\wp$ through the iterative calculation of the pre-image of each discovered counterexample. Recall that we are interested in counterexamples starting from all program locations:

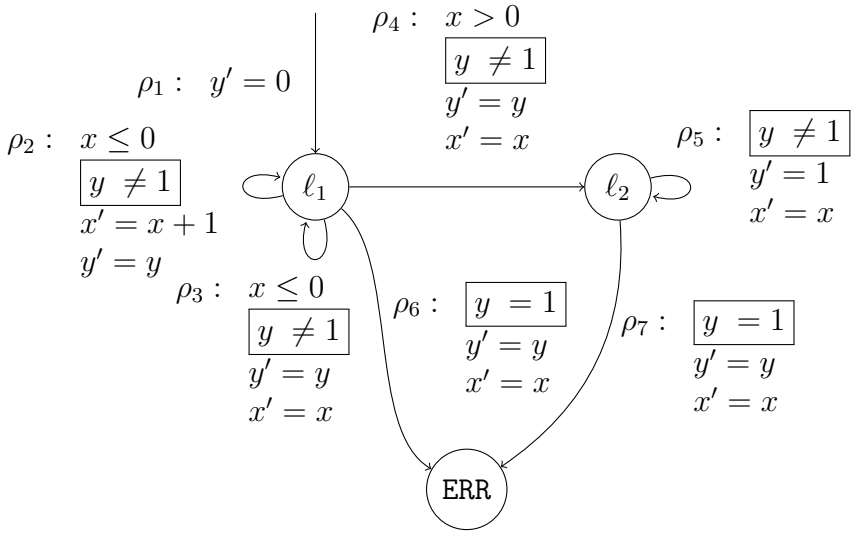


Figure 3.2: The transformation of the program from Fig. 3.1 for the property $\mathsf{EF}\ y = 1$ using its dual $\mathsf{AG}\ y \neq 1$.

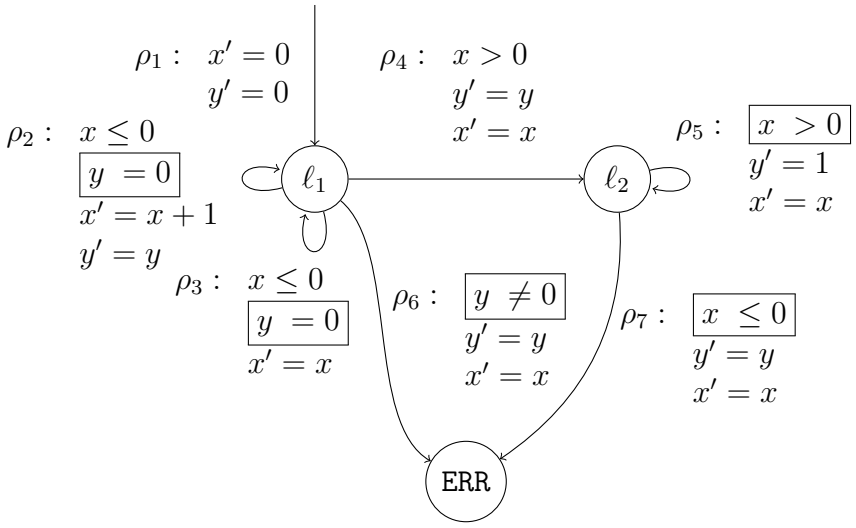


Figure 3.3: The transformation of the program from Fig. 3.1 for the sub-property $\mathsf{AGEF}\ y = 1$ to be utilized in the verification algorithm. The nested property $\mathsf{EF}\ y = 1$ is substituted with its precondition resulting in a transformation for $\mathsf{AG}\ ((\mathsf{pc} = \ell_1 \Rightarrow y = 0) \vee (\mathsf{pc} = \ell_2 \Rightarrow x > 0))$ instead.

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

We begin with ℓ_1. To check $\mathsf{AG}\ y \neq 1$ we use a source-to-source transformation that reduces checking of universal CTL properties to safety [CK13]. In further sections below, we demonstrate how a CTL model checking problem is indeed reduced to a safety problem. The transformation returns the program in Fig. 3.2 (new conditions outlined), on which we use a safety prover to check reachability of ERR. We get counterexample CEX_1: $\langle\ell_0, \rho_1, \ell_1\rangle$, $\langle\ell_1, \rho_3, \ell_1\rangle$, $\langle\ell_1, \rho_2, \ell_1\rangle$, $\langle\ell_1, \rho_4, \ell_2\rangle$, $\langle\ell_2, \rho_5, \ell_2\rangle$, $\langle\ell_2, \rho_7, \mathtt{ERR}\rangle$.

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

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

Note that the technique used by Cook & Koskinen [CK13] imposes that they spend time computing both $\wp\langle\ell_1, \varphi_{\mathsf{EF}}\rangle$, $\wp\langle\ell_2, \varphi_{\mathsf{EF}}\rangle$ separately while the technique used by Beyene *et al.* [BPR13] solves a constraint based on an entire path when it's only necessary to reason about a single state.

We now modify φ by using $\wp\langle\varphi_{\mathsf{EF}}\rangle$ and get $\varphi' = \mathsf{AG}\ ((\mathsf{pc} = \ell_1 \Rightarrow y = 0) \wedge (\mathsf{pc} = \ell_2 \Rightarrow x > 0))$. The constructed transformation reducing the property φ' to safety can be seen in Fig. 3.3. Note that in this particular transformation, the outlined instrumented conditions correspond to each of the location preconditions generated for $\mathsf{EF}\ y = 1$. As φ' is universal, we begin with the initial precondition $\wp\langle\varphi\rangle \triangleq \textsc{True}$. Failures to the proof attempt will result in strengthening the precondition by adding *negated* pre-images of discovered counterexamples. In this case no counterexamples are returned and we get $\wp\langle\varphi\rangle \triangleq \textsc{True}$. This proves that $\mathsf{AGEF}\ y = 1$ holds.

3.3 Procedure

In this section we describe the details of our CTL model checking procedure.

Algorithm 1 depicts Verify, which wraps our main procedure TemporalWP in Algorithm 2 by passing two parameters to TemporalWP, a program and a CTL formula to be verified.

ALGORITHM 1: Procedure VERIFY, which wraps TEMPORALWP and then checks all initial states.

```
1 Let VERIFY(φ, P) : bool =
2 |  (L, E, Vars) = P
3 |  ℘ = TEMPORALWP(φ, P)
4 |  return ∀(ℓ_I, ρ, ℓ) ∈ E ∀f : Vars → Vals . (f_{-1}, f) ⊨ ρ implies (ℓ, f) ⊨ ℘⟨ℓ, φ⟩
```

Other subroutines used in TEMPORALWP are in Algorithms 3–5. In our approach, the table $\wp$ is the key data structure that maps pairs of program locations and sub-formulae to assertions which represent the current candidate *precondition* that would guarantee the sub-formulae at a respective location. Thus TEMPORALWP is to return to VERIFY with a precondition map $\wp$ of the outermost sub-property φ, that is then checked as a satisfying condition against all the initial states of the program. In particular, a precondition $\wp\langle\ell, \varphi\rangle$ should be a sufficient and most general precondition to prove that φ holds at location ℓ. Hence in VERIFY, the precondition $\wp\langle\ell, \varphi\rangle$ produced from $\wp$ is checked against $\ell \in P$. If indeed $\wp\langle\ell, \varphi\rangle$ is a satisfying condition, then we can conclude that the property φ holds for P. After a short description of TEMPORALWP and a brief description of each of its subroutines, we give an in depth explanation of how TEMPORALWP produces $\wp$.

TemporalWP performs both a recursive and a refinement-based computation to construct $\wp$. In TEMPORALWP, each precondition synthesized substitutes its temporal sub-property in the original formula, and we then continue with the next most outer formula. It starts by initializing the map of preconditions using procedure INITIALIZEMAP (Algorithm 5) and then calling itself recursively for each sub-formula. TRANSFORM (Algorithm 3) and REFINE (Algorithm 4) are part of the model checking procedure for the current sub-formula while PROPAGATE (Algorithm 6) updates the map $\wp$ by synthesizing the pre-images given a counterexample. We then reduce the amount of redundant and irrelevant reasoning performed through information sharing extracted from reachability information. That is, several preconditions for each program location can be computed simultaneously.

Refine uses a safety prover [McM06, HB12a, KGC16], to obtain counterexamples from the newly transformed transition system, if a counterexample exists.

Transform implements the reduction of CTL model checking to safety and liveness checking, inspired by the procedure from [CK13]. Our transformation is adapted to handle one CTL sub-formula at a time, with any further nested sub-formulae to have synthesized preconditions ensuring their satisfaction. This contrasts [CK13], where each subsequent CTL sub-formulae transformations are recursively nested within one another. The transformation utilizes the map $\wp$, which would have mapped the preconditions synthesized to the corresponding previous sub-properties. The program is then transformed according to the CTL sub-property by modifying

the program from a given program location $k \in \mathcal{L}$. The reduction is only applied from a location k onwards , that is, we only wish to verify the sub-property starting from transitions stemming from k. The program is transformed in a way such that if φ does not hold for a location ℓ, a new reachable transition to an error location `ERR` is added.

As mentioned, existential path quantifiers are handled by considering their universal dual. For both existential and universal properties, our mapping function is also updated with the precondition for the negation of the property in TemporalWP and Propagate. This allows us to conveniently access the negation of the property when encoding existential properties as their universal duals. Recall that the precondition of a counterexample to a universal property corresponds to a witness of its existential dual. This will be discussed more extensively further below.

3.3.1 Computing $\wp$ for CTL

We now discuss our procedures in more detail, starting with our main algorithm TemporalWP that synthesizes our key data structure $\wp$. In order to synthesize a precondition for a temporal property φ, we first recursively decompose a CTL formula and accumulate the preconditions generated when considering its sub-formulae at lines 8, 13, and 14 in TemporalWP. The base case, α, is trivially computed as the precondition of an atomic predicate is the atomic predicate itself. For the sake of clarity, we omit the descriptions of the utilization of both CyclePoints and the use of sequential locality in Propagate till later, as we solely wish to describe the fundamental procedure underlying our precondition synthesis for each temporal sub-property. We will then discuss how these sub-procedures provide the key to making use of the program's control-flow graph to construct multiple preconditions.

Given the omission of CyclePoints, assume C is the set of *all* locations in a program P, that is $\mathcal{L}$. We wish to synthesize a precondition for each $\ell \in \mathcal{L}$ such that the precondition asserts the satisfaction of φ. Hence, we iterate over these locations (line 16) and generate a transformed program per each location using the subroutine Transform at line 19. Recall that Transform allows us to reduce the checking of temporal properties to a program analysis task from a given program location. We now describe Transform prior to explaining the remaining steps in TemporalWP.

CTL Reduction to Safety and Liveness

We demonstrate how each CTL formula can be reduced to a safety or liveness model checking problem, as carried out in Algorithm 3. As previously noted, we recursively partition a CTL formula, and for each nested sub-formula synthesize a precondition that ensures its satisfaction.

ALGORITHM 2: Two parameters are given to procedure TemporalWP: a program and a sub-property. The procedure returns a function that maps sub-properties to their synthesized preconditions. A precondition of a CTL sub-property is automatically synthesized from counterexamples and then is successively replaced by a condition over program states.

```
1  Let TemporalWP(φ, P) : map =
2      ℘ = InitializeMap(φ, P)
3      M = ∅
4      κ = [ ]
5      (L, E, Vars) = P
6      if φ = α ∈ AP then
7          foreach ℓ. (ℓ, ρ, ℓ') ∈ E do
8              ℘⟨ℓ, φ⟩ = α
9              ℘⟨ℓ, ¬φ⟩ = ¬α
10     else
11         match (φ) with
12             φ1 ∧ φ2 | φ1 ∨ φ2 | φ1 U φ2 | φ1 W φ2 →
13                 ℘ = ℘ ∪ TemporalWP(φ1, P) ∪ TemporalWP(φ2, P)
14             AF φ1 | AG φ1 | ¬φ1 → ℘ = ℘ ∪ TemporalWP(φ1, P)
15     C = CyclePoints(P)
16     foreach (ℓ, ρ, ℓ') ∈ E do
17         G = MinSCS(P, C, ℓ) ∈ SCS(P, C)
18         if G ≠ ∅ then
19             P' = Transform(⟨ℓ, φ⟩, M, P, ℘)
20             CEX, M = Refine(P', φ, ℘, M)
21             while CEX ≠ ∅ do
22                 ℘, P' = Propagate(CEX, P', κ, φ, ℓ, ℘)
23                 κ = CEX :: κ
24                 CEX, M = Refine(P', ⟨ℓ, φ⟩, ℘, M)
25     return ℘
```

ALGORITHM 3: Reduction of model checking of temporal properties to safety and ranking function synthesis.

Let TRANSFORM$(\langle k, \varphi\rangle, \mathcal{M}, P, \wp)$ *: Program* =
  $(\mathcal{L}, E, \mathsf{Vars}) = P$
  match (φ) **with**
    $\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2] \rightarrow$
      $\alpha_1 = \wp\langle k, \varphi_1\rangle$
      $\alpha_2 = \wp\langle k, \varphi_2\rangle$
    $\mathsf{AF}\varphi_1 \mid \mathsf{AG}\varphi_1$
      $\alpha_1 = \wp\langle k, \varphi_1\rangle$
  match (φ) **with**
    $\varphi \wedge \varphi' \rightarrow$
      $E = E \cup (k, \neg\alpha_1 \vee \neg\alpha_2, \texttt{ERR})$
    $\varphi \vee \varphi' \rightarrow$
      $E = E \cup (k, \neg\alpha_1 \wedge \neg\alpha_2, \texttt{ERR})$
    $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2] \rightarrow$
      foreach $(\ell, \rho, \ell') \in E$ *reachable from* k **do**
        $\rho = \rho \wedge \alpha_1 \wedge \neg\alpha_2$
        $E = E \cup (\ell, \neg\alpha_1 \wedge \neg\alpha_2, \texttt{ERR})$
    $\mathbf{E}[\varphi_1 \mathbf{U} \varphi_2] \rightarrow P =$ TRANSFORM$(\langle k, \mathsf{A}[\neg\varphi_2 \mathsf{W} (\neg\varphi_1 \wedge \neg\varphi_2)]\rangle, \mathcal{M}, P, \wp)$
    $\mathbf{AF}\varphi_1 \rightarrow$
      foreach $(\ell, \rho, \ell') \in E$ **do**
        $\rho = \rho \wedge \text{dup} =$ FALSE
        if $(\ell, \rho, \ell') \in E$ *reachable from* k **then**
          $\rho = (\rho \ \wedge \neg\alpha_1) \vee (\rho \wedge \neg\text{dup} \wedge \neg\alpha_1 \wedge \text{dup'} =$ TRUE $\wedge('s = (\ell, f))$
          $\text{c} = \text{dup} \wedge \neg\alpha_1 \wedge \neg(\exists f \in \mathcal{M}.\ f(s) \prec f('s))$
          $E = E \cup (\ell,$ c, $\texttt{ERR})$
    $\mathbf{EG}\varphi_1 \rightarrow \ P =$ TRANSFORM$(\langle k, \mathsf{AF}\neg\varphi_1\rangle, \mathcal{M}, P, \wp)$
    $\mathbf{AG}\varphi_1 \rightarrow$
      foreach $(\ell, \rho, \ell') \in E$ *reachable from* k **do**
        $\rho = \rho \wedge \alpha_1$
        $E = E \cup (\ell, \neg\alpha_1, \texttt{ERR})$
    $\mathbf{EF}\varphi_1 \rightarrow \ P =$ TRANSFORM$(\langle k, \mathsf{AG}\neg\varphi_1\rangle, \mathcal{M}, P, \wp)$
    $\mathbf{AX}\varphi_1 \rightarrow$
      foreach $(k, \rho, \ell') \in E$ **do**
        $\alpha_1 = \wp\langle \ell', \varphi_1\rangle$
        $E = E \cup (k, \rho \wedge \neg\alpha, \texttt{ERR})$
        $\rho = \rho \wedge \alpha_1$
  return P

The provided map $\wp$ precisely provides the mapping from the most immediate CTL sub-formula to its new-found precondition. Thus, on lines 3–8 the CTL formula is further partitioned into its sub-formulae, φ_1 and φ_2 (if applicable) and we retrieve the preconditions for our nested sub-properties through α_1 and α_2, respectively.

- We begin with $\wedge$ and $\vee$ on lines 10–13. Note how for a location k, we utilize the *negation* of the preconditions for φ_1 and φ_2. This is due to the fact that P is to be modified on lines 11 and 13 to include a new transition to the error location ERR if the CTL formula is violated. The negation of assertions α_1 and α_2 denote the violations to the preconditions of φ_1 and φ_2, respectively. In the case of $\wedge$, the property is considered violated if either φ_1 or φ_2 does not hold at a location k. In the case of $\vee$, the property is violated if both φ_1 or φ_2 do not hold at a location k.

- For the case that our CTL property is of the format $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, starting line 15 we traverse all locations reachable from k to instrument a transition to the error location, as the property could be violated from k *onwards*. Then for each transition $(\ell, \rho, \ell') \in E$ reachable from location k, ρ is modified to $\rho = \rho \wedge \alpha_1 \wedge \neg\alpha_2$. This imposes a further restriction on the transition relation denoting that the transition cannot proceed unless φ_1 holds while φ_2 does not. In the case that φ_2 does hold, the program does not need to proceed as the property has indeed been satisfied. In the case that both φ_1 and φ_2 are violated, a new transition to the error location ERR is instrumented in P at location ℓ, as shown on line 17.

- For the CTL property $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$, note that we call Transform again, but instead with its universal CTL dual $\mathsf{A}[\neg\varphi_2 \mathsf{W} (\neg\varphi_1 \wedge \neg\varphi_2)]$ as we are indeed seeking counterexamples to serve as witnesses to the property holding. That is, a counterexample to the negated property denotes that the property does indeed hold on some paths. The same is done for all other ECTL properties in Transform.

- In the case that $\varphi = \mathsf{AF}\varphi_1$, we prove liveness with regards to φ_1, and thus require additional instrumentation as originally proposed by [CPR06] on lines 21–26 in order to generate the appropriate ranking functions. New variables are introduced to record the state before the unrolling of the loop, where the variable "dup" indicates whether the first unrolling of the loop has occurred. The statement on line 24 checks whether the termination argument (which may initially be $\emptyset$) always holds between the current state and the recorded state. If the statement can be invalidated, ERR is then reachable due to the new instrumented transition to the error location on line 25. If ERR is unreachable, we have proved the validity of an existing termination argument, as the instrumentation forces the reachability checker to consider all possible loop unrollings. Our procedure Refine utilizes this transformation to capture liveness counterexamples on line 20 in

TemporalWP. The produced counterexample to a liveness property (such as AF) generated from our instrumentation contains a lasso fragment, we then attempt to find an accompanying set of ranking functions $\mathcal{M}$ that will show that the counterexample is not valid. We thus attempt to enlarge the set of ranking functions $\mathcal{M}$ using the method introduced in [CSZ13]. Otherwise, the absence of a set of ranking function would indicate the possible existence of a recurrence set. A recurrence set denotes the presence of a non-terminating path, indicating that the liveness property indeed does not hold. When verifying EG, the dual of AF, we indeed require the synthesis of a non-empty recurrence set as the precondition for the property [GHM+08]. A detailed flowchart of the termination proving procedure can be found in Chapter 7 Fig. 7.1.

ALGORITHM 4: Procedure Refine accepts a program, a program location, a temporal property, a map from locations and temporal properties to assertions, and a set of ranking functions. Refine proceeds to return a counterexample and a (possibly) larger set of ranking functions.

```
1  Let Refine(P, ⟨ℓ, φ⟩, ℘, M) : counterexample, ranking functions =
2      CEX = Reachable(P,ERR)
3      while P can reach ERR do
4          if CEX contains stem and lasso then
5              if ∃ witness f showing CEX′ w.f. then
6                  M = M ∪ {f}
7              else
8                  return CEX, M
9          else
10             return CEX, M
11         CEX = Reachable((Transform(⟨ℓ, φ⟩, M,P, ℘), ℓ0,ERR)
12     return CEX, M
```

- In the case that $\varphi = \mathsf{AG}\varphi_1$, we again iterate every transition $(\ell, \rho, \ell') \in E$ reachable from k where each transition ρ is further restricted such that the precondition α_1 of φ_1 holds. In the case the α_1 does not hold, we again add a new transition to the error location ERR in P at location ℓ, as shown on line 30. That is, AG can simply be reduced to a safety property in which φ_1 must always hold.

- In the case that $\varphi = \mathsf{AX}\varphi_1$, we iterate every transition $(k, \rho, \ell') \in E$, that is every transition originating from k, where α_1 is then assigned to the precondition of ℓ'. The transition ρ is then restricted such that the precondition α_1 of φ_1 at location ℓ' holds. Hence, at location k, we are indeed verifying that the property φ_1 holds at the "next" state.

- The property $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and its dual $\mathsf{E}[\varphi_1 \mathsf{W} \varphi_2]$ do not necessarily need to be explicitly supported, as they can be seen as syntactic sugar. That is, $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ can ultimately be written as $\mathsf{AF}\varphi_2 \wedge \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$.

Each transformed program P' produced from TRANSFORM on line 19 in TEMPORALWP is then verified through the subroutine REFINE (line 20). A counterexample-guided precondition refinement loop then begins at line 21, where we iteratively refine a precondition for $\ell \in \mathcal{L}$ until no more counterexamples are found. We note that the procedure REACHABLE in REFINE is a placeholder for any existing reachability checker [McM06, HB12b, KGC14]. We now discuss the refinement process for each type of path quantifier in TEMPORALWP separately below.

ALGORITHM 5: Initializing the map from program locations and sub-formulae to assertions. Preconditions of universal CTL formulae are initialized to TRUE as counterexamples are utilized to strengthen the initial condition. Given that existential formulae are handled by considering their universal dual, counterexamples serve as a witness thus weakening the initial condition of FALSE.

```
1  Let InitializeMap(φ, P) : map =
2      ℘ = ∅
3      (L, E, Vars) = P
4      if φ = Eψ' then
5          foreach ℓ ∈ L do
6              ℘⟨ℓ, φ⟩ = False
7              ℘⟨ℓ, ¬φ⟩ = True
8      else
9          foreach ℓ ∈ L do
10             ℘⟨ℓ, φ⟩ = True
11             ℘⟨ℓ, ¬φ⟩ = False
12     return ℘
```

Universal precondition synthesis

For a universal CTL sub-property $\varphi = \mathsf{A}\varphi_1$, a precondition $\wp\langle \ell, \varphi \rangle$ for a program location ℓ is initialized to TRUE (Algorithm 5 line 10). If REFINE returns a counterexample on line 20, we indeed refine $\wp\langle \ell, \varphi \rangle$ by passing said counterexample to PROPAGATE on line 22. Now consider Algorithm 6 PROPAGATE, and how given a CEX and a location ℓ, we compute the pre-image pre_ℓ as defined in Section 2.4, on line 8. Given our temporary omission of the sequential locality description in PROPAGATE, consider that we are only handling the current ℓ. To strengthen the precondition of a property φ and location ℓ, *i.e.* $\wp\langle \ell, \varphi \rangle$, we first negate the pre-image of the returned counterexample at line 13. The precondition thus becomes $\wp\langle \ell, \varphi \rangle = \wp\langle \ell, \varphi \rangle \wedge \neg pre_\ell(\mathsf{CEX})$. This denotes that the precondition has now been strengthened as to ensure that the path CEX, which violates the sup-property φ cannot occur. We then rule out the aforementioned counterexample from occurring again by adding the assumption $\neg\wp(\ell, \varphi)$ to each ingoing transition to the error location on the counterexample path, as shown on lines 15 and 16 in PROPAGATE. We then continue to iterate the loop in TEMPORALWP whenever a new counterexample is discovered while refining $\wp\langle \ell, \varphi \rangle$, resulting in a universal CTL formula precondition to be of the form

ALGORITHM 6: Procedure PROPAGATE receives a counterexample, a program, a list of previous counterexamples and their corresponding locations, and a map of previously discovered preconditions. It returns an updated map and updated program. The map of preconditions is updated by adding the weakest preconditions of the current counterexample. The program is updated by eliminating handled counterexamples from reaching the `ERR` location again.

1 **Let** PROPAGATE$(\mathsf{CEX}, \mathsf{P}, \kappa, \varphi, n, \wp)$ *: map, Program* =
2 $\alpha = \textsc{True}$
3 $(\mathcal{L}, E, \mathsf{Vars}) = P$
4 **foreach** $(\ell, \rho, \ell') \in \mathsf{CEX}$ **do**
5 **if** $\mathsf{CEX} \in \kappa \wedge \ell = n$ **then**
6 $\alpha = \textsc{Strengthen}(pre_\ell(\mathsf{CEX}), \mathsf{CEX})$
7 **else**
8 $\alpha = pre_\ell(\mathsf{CEX})$
9 **if** $\varphi = E\varphi'$ **then**
10 $\wp\langle \ell, \varphi \rangle = \wp\langle \ell, \varphi \rangle \vee \alpha$
11 $\wp\langle \ell, \neg\varphi \rangle = \wp\langle \ell, \neg\varphi \rangle \wedge \neg\alpha$
12 **else**
13 $\wp\langle \ell, \varphi \rangle = \wp\langle \ell, \varphi \rangle \wedge \neg\alpha$
14 $\wp\langle \ell, \neg\varphi \rangle = \wp\langle \ell, \neg\varphi \rangle \vee \alpha$
15 **if** $\ell' = \mathit{ERR}$ **then**
16 $\rho \in E = \rho \wedge \neg\wp\langle \ell, \varphi \rangle$
17 return $\wp, P$

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

Existential precondition synthesis

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

A precondition $\wp\langle \ell, \varphi \rangle$ for a program state is initially FALSE (line 6 in Algorithm 5). If a counterexample is returned, $\wp\langle \ell, \varphi \rangle$ is refined through the disjunction of the pre-image of the counterexample returned, that is $\wp\langle \ell, \varphi \rangle = \wp\langle \ell, \varphi \rangle \vee pre(\ell, \mathsf{CEX})$ (line 10 in Algorithm 6). Unlike for universal quantifiers, the pre-images need not be negated as they are witnesses for the property being satisfied.

We then rule out the aforementioned counterexample by adding the assumption $\neg pre_\ell(\mathsf{CEX})$ to each transition reaching `ERR`, and continue to unfold the loop with each newly discovered counterexample while iteratively refining $\wp\langle \ell, \varphi \rangle$. Note that finding one witness is not sufficient to satisfy an existential property, as $\wp\langle \ell, \varphi \rangle$ must characterize *all* the states satisfying the

sub-property φ at a location. Thus

$$\wp\langle \ell, \varphi \rangle = \bigvee_{n \in \mathbb{N}} pre_\ell(\mathsf{CEX}_n)$$

ALGORITHM 7: If divergence is suspected due to infinitely many counterexamples, the sub-procedure strengthens the candidate precondition towards the limit.

```
1 Let STRENGTHEN(α, CEX) : AP =
2 |  V = {v'. v ∈ Vars ∧ v' ∈ CEX}
3 |  QE(∃V.α)
4 |  return α
```

Upon the return of our precondition method to its caller, $\wp$ will contain the precondition for the most outer temporal property of the original CTL property φ. However, note that in our procedure, divergence can occur due to the possibility of generating infinitely many counterexamples. In practice this is rare, but not unheard of. We thus implement the following heuristic introduced by [CCF+14]:

- If we suspect we are looking at a sequence of repeated counterexamples that will result in divergence, we call the procedure STRENGTHEN (Algorithm 7 on line 6 in PROPAGATE). The sub-procedure strengthens the candidate precondition towards the limit.
- STRENGTHEN takes a calculated pre-image α, then proceeds to quantify out all variables that are updated proceeding the program location ℓ by applying quantifier elimination (QE).
- This heuristic can lead to unsoundness, as STRENGTHEN may over-approximate the set of states, causing $\wp$ to be potentially unsound for temporal properties involving existential path quantifiers. To check that the guessed candidate precondition is in fact a real precondition, *e.g.* that $\wp \Rightarrow \mathsf{EG}\ \wp'$, we can use the approach from Beyene *et al.* [BPR13] to double check the small lemma.
- If the check succeeds we continue, if the check fails we raise an exception.

Reducing redundant and irrelevant reasoning

Our approach synthesizes counterexample guided preconditions over program locations, but so far we have only shown how to do so for a single location at a time. We now demonstrate how we utilize sequential locality to simultaneously calculate preconditions for the set of locations that are arranged and can be accessed from a CEX starting from a given location ℓ. Our propagation sub-procedure PROPAGATE (Algorithm 6) is called from TEMPORALWP at line 22. We iterate

along the counterexample path, and for every reachable location $\ell \in \mathcal{L}$, we compute a pre-image utilizing a suffix of CEX from ℓ onwards. In more informal terms, every program location along the path can utilize the same counterexample to show that the property does or does not hold. Practically, the computation of a pre-image is performed by going backwards over the counterexample.

Propagate alone does not eliminate redundant or irrelevant reasoning, as we would still iterate over all locations whose preconditions have already been computed. Recall the program in Fig. 6.1 and the counterexample CEX_1 retrieved when model checking $\mathsf{AG}\ y \neq 1$: $\langle \ell_0, \rho_1, \ell_1 \rangle$, $\langle \ell_1, \rho_3, \ell_1 \rangle$, $\langle \ell_1, \rho_2, \ell_1 \rangle$, $\langle \ell_1, \rho_4, \ell_2 \rangle$, $\langle \ell_2, \rho_5, \ell_2 \rangle$, $\langle \ell_2, \rho_7, \texttt{ERR} \rangle$. More specifically, this counterexample is produced from ℓ_1, but as discussed, we can compute a pre-image utilizing the suffix of CEX_1 from any ℓ onwards. Thus we can avoid redundant reasoning by utilizing sequential locality based upon the program's control-flow graph to compute a refinement for ℓ_2 from a counterexample generated for ℓ_1. Given that ℓ_2 now has an accompanying precondition, it would be redundant to iterate over ℓ_2 in TemporalWP to calculate the same precondition. Thus in TemporalWP, we eliminate irrelevant locations such as ℓ_2 by solely iterating over strongly-connected subgraphs.

We thus calculate a strongly-connected subgraph set G from a set of cycles $C \subseteq \mathcal{L}$ (lines 15–18 in TemporalWP), as defined in 2.4, in which we synthesize a precondition over each program location ℓ that is an element of a minimal SCS. Such a location provides locality across program locations given the nature of cycles. That is, we will be able to propagate ℓ's precondition to all locations in $\textsc{MinSCS}(P, C, \ell) \in \text{SCS}(P, C)$ given they are within a generated counterexample for ℓ. Other program analysis inspired techniques may be used for the selection of initial locations to be verified. A cycle independent analysis can be run for those locations unreachable from program G.

We now state the correctness and soundness of our procedure.

3.3.2 Proof of Soundness

Proposition 3.1. *If the algorithm in Algorithm 2 terminates, for every sub-formula φ' in φ, every location $\ell \in \mathcal{L}$, and every reachable state s, we have $s \models \neg\wp\langle \ell, \varphi \rangle$ implies $P, (\ell, f) \models \neg\varphi$. $P, (\ell, f) \models \varphi$ implies $s \models \wp\langle \ell, \varphi \rangle$ provided that no spurious counterexamples are produced, and ranking functions are enumerable.*

We prove the proposition by induction on the structure of the formula. Consider a universal path formula, in which the counterexamples obtained from the underlying program analysis tool are real counterexamples, it follows that their pre-images do not satisfy the formula. Additional counterexamples further obtained are additionally sound, thus the termination of the

loop searching for counterexamples denotes that the disjunction of all pre-images is sound and complete. Given that existential path formulae are dual, the conjunction of all pre-images of counterexamples are also sound. The cases of atomic predicates and Boolean operators are trivial.

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

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

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

Soundness of our Technique.

We now restate Proposition 3.1 using the introduced notation considering the CTL formula $\mathsf{AG}\varphi$.

Proposition 3.2. *If the algorithm in Algorithm 2 terminates, for every sub-formula φ' in φ and every location $\ell \in \mathcal{L}$ we have*

$$P[\wp\langle \ell, \varphi' \rangle @ \ell] \models \mathsf{AG}(\mathsf{pc} = \ell^+ \Rightarrow \varphi') \wedge \mathsf{AG}(\mathsf{pc} = \ell^- \Rightarrow \neg\varphi')$$

We note how our formula φ is now wrapped in an AG formula, simply to indicate that *from all* paths, if location $\ell^{+\backslash -}$ is reached, then $\varphi' \backslash \neg\varphi'$ must hold. In our model checking procedure, this is not necessary as in VERIFY it is only necessary to check the initial state of a program P. Thus we are only using AG in our proof notation to denote how $\ell^{+\backslash -}$ is reached.

Proof. Consider the base case, that being the atomic predicate α. By construction, for every location ℓ we have $\wp\langle \ell, \alpha \rangle = \alpha$ and $\wp\langle \ell, \neg\alpha \rangle = \neg\alpha$. Then, $P[\alpha@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \alpha)$ and $P[\neg\alpha@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \neg\alpha)$.

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

- If $\varphi' = \varphi'_1 \vee \varphi'_2$. Consider a location ℓ where we denote $\wp_1 = \wp\langle \ell, \varphi'_1 \rangle$ and $\wp_2 = \wp\langle \ell, \varphi'_2 \rangle$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_1)$ and $P[\wp_2@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_2)$.
  Suppose that $P[\wp_{\varphi'}@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \varphi')$. Then, there is a state (ℓ', f) that is reachable in the program such that $(\ell', f) \not\models \ell^+ \Rightarrow \varphi'$ where $\ell' = \ell^+$. However, note that the encoding of φ' in Algorithm 3 adds the transition $(\ell, \neg\wp_1 \wedge \neg\wp_2, \texttt{ERR})$ to P. Then, the precondition synthesis terminates only when $\wp\langle \ell, \varphi' \rangle$ is strong enough to guarantee that ERR is not reachable in the modified program. Thus, it must be the case that the same state (ℓ, f) that serves as counterexample to $\mathsf{AG}(\ell \Rightarrow \varphi')$ would serve as a counterexample to $\mathsf{AG}\neg\texttt{ERR}$ in the modified program. Note that the dual argument (for ℓ^-) is similar and thus omitted.
- If $\varphi' = \varphi'_1 \wedge \varphi'_2$ the proof is similar to the previous case.
- If $\varphi' = \mathsf{A}[\varphi'_1 \mathsf{W} \varphi'_2]$. Consider a location ℓ where we denote $\wp_1 = \wp\langle \ell, \varphi'_1 \rangle$ and $\wp_2 = \wp\langle \ell, \varphi'_2 \rangle$. By induction, we know that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_i)$ and that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\varphi'_i)$. Suppose that $P[\wp'_\varphi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \varphi')$. Then, there is a state (ℓ', f) that is reachable in the program such that $(\ell', f) \not\models \ell^+ \Rightarrow \varphi'$. Then, $\ell' = \ell^+$. However, the encoding of φ' in Algorithm 3 adds to every location ℓ' the transition $(\ell', \neg\wp_1 \wedge \neg\wp_2, \texttt{ERR})$ to P. It additionally changes every transition to two transitions: one augmented by $\wp_1 \wedge \neg\wp_2$ to the same target and one augmented by $\wp_2$ that leads to locations from where the error is no longer reachable. Our precondition synthesis terminates only when $\wp\langle \ell, \varphi' \rangle$ is strong enough to guarantee that ERR is not reachable in the modified program. The completeness of the case of $\mathsf{A}[\varphi'_1 \mathsf{W} \varphi'_2]$ follows from the proof of $\mathsf{E}[\varphi'_1 \mathsf{U} \varphi'_2]$ below.
- If $\varphi' = \mathsf{E}[\varphi'_1 \mathsf{U} \varphi'_2]$, note that it is the dual of $\mathsf{A}[\varphi'_1 \mathsf{W} \varphi'_2]$ above. Now consider a location ℓ where we denote $\wp_1 = \wp\langle \ell, \varphi'_1 \rangle$ and $\wp_2 = \wp\langle \ell, \varphi'_2 \rangle$. By induction, we know that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_i)$ and that $P[\wp_i@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\varphi'_i)$.
  Suppose that $P[\wp'_\varphi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \varphi')$. Then, there is a state (ℓ', f) that is reachable in the program such that $(\ell', f) \not\models \ell^+ \Rightarrow \varphi'$. Then, $\ell' = \ell^+$. However, the encoding of φ' in Algorithm 3 treats EU as the dual of AW. Thus, it adds to every location ℓ a transition $(\ell, \wp_1 \wedge \wp_2, \texttt{ERR})$ and every other transition is replaced with two transitions one augmented by $\neg\wp_1 \wedge \wp_2$, leading to the same target, and one augmented by $\neg\wp_2$, which is then leading to a region where the transitions to ERR are no longer reachable. Then, the precondition synthesis extracts counterexamples that reach the ERR state. A path reaching ERR is a path that violates the dual AW and thus satisfies φ'. Thus, from every state satisfying the weakest precondition of this counterexample the formula φ' holds.
- If $\varphi' = \mathsf{AF}\varphi'_1$. Consider a location ℓ where we denote $\wp_1 = \wp\langle \ell, \varphi'_1 \rangle$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_1)$ and that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\varphi'_1)$. Now suppose that $P[\wp'_\varphi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \varphi')$, then there is a state (ℓ', f) that is reachable in the program such that $(\ell', f) \not\models \ell^+ \Rightarrow \varphi'$ indicating that $\ell' = \ell^+$. Recall that a path reaching

ERR is further analyzed in REFINE, where we then attempt to find an accompanying set of ranking functions $\mathcal{M}$ that will show that the counterexample is not valid, that is, we enlarge the set of ranking functions $\mathcal{M}$ using the well known method of [CPR06]. That is, the encoding of φ' in Algorithm 3 adds a transition to ERR only in case that a loop is found that does not have a ranking function. In the case that $\wp_1$ holds, it no longer becomes possible to reach ERR. Otherwise, from every state either it duplicates the state and searches for a loop to that state or continues. From the soundness for this program analysis procedure for ACTL, we know that when the program analysis task returns that the system is safe. It follows that the precondition synthesized is strong enough to ensure that the error location is not reached implying that there are no loops where φ'_1 does not hold. The completeness of the case of $\mathsf{AF}\varphi'_1$ follows from the proof of $\mathsf{EG}\varphi'_1$ below.

- If $\varphi' = \mathsf{EG}\varphi'_1$. Consider a location ℓ where we denote $\wp_1 = \wp\langle \ell, \varphi'_1 \rangle$. By induction, we know that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^+ \Rightarrow \varphi'_1)$ and that $P[\wp_1@\ell] \models \mathsf{AG}(\ell^- \Rightarrow \neg\varphi'_1)$. Suppose that $P[\wp'_\varphi@\ell] \not\models \mathsf{AG}(\ell^+ \Rightarrow \varphi')$, then there is a state (ℓ', f) that is reachable in the program such that $(\ell', f) \not\models \ell^+ \Rightarrow \varphi'$. Then, $\ell' = \ell^+$. However, the encoding of φ' in Algorithm 3 treats EG as the dual of AF. Thus, it adds transitions to ERR whenever a loop is found that does not visit $\neg\wp_1$. That is, the precondition synthesis extracts counterexamples that reach the ERR state. Given that a path reaching ERR is further analyzed in REFINE, where the absence of a set of ranking function would potentially lead to the existence of a recurrence set. We utilize the strategy in [GHM+08] to find a recurrence set to serve as the weakest precondition guaranteeing the satisfiability of EG. Thus, from every state satisfying the weakest precondition of this counterexample the formula φ' holds.

□

Corollary 3.2.1. *For every program P, if for every $(\ell_I, \rho, \ell) \in E$, $\rho \Rightarrow \wp\langle \ell_I, \varphi \rangle$ then $P \models \varphi$. Completeness holds (i.e., $\Leftarrow$) provided that no spurious counterexamples are produced, and ranking functions are enumerable. That is, if $P \models \varphi$, then for every $(\ell_I, \rho, \ell) \in E$, $\rho \Rightarrow \wp\langle \ell_I, \varphi \rangle$.*

As previously mentioned, assuming that the counterexamples obtained from the underlying program analysis tools are real counterexamples (*i.e.,* they are non-spurious), the termination of the loop searching for counterexamples denotes that the disjunction of all pre-images is sound and complete. For our corollary itself to be relatively complete, we rely on the further assumptions of the following algorithms. We assume that REFINE finds paths to the error state instrumented in the program. We assume that the computed weakest preconditions are accurate. Finally, we assume that ranking functions that rule out counterexamples to liveness properties can be found and are enumerable, that is, they can be represented as a possibly infinite list of ranking pairs.

3.4 Concluding remarks

In this chapter, we have described a procedure for CTL model checking that takes advantage of the structure of control-flow graphs available in programs. Our procedure works recursively on the structure of the property and computes (location-based) preconditions for the satisfaction of each sub-formula. The idea is to use a decomposition based on program-location (thus facilitating the use of program analysis techniques), but to maintain the current state of the intermediate lemmas in a way their results can be used to quickly facilitate the computation of results for nearby program locations. As is evident from the outcome of our experimental evaluation in Chapter 7, our method leads to dramatic performance improvement over competing tools that support CTL verification for infinite-state programs. Additionally, we wish to further experiment with the scalability that our methodology can perhaps provide.

Chapter 4

Fairness for Infinite-State Programs

In this chapter we introduce the first known tool for symbolically proving *fair*-CTL properties of (infinite-state) integer programs. Our solution is based on a reduction to our technique for fairness-free CTL model checking in Chapter 3, via the use of infinite non-deterministic branching to symbolically partition fair from unfair executions. We show the viability of our approach in practice using examples drawn from device drivers and algorithms utilizing shared resources further on in Chapter 7.

4.1 Introduction

In model checking, fairness allows us to bridge *some* of the expressive gaps between linear-time (*i.e.* trace-based) and branching-time (*i.e.* state-based) reasoning. Fairness is crucial, for example, to Vardi & Wolper's automata-theoretic technique for LTL verification [VW94]. Furthermore, when proving state-based CTL properties, we must often use fairness to model trace-based assumptions about the environment both in a sequential setting, and when reasoning about concurrent environments, where fairness is used to abstract away the scheduler.

In this chapter we thus introduce the first-known fair-CTL model checking technique for (infinite-state) integer programs. Our solution reduces fair-CTL to fairness-free CTL using prophecy variables, which determine future outcomes of the program execution, to encode a partition of fair from unfair paths. That is, prophecy variables introduce additional information into the state-space of the program under consideration, thus allowing fairness-free CTL proving techniques to reason only about fair executions. Cognoscenti may at first find this result surprising. It is well known that fair termination of Turing machines cannot be reduced to termination of Turing machines. The former is Σ_1^1-complete and the latter is RE-complete [Har86].[1] For

[1]Sometimes termination refers to *universal termination*, which entails termination for *all* possible inputs.

similar reasons fair-CTL model checking of Turing machines cannot be reduced to CTL model checking of Turing machines. The key to our reduction is the use of infinite non-deterministic branching when model checking fairness-free CTL. As a consequence, in the context of infinite branching, fair and fairness-free CTL are equally difficult (and similarly for termination).

4.1.1 Related Work

Support for fairness in finite and other decidable settings has been well studied. Tools for these settings (*e.g.* NuSMV for finite state systems [CCG+02, CES86], Moped and PuMoc for pushdown automata [EKS03, ST12], Prism for probabilistic timed automata [KNP11, KNP02], and Uppaal for timed automata [DHLP06]) provide support for fairness constraints. Proof systems for the verification of temporal properties of fair systems (e.g., [BBC+00], [PS08]) also exist. However, such systems require users to construct auxiliary assertions and participate in the proof process.

As in Chapter 3, we however seek to automatically verify the undecidable general class of (infinite-state) integer programs supporting both control-sensitive and numerical properties. And as discussed in Chapter 3, some of these tools do not fully support CTL model checking, as they do not reliably support mixtures of nested universal/existential path quantifiers, etc. The tools which consider full CTL and the general class of integer programs as we do again are [BPR13], [CKP14], and [CK13]. We emphasize that these tools provide no support for verifying fair-CTL.

When we consider the general class of integer programs, the use of infinite nondeterminism to encode fairness policies has been previously utilized by Olderog *et al.* [AO88]. However, they do not rely on nondeterminism alone but require refinement of the introduced nondeterminism to derive concrete schedulers which enforce a given fairness policy. Thus, their technique relies on the ability to force the occurrence of fair events whenever needed by the reduction. We support general fairness constraints, rather than just fair scheduling. The ability to force the occurrence of fair events is too strong for our needs. Indeed, in the context of model checking we rely on the program continuing a normal execution until the "natural" fulfillment of the fairness constraint. Olderog et al. explicitly note that implementing fairness can be done independently of temporal logic, and no further observances are made between their transformation and fair temporal logic. In contrast, we are able to demonstrate how our transformation can easily build upon recent CTL model checking techniques in order to solve CTL model checking for infinite-state programs with fairness.

An analysis of fair discrete systems which separates reasoning pertaining to fairness and well-foundedness through the use of inductive transition invariants was introduced in [PPR05]. Their

This is a harder problem and is co-RE^{RE}-complete.

strategy is the basis of the support for fairness added to TERMINATOR [CGP+07]. However, this approach relies on the computation of transition invariants [PR04b], whereas our approach does not. It has shown that, in practice, state-based techniques that circumvent the computation of transition invariants perform significantly better [CSZ13]. As shown in Chapter 3, our technique builds upon the latter and not the former. Finally a technique utilized to reduce LTL model checking to fairness-free CTL model checking introduced by [CK11] is largely incomplete, as it does not sufficiently determinize all possible branching traces. Note that these methodologies are used to verify fairness and liveness constraints expressible within linear temporal logic, and are thus not applicable to verify fair branching-time logic or branching-time logic. Indeed, this was part of our motivation for studying alternative approaches to model checking with fairness.

4.2 Fairness

Generally speaking, fairness is a related notion to liveness in that they both characterize progress over a program's execution. More specifically, a fairness constraint is a condition on paths of a program's model. Consider a transition system, where traces contain no information with regards to how often a process is executed, or how the next process to be executed is chosen. This is due to the fact that transition systems are independent of an underlying scheduler enforced perhaps by an operating system. That is, scheduling is treated nondeterministically. Hence, unlike liveness, fairness constraints are not properties to be verified over a transition system, but conditions assumed to be enforced by an underlying environment, *e.g., a scheduler*. Thus when verifying certain temporal properties, we might find that they are not satisfied, however, if the environment behaved fairly in its selection of processes to be executed, the property would indeed hold. Given that we cannot predict the behavior of an underlying environment, such as an operating system's scheduler, we instead can model-check various properties assuming the environment in which a program behaves fairly. Despite fairness being a *condition*, it can indeed still be expressed in temporal logic, more specifically in LTL. There exists 3 common notions of fairness [Wah]:

- Absolute Fairness, Impartiality: $\mathsf{GF}\ q$, q is executed infinitely often.
- Strong Fairness: $\mathsf{GF}\ p \Rightarrow \mathsf{GF}\ q$, if p holds infinitely often, then p must hold infinitely often.
- Weak Fairness: $\mathsf{FG}\ p \Rightarrow \mathsf{F}\ q$ or $\mathsf{FG}\ p \Rightarrow \mathsf{GF}\ q$, if p holds from some point and onwards, then q will eventually hold or will hold infinitely often, respectively.

We note that if a property holds under the assumption of weak fairness, then it also holds under both strong and absolute fairness. However, absolute fairness does not imply strong fairness,

$$\textsc{Fair}((S, S_0, R, L), (p, q)) \triangleq (S_\Omega, S^0_\Omega, R_\Omega, L_\Omega) \text{ where}$$

$$\begin{array}{rcll} S_\Omega & = & S \times \mathbb{N} & \\ R_\Omega & = & \{((s,n),(s',n')) \mid (s,s') \in R\} \wedge & \left(\begin{array}{c} (\neg p \wedge n' \leq n) \vee \\ (p \wedge n' < n) \vee \\ q \end{array}\right) \\ S^0_\Omega & = & S^0 \times \mathbb{N} & \\ L_\Omega(s,n) & = & L(s) & \end{array}$$

Figure 4.1: Fair takes a system (S, S_0, R, L) and a fairness constraint (p, q) where $p, q \subseteq S$, and returns a new system $(S_\Omega, S^0_\Omega, R_\Omega, L_\Omega)$. Note that $n \geq 0$ is implicit, as $n \in \mathbb{N}$.

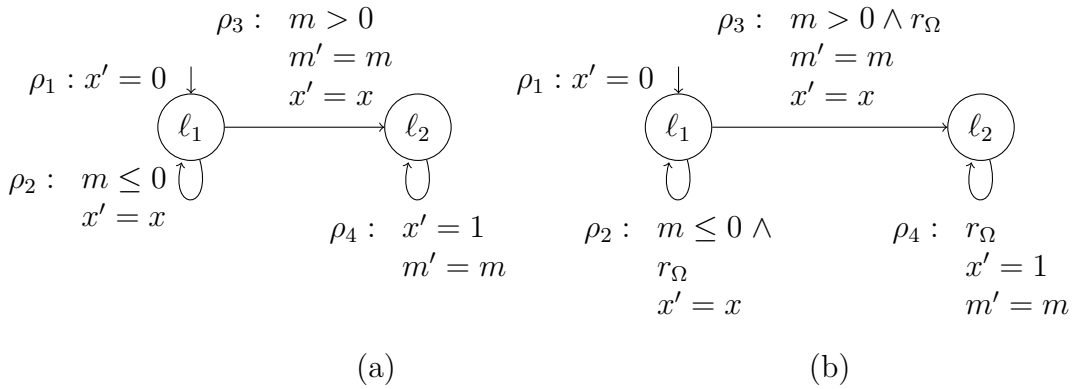
nor does strong fairness imply absolute fairness. In this chapter, we support both absolute and strong fairness with one pair of sets of states. Extending our results to fairness over multiple pairs is simple and omitted for clarity of exposition.

Recall that for a transition system P, a fairness condition is $\Omega = (p, q)$, where $p, q \subseteq S$. The condition Ω denotes a strong fairness constraint. Absolute fairness constraints can be trivially expressed by $\Omega = (\textsc{True}, q)$, that is, states from q must occur infinitely often. For a transition system P and a CTL property φ, the definition of when φ holds in a state $s \in S$ is defined as in Chapter 2 Section 2.3 except that $\Pi_m(s)$ is redefined to be $\Pi_f \cup \{\pi \in \Pi_\infty \mid P, \pi \models \Omega\}$. We use the notation $\models_{\Omega_+}$ when expressing fairness, that is, we say that φ holds in P, denoted by $P \models_{\Omega_+} \varphi$, if $\forall s \in S_0.P, s \models_{\Omega_+} \varphi$. When clear from the context, we may omit P and simply write $s \models_{\Omega_+} \varphi$ or $s \models_m \varphi$.

4.3 Fair-CTL Verification

In this section we present a procedure for reducing fair-CTL model checking to CTL model checking. The procedure builds on a transformation of infinite-state programs by adding a prophecy variable that truncates unfair paths. We start by presenting the transformation, followed by a program's adaptation for using said transformation, and subsequently the model checking procedure.

In Fig. 4.1, we propose a reduction $\textsc{Fair}(P, \Omega)$ that encodes an instantiation of the fairness constraint within a transition system. When given a transition system (S, S_0, R, L, Ω), where $\Omega = (p, q)$ is a strong-fairness constraint, $\textsc{Fair}(P, \Omega)$ returns a new transition system (without fairness) that, through the use of a prophecy variable n, infers all possible paths that satisfy the fairness constraint, while avoiding all paths violating the fairness policy. Intuitively, n is decreased whenever a transition imposing $p \wedge n' < n$ is taken. Since $n \in \mathbb{N}$, n cannot decrease infinitely often, thus enforcing the eventual invalidation of the transition $p \wedge n' < n$. Therefore, R_Ω would only allow a transition to proceed if q holds or $\neg p \wedge n' \leq n$ holds. That is, either q occurs infinitely often or p will occur finitely often. Note that a q-transition imposes no



$$r_\Omega : \{\ (\neg\rho_2 \wedge n' \leq n)\ \vee (\rho_2 \wedge n' < n)\ \vee m > 0\ \} \wedge n \geq 0$$

Figure 4.2: Reducing a transition system with the fair-CTL property $\mathsf{AG}(x = 0 \Rightarrow \mathsf{AF}(x = 1))$ and the fairness constraint $\mathsf{GF}\ \rho_2 \Rightarrow \mathsf{GF}\ m > 0$. The original transition system is represented in (a), followed by the application of our fairness reduction in (b).

constraints on n', which effectively resets n' to an arbitrary value. Recall that extending our results to multiple fairness constraints is simple and omitted for clarity of exposition.

The conversion of P with fairness constraint Ω to $\textsc{Fair}(P, \Omega)$ involves the truncation of paths due to the wrong estimation of the number of p-s until q. This means that $\textsc{Fair}(P, \Omega)$ can include (maximal) finite paths that are prefixes of unfair infinite paths. So when model checking CTL we have to ensure that these paths do not interfere with the validity of our model checking procedure. Hence, we distinguish between maximal (finite) paths that occur in P and those introduced by our reduction. We do this by adding a predicate t to mark all original "valid" terminating states prior to the reduction in Fig. 4.1 and by adjusting the CTL specification. These are presented in Section 4.3.3. We first provide high-level understanding of our approach through an example.

4.3.1 Illustrative Example

Consider the example in Fig. 4.2 for the fair-CTL property $\mathsf{AG}(x = 0 \Rightarrow \mathsf{AF}(x = 1))$ and the fairness constraint $\mathsf{GF}\ \rho_2 \Rightarrow \mathsf{GF}\ m > 0$ for the initial transition system introduced in (a). We demonstrate the resulting transformation for this infinite-state program, which allows us to reduce fair model checking to model checking. By applying $\textsc{Fair}(P, \Omega)$ from Fig. 4.1, we obtain (b) where each original transition, ρ_2, ρ_3, and ρ_4, are adjoined with restrictions such that $\{(\neg\rho_2\ \wedge\ n' \leq n)\ \vee (\rho_2\ \wedge\ n' < n)\ \vee m > 0\ \} \wedge n \geq 0$ holds. That is, we wish to restrict our transition relations such that if ρ_2 is visited infinitely often, then the variable m must be positive infinitely often. In ρ_2, the unconstrained variable m indicates that the variable m is being assigned to a nondeterministic value, thus with every iteration of the loop, m acquires a new value. In the original transition system, ρ_2 can be taken infinitely often given said non-

$$
\begin{aligned}
\textsc{Term}(\alpha, t) &::= \alpha \\
\textsc{Term}(\varphi_1 \wedge \varphi_2, t) &::= \textsc{Term}(\varphi_1, t) \wedge \textsc{Term}(\varphi_2, t) \\
\textsc{Term}(\varphi_1 \vee \varphi_2, t) &::= \textsc{Term}(\varphi_1, t) \vee \textsc{Term}(\varphi_2, t) \\
\textsc{Term}(\mathsf{EX}\varphi, t) &::= \neg t \wedge \mathsf{EX}(\textsc{Term}(\varphi, t)) \\
\textsc{Term}(\mathsf{AX}\varphi, t) &::= t \vee \mathsf{AX}(\textsc{Term}(\varphi, t)) \\
\textsc{Term}(\mathsf{EG}\varphi, t) &::= \mathsf{EG}\textsc{Term}(\varphi, t) \\
\textsc{Term}(\mathsf{AF}\varphi, t) &::= \mathsf{AF}\textsc{Term}(\varphi, t) \\
\textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t) &::= \mathsf{A}[\textsc{Term}(\varphi_1, t)\ \mathsf{W}\ \textsc{Term}(\varphi_2, t)] \\
\textsc{Term}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], t) &::= \mathsf{E}[\textsc{Term}(\varphi_1, t)\ \mathsf{U}\ \textsc{Term}(\varphi_2, t)]
\end{aligned}
$$

Figure 4.3: Transformation $\textsc{Term}(\varphi, t)$.

determinism, however in (b), such a case is not possible. The transition ρ_2 in (b) now requires that n be decreased on every iteration. Since $n \in \mathbb{N}$, n cannot be decreased infinitely often, causing the eventual restriction to the transition ρ_2. Such an incidence is categorized as a finite path that is a prefix of some unfair infinite paths. As previously mentioned, we will later discuss how such paths are disregarded. This leaves only paths where the prophecy variable "guessed" correctly. That is, it prophesized a value such that ρ_3 is reached, thus allowing our property to hold.

4.3.2 Prefixes of Infinite Paths

We explain how to distinguish between maximal (finite) paths that occur in P, and those that are prefixes of unfair infinite paths introduced by our reduction. Consider a transition system $P = (S, S_0, R, L, \Omega)$, where $\Omega = (p, q)$, and let φ be a CTL formula. Let t be an atomic predicate not appearing in L or φ. The transformation that marks "valid" termination states is $\textsc{Term}(P, t) = (S, S_0, R', L', \Omega')$, where $R' = R \cup \{(s, s) \mid \forall s'.(s, s') \notin R\}$, $\Omega' = (p, q \vee t)$ and for a state s we set $L'(s) = L(s) \cup \{t\}$ if $\forall s' \,.\, (s, s') \notin R$ and $L'(s) = L(s)$ otherwise.

That is, we eliminate all finite paths in $\textsc{Term}(P, t)$ by instrumenting self loops and adding the predicate t on all terminal states. The fairness constraint is adjusted to include paths that end in such states. We now adjust the CTL formula φ that we wish to verify on P. Recall that t does not appear in φ. Now let $\textsc{Term}(\varphi, t)$ denote the CTL formula transformation in Fig. 4.3 (we note that AG can be constructed using AW).

The combination of the two transformations maintains the validity of a CTL formula in a given system.

Theorem 4.1. $P \models_{\Omega_+} \varphi \Leftrightarrow \textsc{Term}(P, t) \models_{\Omega_+} \textsc{Term}(\varphi, t)$

Proof. For every fair path of $\textsc{Term}(P, t)$, we show that it corresponds to a maximal path in

P and vice versa. The proof then proceeds by induction on the structure of the formula. For existential formulae, witnesses are translated between the models. For universal formulae, we consider all paths and translate them between the models.

Base case: Consider an atomic predicate $\alpha \neq t$, assume $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\alpha,t)$. Given that $\textsc{Term}(\alpha,t) = \alpha$, we now have $\textsc{Term}(P,t), s \models_{\Omega_+} \alpha$ and $P, s \models_{\Omega_+} \alpha$. These statements are equivalent.

Induction hypothesis: For every state s and sub-formula φ (of lower temporal height) assume $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\varphi,t) \Leftrightarrow P, s \models_{\Omega_+} \varphi$.

($\Leftarrow$) $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\varphi,t) \Rightarrow P, s \models_{\Omega_+} \varphi$.

Proof by structural induction:

1. $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t) \Rightarrow P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$.
   Recall that $\textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t) = \mathsf{A}[\textsc{Term}(\varphi_1,t)\ \mathsf{W}\ \textsc{Term}(\varphi_2,t)]$. Consider a maximal path $\pi = (s_0, s_1, \ldots)$ starting at s in P. If $\pi \in \Pi_\infty$ and π is fair, then π is in $\textsc{Term}(P,t)$ as well and by $\textsc{Term}(P,t), s \models_{\Omega_+} \mathsf{A}[\textsc{Term}(\varphi_1,t)\ \mathsf{W}\ \textsc{Term}(\varphi_2,t)]$ we know either $\exists j \geq 0.\ \textsc{Term}(P,t), s_j \models_{\Omega_+} \textsc{Term}(\varphi_2,t) \wedge \forall i \in [0,j).\ \textsc{Term}(P,t)$, $s_i \models_{\Omega_+} \textsc{Term}(\varphi_1,t)$ or $\forall i \geq 0\ \textsc{Term}(P,t), s_j \models_{\Omega_+} \textsc{Term}(\varphi_1,t)$. Using our induction hypothesis we then have either $P, s_j \models_{\Omega_+} \varphi_2 \wedge \forall i \in [0,j).\ P, s_i \models_{\Omega_+} \varphi_1$ or $\forall i \geq 0.\ P, s_i \models_{\Omega_+} \varphi_2$. If $\pi \in \Pi_f$ then $\pi = (s_0, s_1, \ldots, s_n)$ and $\pi' = (s_0, s_1, \ldots, s_n, s_{n+1}, \ldots)$ is a path in $\textsc{Term}(P,t)$, where for every $k > 0, s_{n+k} = s_n$. As $\textsc{Term}(P,t), s \models_{\Omega_+} \mathsf{A}[\textsc{Term}(\varphi_1,t)\mathsf{W}\textsc{Term}(\varphi_2,t)]$ then either: There is a j such that $\textsc{Term}(P,t), s_j \models_{\Omega_+} \textsc{Term}(\varphi_2,t)$ and $\forall i \in [0,j).\ \textsc{Term}(P,t), s_i \models_{\Omega_+} \textsc{Term}(\varphi_1,t)$ If $j < n$, then clearly π satisfies $\varphi_1 \mathsf{W} \varphi_2$. If $j \geq n$ then $s_j = s_n$ and then $\textsc{Term}(P,t), s_n \models_{\Omega_+} \textsc{Term}(\varphi_2,t) \wedge \forall i \in [0,n).\ \textsc{Term}(P,t), s_i \models_{\Omega_+} \textsc{Term}(\varphi_1,t)$ implying $P, s_n \models_{\Omega_+} \varphi_2 \wedge \forall i \in [0,n).\ P, s_i \models_{\Omega_+} \varphi_1$. Or $\forall i \in [0,|\pi|).\textsc{Term}(P,t), s_i \models_{\Omega_+} \textsc{Term}(\varphi_2,t)$. We have $\forall i \in [0,|\pi|).\ P, s_i \models_{\Omega_+} \varphi_1$. It thus follows that $P, s \models_{\Omega_+} \mathsf{A}\varphi_1 \mathsf{W} \varphi_2$.
2. $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\mathsf{EX}\varphi,t) \Rightarrow P, s \models_{\Omega_+} \mathsf{EX}\varphi$.
   Given that $\textsc{Term}(\mathsf{EX}\varphi,t) = \neg t \wedge \mathsf{EX}(\textsc{Term}(\varphi,t))$ we now have $\textsc{Term}(P,t), s \models_{\Omega_+} \neg t \wedge \mathsf{EX}(\textsc{Term}(\varphi,t))$. First, consider $\textsc{Term}(P,t), s \models_{\Omega_+} \mathsf{EX}(\textsc{Term}(\varphi,t))$, which implies $\exists \pi = (s_0, s_1, \ldots).\ \textsc{Term}(P,t), s_1 \models_{\Omega_+} \textsc{Term}(\varphi,t)$. Now consider $\textsc{Term}(P,t)$, $s \models_{\Omega_+} \neg t$, that is, the predicate $\neg t$ indicates that (s, s_1) was not an instrumented transition. Given that we only instrument transitions in terminal states, $\neg t$ guarantees that s_1 is a successor of s as s is a non-terminal state in P. Recall our assumption that $\forall s'.\ \textsc{Term}(P,t), s' \models_{\Omega_+} \textsc{Term}(\varphi,t) \Rightarrow P, s' \models_{\Omega_+} \varphi$ and $(s, s_1) \in R$ thus $P, s_1 \models_{\Omega_+} \varphi$.
3. A similar proof to AW and EX follows for AX, AF, AG, EU, and EG.

($\Rightarrow$) $P, s \models_{\Omega_+} \varphi \Rightarrow \textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\varphi,t)$.

Proof by structural induction:

1. $P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2] \Rightarrow \textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t)$.
   Recall that $\textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t) = \mathsf{A}[\textsc{Term}(\varphi_1, t) \mathsf{W} \textsc{Term}(\varphi_2, t)]$. Consider a path π' starting at s in $\textsc{Term}(P,t)$, if π' is of the form $\pi' = (s_0, s_1, \ldots, s_n, s_{n+1}, s_{n+2}, \ldots)$ such that for every $k > 1$ we have $s_{n+k} = s_n$ and $s_n \models_{\Omega_+} t$, then $\pi = (s_0, s_1, \ldots, s_n)$ is a maximal path in P since terminating paths are instrumented as self-loops in $\textsc{Term}(P,t)$ and marked with the predicate t. If $P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ then either there is a $j \leq n$ such that $P, s_j \models_{\Omega_+} \varphi_2$ and $\forall i \in [0, j).\ P, s_i \models_{\Omega_+} \varphi_1$ or $\forall i \in [0, n].\ P, s_i \models_{\Omega_+} \varphi_1$. By our induction hypothesis in the first case we have $\textsc{Term}(P,t), s_j \models_{\Omega_+} \textsc{Term}(\varphi_2, t) \ \wedge\ \forall i \in [0, j).\ \textsc{Term}(P,t), s_i \models_{\Omega_+} \textsc{Term}(\varphi_1, t)$ and in the second case we have $\forall i \in [0, n].\ \textsc{Term}(P,t), s_i \models \textsc{Term}(\varphi_1, t)$. If π' is of the form $\pi' = (s_0, s_1, \ldots)$ such that the predicate t does not hold anywhere along the path, then $\pi = (s_0, s_1, \ldots)$ is an infinite path in P and by $P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ we know that either $\exists j \geq 0.\ P, s_j \models_{\Omega_+} \varphi_2 \ \wedge\ \forall i \in [0, j).\ P, s_i \models_{\Omega_+} \varphi_1$ or $\forall i \geq 0.\ P, s_i \models_{\Omega_+} \varphi_1$. Using our induction hypothesis we then have that either $\exists j \geq 0.\ \textsc{Term}(P,t), s_j \models_{\Omega_+} \textsc{Term}(\varphi_2, t) \ \wedge\ \forall i \in [0, j).\ \textsc{Term}(P,t), s_i \models_{\Omega_+} \textsc{Term}(\varphi_1, t)$ or $\forall i \geq 0.\ \textsc{Term}(P,t) \models \textsc{Term}(\varphi_1, t)$. Thus, $\textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2], t)$.
2. $P, s \models_{\Omega_+} \mathsf{EX}\varphi \Rightarrow \textsc{Term}(P,t), s \models_{\Omega_+} \textsc{Term}(\mathsf{EX}\varphi, t)$.
   We expand our formula to $P \models_{\Omega_+} \mathsf{EX}\varphi \Rightarrow \textsc{Term}(P,t) \models_{\Omega_+} \neg t \wedge \mathsf{EX}\textsc{Term}(\varphi, t)$. By $P, s \models_{\Omega_+} \mathsf{EX}\varphi$ we know that there is s_1 such that $(s, s_1) \in R$ and $P, s_1 \models_{\Omega_+} \varphi$, hence s is not terminal in P and $\textsc{Term}(P,t), s \models_{\Omega_+} \neg t$. Through our induction hypothesis it then follows that $\textsc{Term}(P,t), s_1 \models_{\Omega_+} \textsc{Term}(\varphi, t)$. Both arguments of the conjunction are thus satisfied hence $\textsc{Term}(P,t), s \models_{\Omega_+} \neg t \wedge \mathsf{EX}\textsc{Term}(\varphi, t)$.
3. A similar proof to AW and EX follows for AX, AF, AG, EU, and EG.

□

After having marked the "valid" termination points in P by using the transformation $\textsc{Term}(P,t)$, we must ensure that our fair-CTL model checking procedure ignores "invalid" finite paths in $\textsc{Fair}(P, \Omega)$. The finite paths that need to be removed from consideration are those that arise by wrong prediction of the prophecy variable n. The formula $term = \mathsf{AFAX}\ \textsc{False}$ holds in a state s iff all paths from s are finite. We denote its negation $\mathsf{EGEX}\ \textsc{True}$ by $\neg term$. Intuitively, when considering a state (s, n) of $\textsc{Fair}(P, \Omega)$, if (s, n) satisfies $term$, then (s, n) is part of a wrong prediction. If (s, n) satisfies $\neg term$, then (s, n) is part of a correct prediction. Further on, we will set up our model checking technique such that universal path formulae ignore violations that occur on terminating paths (which correspond to wrong predictions) and existential path formulae use only non-terminating paths (which correspond to correct predictions).

4.3.3 Fair-CTL Model Checking

ALGORITHM 8: Our procedure FAIRCTL(P, Ω, φ) which employs both an existing CTL model checker and the reduction FAIR(P, Ω). An assertion characterizing the states in which φ holds under the fairness constraint Ω is returned.

Let FAIRCTL(P, Ω, φ) : AP =
  match (φ) **with**
    $\varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{A}\varphi_1 \circ \varphi_2 \mid \mathsf{E}\varphi_1 \circ \varphi_2 \rightarrow$
      $a_{\varphi_1} =$ FAIRCTL(P, Ω, φ_1)
      $a_{\varphi_2} =$ FAIRCTL(P, Ω, φ_2)
    $\mathsf{A}\circ\varphi_1 \mid \mathsf{E}\circ\varphi_1 \rightarrow$
      $a_{\varphi_1} =$ FAIRCTL(P, Ω, φ_1)
    $\alpha \rightarrow a_{\varphi_1} = \alpha$
  match (φ) **with**
    $\mathsf{E}\varphi_1\mathsf{U}\varphi_2 \rightarrow \ \varphi' = \mathsf{E}[a_{\varphi_1}\mathsf{U}(a_{\varphi_2} \wedge \neg term)]$
    $\mathsf{EG}\varphi_1 \rightarrow \ \varphi' = \mathsf{EG}(a_{\varphi_1} \wedge \neg term)$
    $\mathsf{EX}\varphi_1 \rightarrow \ \varphi' = \mathsf{EX}(a_{\varphi_1} \wedge \neg term)$
    $\mathsf{A}\varphi_1\mathsf{W}\varphi_2 \rightarrow \ \varphi' = \mathsf{A}[a_{\varphi_1}\mathsf{W}(a_{\varphi_2} \vee term)]$
    $\mathsf{AF}\varphi_1 \rightarrow \ \varphi' = \mathsf{AF}(a_{\varphi_1} \vee term)$
    $\mathsf{AX}\varphi_1 \rightarrow \ \varphi' = \mathsf{AX}(a_{\varphi_1} \vee term)$
    $\varphi_1 \wedge \varphi_2 \rightarrow \ \varphi' = a_{\varphi_1} \wedge a_{\varphi_2}$
    $\varphi_1 \vee \varphi_2 \rightarrow \ \varphi' = a_{\varphi_1} \vee a_{\varphi_2}$
    $\alpha \rightarrow \ \varphi' = a_{\varphi_1}$
  $P' =$ FAIR(P, Ω)
  $a = CTL(P', \varphi')$
  match (φ) **with**
    $\mathsf{E}\varphi_1 \rightarrow$ return $\exists n \geq 0$. a
    $\mathsf{A}\varphi_1 \rightarrow$ return $\forall n \geq 0$. a
    $\_ \rightarrow$ return a

We use FAIR(P, Ω) to handle fair-CTL model checking. Our procedure employs an existing CTL model checking algorithm for infinite-state systems. We assume that the CTL model checking algorithm returns an assertion characterizing all the states in which a CTL formula holds, as proposed in Chapter 3. Additionally tools proposed by Beyene *et al.* [BPR13] and support this functionality. We denote such CTL verification tools by CTL(P, φ), where P is a transition system and φ a CTL formula.

ALGORITHM 9: CTL model checking procedure VERIFY, which utilizes the subroutine in Algorithm 8 to verify if a CTL property φ holds over P under the fairness constraints Ω.

Let VERIFY(P, Ω, φ) : *bool* =
  $a =$ FAIRCTL$(\text{TERM}(P, t), \Omega, \text{TERM}(\varphi, t))$
  return $\forall(\ell_I, \rho, \ell) \in E \ \forall f : \mathsf{Vars} \rightarrow \mathsf{Vals}$. $(f_{-1}, f) \models \rho$ implies $(\ell, f) \models a$

Our procedure adapting FAIR(P, Ω) is presented in Algorithm 8. Given a transition system P, a fairness constraint Ω, and a CTL formula φ, FAIRCTL returns an assertion characterizing the

states in which φ fairly holds. Initially, our procedure is called by VERIFY in Algorithm 9 where P and φ are initially transformed by $\textsc{Term}(P, t)$ and $\textsc{Term}(\varphi, t)$ discussed in Section 4.3.2. That is, $\textsc{Term}(P, t)$ marks all "valid" termination states in P to distinguish between maximal (finite) paths that occur in P and those introduced by our reduction. $\textsc{Term}(\varphi, t)$ allows us to disregard all aforementioned finite paths, as we only consider infinite paths, which correspond to a fair path in the original system.

Our procedure then begins by recursively enumerating over each CTL sub-property, wherein we attain an assertion characterizing all the states in which the sub-property holds under the fairness constraint Ω. These assertions will subsequently replace their corresponding CTL sub-properties as shown on lines 4, 5, and 7. Recall that $\circ$ denotes a temporal operator. A new CTL formula φ' is then acquired by adding an appropriate termination or non-termination clause (lines 10–18). This clause allows us to ignore finite paths that are prefixes of unfair infinite paths. Recall that other finite paths were turned infinite and marked by the predicate t in $\textsc{Term}(P, t)$.

Ultimately, our reduction $\textsc{Fair}(P, \Omega)$ is utilized on line 19, where we transform the input transition system P according to Fig. 4.1. With our modified CTL formula φ' and transition system P', we call upon the existing CTL model checking algorithm to return an assertion characterizing all the states in which the formula holds. The returned assertion is then examined on lines 21–24 to determine whether or not φ' holds under the fairness constraint Ω. If the property is existential, then it is sufficient that there exists at least one value of the prophecy variable such that the property holds. If the property is universal, then the property must hold for all possible values of the prophecy variable.

We state the correctness and completeness of our model checking procedure.

Theorem 4.2. *For every CTL formula φ and every transition system P with no terminating states we have for every $(\ell_I, \rho, \ell) \in E$, if $\rho \Rightarrow \textsc{FairCTL}\ (P, \Omega, \varphi)$ then $P \models_{\Omega_+} \varphi$. Relative completeness holds (i.e., $\Leftarrow$) provided that no spurious counterexamples are produced, and ranking functions are enumerable. That is, if $P \models_{\Omega_+} \varphi$ then for every $(\ell_I, \rho, \ell) \in E, \rho \Rightarrow \textsc{FairCTL}\ (P, \Omega, \varphi)$.*

Proof. First, we demonstrate that every infinite path in $\textsc{Fair}(P, \Omega)$ starting in (s, n) for some prophecy variable $n \in \mathbb{N}$ corresponds to an infinite path in P starting in s satisfying Ω. Consider an infinite path $\pi = (s_0, n_0), (s_1, n_1), \ldots$ in $\textsc{Fair}(P, \Omega)$. Let $\pi' = s_0, s_1, \ldots$ be its projection in P. Suppose that π' is unfair. That is, states of the form (s, n) where $s \in p$ occur infinitely often along π but states of the form (s, n) where $s \in q$ occur only finitely often. This implies that for infinitely many i's we have $n_i > n_{i+1}$. However, $\forall i \geq 0$ we have $n \geq 0$. As the only conjunct in R_Ω that allows n to increase requires q to hold. It thus follows that this is a contradiction.

Now we demonstrate that every fair path in P according to Ω starting in s corresponds to an infinite path in $\textsc{Fair}(P, \Omega)$ starting in (s, n) for some $n \in \mathbb{N}$. Consider an infinite path $\pi = s_0, s_1, \ldots$ in P such that $\pi \models \Omega$. If there are finitely many occurrences of states from p in π then let n_0 be the number of such occurrences plus 1. Then, the path $\pi' = (s_0, n_0), (s_1, n_1), \ldots$ where n_{i+1} is $n_i - 1$ if $s_i \in p$ and $n_{i+1} = n_i$. Otherwise, it is a path in $\textsc{Fair}(P, \Omega)$. Indeed, every transition such that $s_i \in p$ satisfies $p \wedge n' < n$ and every other transition satisfies $n' \leq n$.

From this correspondence of fair paths in P and infinite paths in $\textsc{Fair}(P, \Omega)$, we can safely disregard all the newly introduced finite paths given a transition system with no finite paths (i.e., $\textsc{Term}(P, t)$).

We now turn to the main theorem, which we prove by induction on the structure of the formula. Namely, we show that the assertion returned by $\textsc{FairCTL}(P, \Omega, \varphi)$ characterizes the set of states of P that satisfy φ. We note that for our base case, that is, atomic predicates and Boolean operators, the proof is immediate as it has been similarly shown by Theorem 4.1.

Induction hypothesis: Let a_i be the assertion characterizing the set of states satisfying φ_i. For every state s and every formula φ_i (of lower temporal height) assume $P, s \models \varphi_i \Leftrightarrow (s \Rightarrow a_i)$ for $i \in \{1, 2\}$.

($\Rightarrow$)

1. Consider the case that $\varphi = \mathsf{AX}\varphi_1$. Suppose that in $\textsc{Fair}(P, \Omega)$ we have that for every $n \geq 0$, $(s, n) \models \mathsf{AX}(a_1 \vee \mathit{term})$, we show that $s \models \mathsf{AX}\varphi_1$. Consider if there were no fair paths starting in s, then this trivially holds. Now consider if a fair path $\pi' = s, s_1, s_2, \ldots$ in P existed, and recall the correspondence established above between fair paths in P and infinite paths in $\textsc{Fair}(P, \Omega)$. We can then conclude that $\pi = (s, n), (s_1, n_1), (s_2, n_2), \ldots$ is an infinite path in $\textsc{Fair}(P, \Omega)$, hence, (s_1, n_1) cannot satisfy term. It thus must be the case that s_1 satisfies a_1 and thus $P, s \models_{\Omega_+} \mathsf{AX}\varphi_1$.
2. Consider the case that $\varphi = \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. Suppose that for every value of n we have $(s, n) \models \mathsf{A}[a_1 \mathsf{W} (a_2 \vee \mathit{term})]$, we show that $s \models \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. Consider if there were no fair paths starting in s, then this trivially holds. Now consider a fair path $\pi' = s, s_1, s_2, \ldots$ in P existed, and recall the correspondence established above between fair paths in P and infinite paths in $\textsc{Fair}(P, \Omega)$. We can then conclude that $\pi = (s, n), (s_1, n_1), (s_2, n_2), \ldots$ is an infinite path in $\textsc{Fair}(P, \Omega)$, hence, every state on π cannot satisfy term and we conclude that π satisfies $a_1 \mathsf{W} a_2$. It thus must be the case that as π' satisfies $\varphi_1 \mathsf{W} \varphi_2$ as required.
3. The the case of $\varphi = \mathsf{AF}\varphi_1$ is similar.
4. Consider the case that $\varphi = \mathsf{EX}\varphi_1$. Now suppose that for some value of n such that $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{EX}(a_1 \wedge \neg \mathit{term})$, we show that $P, s \models_{\Omega_+} \mathsf{EX}\varphi_1$. As $(s, n) \models \mathsf{EX}(a_1 \wedge \neg \mathit{term})$ it follows that there is an infinite path $\pi = (s, n), (s_1, n_1), \ldots$ such that $(s_1, n_1) \models a_1$. Recall the correspondence between fair paths in P and infinite paths in $\textsc{Fair}(P, \Omega)$. We conclude that $\pi' = s, s_1, \ldots$ is a fair path in P and $s_1 \models \varphi_1$. Thus, $P, s \models_{\Omega_+} \mathsf{EX}\varphi_1$.

5. Consider the case that $\varphi = \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Suppose that for some $n \geq 0$ we have $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{E}[a_1 \mathsf{U}(a_2 \wedge \neg term)]$, we show that $P, s \models_{\Omega_+} \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Given our assumption, there is a path $(s, n), (s_1, n_1), \ldots, (s_i, n_i)$ that satisfies $a_1 \mathsf{U}(a_2 \wedge \neg term)$ such that $s, s_1, \ldots, s_{i-1}$ satisfy a_1 and s_i satisfies $a_2 \wedge \neg term$. From (s_i, n_i) satisfying $\neg term$ there is an infinite path $(s, n), (s_1, n_1), \ldots, (s_i, n_i), (s_{i+1}, n_{i+1}), \ldots$ in $\textsc{Fair}(P, \Omega)$. Now recall the correspondence between fair paths in P and infinite paths in $\textsc{Fair}(P, \Omega)$. We can thus conclude that $\pi' = s, s_1, \ldots, s_i, s_{i+1}, \ldots$ is a fair path in P such that $s, s_1, \ldots, s_{i-1}$ satisfy φ_1 and s_i satisfies φ_2.
6. The case of $\varphi = \mathsf{EG}\varphi_1$ is similar.

($\Leftarrow$)

1. Consider the case that $\varphi = \mathsf{AX}\varphi_1$. Suppose that $P, s \models_{\Omega_+} \mathsf{AX}\varphi_1$. We show that for every $n \geq 0$ we have $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{AX}(a_1 \vee term)$. Consider a state (s, n) such that (s, n) has no successors, then it follows that $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{AX}(a_1 \vee term)$. Now consider a state (s, n) such that (s, n) has some successors, and let (s_1, n_1) be some successor of (s, n). If $(s_1, n_1) \models term$ then we have proved the property. Otherwise, there is an infinite path $(s_1, n_1), (s_2, n_2), \ldots$ in $\textsc{Fair}(P, \Omega)$. By the correspondence established above it must be the case that $s_1, s_2, \ldots$ is a fair path in P. Thus, as s_1 is a successor of s in P and $s, s_1, \ldots$ is a fair path in P, it must be the case that $s_1 \models \varphi_1$. Hence, by induction, $s_1 \models a_1$, and since n was arbitrary it follows that this holds for every possible value of n.
2. Consider the case that $\varphi = \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. Suppose that $P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. We show that for every $n \geq 0$ we have $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{A}[a_1 \mathsf{W}(a_2 \vee term)]$. Consider a state (s, n) that has no infinite paths starting from it, then it follows that $\textsc{Fair}(P, \Omega), (s, n) \models term$. Now consider a state (s, n) such that it has some infinite path starting from it and let $\pi = (s, n), (s_1, n_1), (s_2, n_2), \ldots$ be a maximal path in $\textsc{Fair}(P, \Omega)$. If π is infinite then by the correspondence established above it must be the case that $\pi' = s, s_1, s_2, \ldots$ is a fair path in P. As $P, s \models_{\Omega_+} \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ it follows that that π' satisfies $\varphi_1 \mathsf{W} \varphi_2$, thus π satisfies $a_1 \mathsf{W} a_2$.
   If π is finite then let (s_i, n_i) be the last state on π such that from (s_i, n_i) there is some infinite path. Let $(s, n), (s_1, n_1), \ldots, (s_i, n_i), (s'_{i+1}, n'_{i+1}), \ldots$ be this infinite path. It follows that $s, s_1, \ldots, s_i, s'_{i+1}, \ldots$ is a fair path in P and it must satisfy $\varphi_1 \mathsf{W} \varphi_2$. If φ_2 is satisfied on or before i then π satisfies $a_1 \mathsf{W} a_2$.Otherwise, all states $s, s_1, \ldots, s_i$ satisfy φ_1. Thus, the state (s_{i+1}, n_{i+1}) satisfies $term$ in $\textsc{Fair}(P, \Omega)$ and hence π satisfies $a_1 \mathsf{W}(a_2 \vee term)$. As n was arbitrary it follows that this holds for every possible value of n.
3. The case of $\varphi = \mathsf{AF}\varphi_1$ is similar to the above.
4. Consider the case that $\varphi = \mathsf{EX}\varphi_1$. Suppose that $P, s \models_{\Omega_+} \mathsf{EX}\varphi_1$. We show that for some $n \geq 0$, $\textsc{Fair}(P, \Omega), (s, n) \models \mathsf{EX}(a_1 \wedge \neg term)$. By assumption, there exists a fair path $s, s_1, s_2, \ldots$ in P such that $s_1 \models \varphi_1$. Recall the correspondence be-

tween infinite paths in $\textsc{Fair}(P,\Omega)$ and fair paths in P, then there is an infinite path $(s,n),(s_1,n_1),(s_2,n_2),\ldots$ in $\textsc{Fair}(P,\Omega)$. It thus follows that state (s_1,n_1) satisfies $\neg term$. As $s_1 \Rightarrow a_1$, it must be the case that $(s,n) \models \mathsf{EX}(a_1 \wedge \neg term)$.

5. Consider the case that $\varphi = \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Suppose that $P,s \models_{\Omega_+} \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$, and let a_1 be the assertion characterizing the set of states satisfying φ_1 and a_2 the assertion characterizing the set of states satisfying φ_2. By induction for every state s, we have $P,s \models \varphi_i \Leftrightarrow s \Rightarrow a_i$, for $i \in \{1,2\}$. We show that for some $n \geq 0$, $\textsc{Fair}(P,\Omega),(s,n) \models \mathsf{E}[a_1 \mathsf{U}(a_2 \wedge \neg term)]$. By assumption, there is a fair path $\pi' = s, s_1, s_2, \ldots$ in P such that $\pi' \models \varphi_1 \mathsf{U} \varphi_2$. Recall the correspondence between infinite paths in $\textsc{Fair}(P,\Omega)$ and fair paths in P, then there is an infinite path $\pi = (s,n),(s_1,n_1),(s_2,n_2),\ldots$ in $\textsc{Fair}(P,\Omega)$. Due to π, we have that $(s,n) \models \neg term$ and similarly for every state on this path. Since $s_i \Rightarrow a_j$ iff $P,s_i \models_{\Omega_+} \varphi_j$, it thus follows that $(s,n) \models \mathsf{E}[a_1 \mathsf{U}(a_2 \wedge \neg term)]$.
6. The case of $\varphi = \mathsf{EG}\varphi_1$ is similar.

□

Corollary 4.2.1. *For every CTL formula φ and every transition system P, if* $\textsc{Verify}(P,\Omega,\varphi)$ *returns true, then $P \models_{\Omega_+} \varphi$. Relative completeness holds (i.e., $\Leftarrow$) provided that no spurious counterexamples are produced, and ranking functions are enumerable. That is, if $P \models_{\Omega_+} \varphi$, then* $\textsc{Verify}(P,\Omega,\varphi)$ *returns true.*

Proof. Verify calls FairCTL on $\textsc{Term}(P,t)$ and $\textsc{Term}(\varphi,t)$. It follows that $\textsc{Term}(P,t)$ has no terminating states and hence Theorem 4.2 applies to it. By Theorem 4.1, the mutual transformation of P to $\textsc{Term}(P,t)$ and φ to $\textsc{Term}(\varphi,t)$ preserves whether or not $P \models_{\Omega_+}$. The corollary follows. □

4.4 Fair-ACTL Model Checking

In this section we show that in the case that we are only interested in universal path properties, i.e., formulas in ACTL, there is a simpler approach to fair-CTL model checking. In this simpler case, we can solely use the transformation $\textsc{Fair}(P,\Omega)$. Just like in Fair-CTL, we still must ignore truncated paths that correspond to wrong predictions. However, in this case, this can be done by a formula transformation.

Let $\textsc{NTerm}(\varphi)$ denote the transformation in Figure 4.4. The transformation ensures that universal path quantification ignores states that lie on finite paths that are due to wrong estimations of the number of p-s until q. Using this transformation, it is possible to reduce fair-ACTL model checking to (A)CTL model checking over $\textsc{Fair}(P,\Omega)$. Formally, this is stated in the following theorem.

$$
\begin{aligned}
\textsc{NTerm}(\alpha) &::= \alpha \\
\textsc{NTerm}(\varphi_1 \wedge \varphi_2) &::= \textsc{NTerm}(\varphi_1) \wedge \textsc{NTerm}(\varphi_2) \\
\textsc{NTerm}(\varphi_1 \vee \varphi_2) &::= \textsc{NTerm}(\varphi_1) \vee \textsc{NTerm}(\varphi_2) \\
\textsc{NTerm}(\mathsf{AX}\varphi) &::= \mathsf{AX}(\textsc{NTerm}(\varphi) \vee \mathit{term}) \\
\textsc{NTerm}(\mathsf{AF}\varphi) &::= \mathsf{AF}(\textsc{NTerm}(\varphi) \vee \mathit{term}) \\
\textsc{NTerm}(\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]) &::= \mathsf{A}[\textsc{NTerm}(\varphi_1)\ \mathsf{W}\ (\textsc{NTerm}(\varphi_2) \vee \mathit{term})]
\end{aligned}
$$

Figure 4.4: Transformation NTerm( ).

Theorem 4.3. *For every ACTL formula φ and every transition system P with no terminating states, we have $P \models_{\Omega_+} \varphi \Leftrightarrow \textsc{Fair}(P, \Omega) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$.*

Proof. The proof proceeds by induction on the structure of the formula. We show that if the property holds from s in P then for every n the (modified) property holds from (s, n) in $\textsc{Fair}(P, \Omega)$ and vice versa. Note that the initial states of $\textsc{Fair}(P, \Omega)$ are all the initial states of P annotated by all possible options of $n \in \mathbb{N}$. It follows that the combination of all transformations reduce fair-ACTL model checking to ACTL model checking. We start with an auxiliary claim:

Claim 1. *For every ACTL formula φ, every transition system with no terminating states P, and for every state s of P, if for infinitely many $n \in \mathbb{N}$ we have $\textsc{Fair}(P, \Omega), (s, n) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$ for $\forall n \in \mathbb{N}$, then we have $\textsc{Fair}(P, \Omega), (s, n) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$.*

We prove this claim by induction over the structure of the formula. It holds trivially for predicates and Boolean combinations of formulae.

1. Consider the case that $\varphi = \mathsf{AX}\varphi_1$. We recall that $\textsc{NTerm}(\varphi)$ is $\mathsf{AX}\varphi_1 \vee \mathit{term}$. Suppose that for infinitely many n's we have $\textsc{Fair}(P, \Omega) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$. Consider a successor s' of s and a value $n' \in \mathbb{N}$, then there is $n'' > n$ such that $\textsc{Fair}(P, \Omega) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$. If $\textsc{Fair}(P, \Omega), (s, n'') \models \mathit{term}$ then clearly $\textsc{Fair}(P, \Omega), (s', n') \models \mathit{term}$ as well. Otherwise, $\textsc{Fair}(P, \Omega), (s', n') \models \varphi_1 \vee \mathit{term}$. Thus, for infinitely many n' we have $\textsc{Fair}(P, \Omega), (s', n') \models \varphi_1 \vee \mathit{term}$ and by assumption it follows that this holds for all values of n. As s' was arbitrarily chosen it follows that the same holds for every successor of s'. Thus, for every $n' \in \mathbb{N}$ we have $\textsc{Fair}(P, \Omega), (s, n') \models \varphi \vee \mathit{term}$.
2. Consider the case that $\varphi = \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. We recall that $\textsc{NTerm}(\varphi)$ is $\mathsf{A}[\textsc{NTerm}(\varphi_1) \mathsf{W} (\textsc{NTerm}(\varphi_2) \vee \mathit{term})]$. Now suppose that for infinitely many n's we have $\textsc{Fair}(P, \Omega), (s, n) \models \textsc{NTerm}(\varphi) \vee \mathit{term}$. Consider a value n' and assume by contradiction that $\textsc{Fair}(P, \Omega), (s, n') \not\models \textsc{NTerm}(\varphi) \vee \mathit{term}$, then it follows that (s, n') is not terminating. Hence, there is an infinite path π starting in (s, n) that does not satisfy $\textsc{NTerm}(\varphi)$. By assumption there is an $n'' > n$ such that $\textsc{Fair}(P, \Omega), (s, n'') \models \textsc{NTerm}(\varphi) \vee \mathit{term}$. The path π' that is identical to π except that it starts in (s, n'') instead of in (s, n') is also

an infinite path from (s, n''), it thus follows that (s, n'') is not terminating and that the path π' does satisfy $\text{NTERM}(\varphi)$. Note that the only difference between π and π' is the initial state (and in it the value of n), it thus follows that either $(s, n'') \models \text{NTERM}(\varphi_1)$ or $(s, n'') \models \text{NTERM}(\varphi_2)$. In either case, we can find infinitely many different $n''' > n''$ such that the same thing will hold. It follows by induction that for infinitely many value n''' we have $(s, n''') \models \text{NTERM}(\varphi_1)$ or $(s, n''') \models \text{NTERM}(\varphi_2)$. Thus in all cases, the same follows for all values of n and we conclude that π cannot be a counterexample to $\text{NTERM}(\varphi_1)$.

3. Consider the case that $\varphi = \mathsf{AF}\varphi_1$. We recall that $\text{NTERM}(\varphi)$ is $\mathsf{AF}(\text{NTERM}(\varphi_1) \vee term)$. Suppose that for infinitely many n's we have $\text{FAIR}(P, \Omega), (s, n) \models \text{NTERM}(\varphi) \vee term$. Now consider a value n' and assume by contradiction that $\text{FAIR}(P, \Omega), (s, n') \not\models \text{NTERM}(\varphi) \vee term$, it follows that (s, n') is not terminating. Hence, there is an infinite path π starting in (s, n) that does not satisfy $\text{NTERM}(\varphi)$. By assumption there is an $n'' > n$ such that $\text{FAIR}(P, \Omega), (s, n'') \models \text{NTERM}(\varphi) \vee term$. Note that the path π' is identical to π except that it starts in (s, n'') instead of in (s, n') is also an infinite path from (s, n''), it thus follows that (s, n'') is not terminating, and the path π' does satisfy $\text{NTERM}(\varphi)$. The only difference between π and π' is the initial state (and in it the value of n), similarly it follows that $(s, n'') \models \text{NTERM}(\varphi_1)$. and hence we can find infinitely many values for which $(s, n''') \models \text{NTERM}(\varphi_1)$. By assumption it follows that for every value of n we have $(s, n) \models \varphi_1 \vee term$, thus, it must be the case that $(s, n') \models \text{NTERM}(\varphi) \vee term$.

We now prove a stronger claim, which implies the Theorem: For every ACTL formula φ, every transition system with no terminating states P, and every state s of P we have

$$P, s \models_{\Omega_+} \varphi \Leftrightarrow \forall n \in \mathbb{N}.\ \text{FAIR}(P, \Omega), (s, n) \models \text{NTERM}(\varphi) \vee term$$

We prove this claim by induction on the structure of the formula.

Base case: Consider an atomic predicate $\alpha \neq t$. Clearly, for every s and every n we have $P, s \models_{\Omega_+} \alpha \Leftrightarrow \text{FAIR}(P, \Omega), (s, n) \models \text{NTERM}(\alpha)$. The proof for Boolean operators is immediate.

($\Rightarrow$)

1. Consider the case that $\varphi = \mathsf{AX}\varphi_1$. Suppose that $P, s \models_{\Omega_+} \mathsf{AX}\varphi_1$, we show that $\text{FAIR}(P, \Omega), (s, n) \models \text{NTERM}(\mathsf{AX}\varphi_1) \vee term$ for an arbitrary $n \in \mathbb{N}$. Recall that $\text{NTERM}(\mathsf{AX}\varphi_1) = \mathsf{AX}(\varphi_1 \vee term)$, and thus if (s, n) is terminating in $\text{FAIR}(P, \Omega)$ then clearly $\text{FAIR}(P, \Omega), (s, n) \models term$. Otherwise, consider an infinite path $\pi' = (s_0, n_0), (s_1, n_1), \ldots$ starting in (s, n) in $\text{FAIR}(P, \Omega)$. Let $\pi = s_0, s_1, \ldots$ be the projection of π' on the states in P. As shown above π is a fair path in P. Hence, as $P, s \models_{\Omega_+} \mathsf{AX}\varphi_1$ it follows that $P, s_1 \models_{\Omega_+} \varphi_1$ and by induction assumption $\text{FAIR}(P, \Omega), (s_1, n_1) \models \varphi_1 \vee term$. The same argument works for every successor (s_1, n_1) of (s, n), thus $\text{FAIR}(P, \Omega), (s, n) \models \text{NTERM}(\mathsf{AX}\varphi_1) \vee term$.
2. Consider the case that $\varphi = \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$. Suppose that $P, s \models_{\Omega_+} \varphi$, we show that

$\textsc{Fair}(P,\Omega),(s,n) \models \textsc{NTerm}(\varphi) \vee term$.

Recall that $\textsc{NTerm}(\varphi) = \mathsf{A}[\textsc{NTerm}(\varphi_1)\mathsf{W}(\textsc{NTerm}(\varphi_2) \vee term)]$, if (s,n) is terminating then the property trivially holds. Otherwise, consider an infinite path $\pi' = (s_0,n_0),(s_1,n_1),\ldots$ starting in (s,n) in $\textsc{Fair}(P,\Omega)$. Let $\pi = s_0,s_1,\ldots$ be the projection of π' on the states in P. As shown above π is a fair path in P. Hence, as $P,s \models_{\Omega_+} \varphi$ it follows that either for every $i \geq 0$ we have $P,s_i \models_{\Omega_+} \varphi_1$ or there is some $j \geq 0$ such that $P,s_j \models_{\Omega_+} \varphi_2$ and for every $i \in [0,j)$ we have $P,s_i \models_{\Omega_+} \varphi_1$. In both cases, it follows from the induction hypothesis that the path π' satisfies $\textsc{NTerm}(\varphi_1)\mathsf{W}(\textsc{NTerm}(\varphi_2) \vee term)$. Now consider a finite path $\pi' = (s_0,n_0),(s_1,n_1),\ldots,(s_m,n_m)$ starting in (s,n) in $\textsc{Fair}(P,\Omega)$. Let $\pi'' = (s_0,n_0),\ldots,(s_i,n_i),(s'_{i+1},n'_{i+1}),\ldots$ be an infinite path in $\textsc{Fair}(P,\Omega)$ that has a maximal joint prefix with π'. That is, (s_{i+1},n_{i+1}) is terminating in $\textsc{Fair}(P,\Omega)$. As above, the projection of π'' is going to be a fair path in P and thus, the prefix of π'' that agrees with π' either all satisfies $\textsc{NTerm}(\varphi_1)$ or $\textsc{NTerm}(\varphi_2)$ holds somewhere along it. In the first case, as (s_{i+1},n_{i+1}) is terminating we have that π' satisfies the AW formula. In the second case, the AW formula holds already by inspecting only the prefix of π', proving our claim.

3. The case for AF is similar.

($\Leftarrow$)

1. Consider the case that $\varphi = \mathsf{AX}\varphi_1$. Suppose that $\forall n \in \mathbb{N}$ we have $\textsc{Fair}(P,\Omega),(s,n) \models \varphi \vee term$. If there are no fair paths starting from s in P, then, clearly, $P,s \models \varphi$. Otherwise, consider a fair path $\pi = s_0,s_1,\ldots$ starting in s, and let n_i be the size of $\{j \geq i \mid s_j \in q \text{ and } \forall k \in [i,j] s_k \notin p\}$. Then, $(s_0,n_0),(s_1,n_1),\ldots$ is an infinite path in $\textsc{Fair}(P,\Omega)$, implying that $\textsc{Fair}(P,\Omega),(s_1,n_1) \models \textsc{NTerm}(\varphi_1) \vee term$. As we have shown, $\textsc{Fair}(P,\Omega),(s_1,n_1) \not\models term$, it thus follows that $\textsc{Fair}(P,\Omega),(s_1,n_1) \models \textsc{NTerm}(\varphi_1)$, and the same clearly holds for every $n'_1 > n_1$. We can now conclude that for every $n \in \mathbb{N}$ we have $(s_1,n) \models \textsc{NTerm}(\varphi_1) \vee term$. Hence, for every successor s'_1 of s and for every $n \in \mathbb{N}$ we have $(s'_1,n) \models \textsc{NTerm}(\varphi_1 \vee term)$, and thus by induction we can conclude that $s'_1 \models \varphi_1$, proving our claim.
2. Consider the case that $\varphi = \mathsf{A}[\varphi_1\mathsf{W}\varphi_2]$.

   Suppose that $\forall n \in \mathbb{N}$ we have $\textsc{Fair}(P,\Omega),(s,n) \models \varphi \vee term$. If there are no fair paths starting from s in P, then, clearly, $P,s \models \varphi$. Otherwise, consider a fair path $\pi = s_0,s_1,\ldots$ starting in s. As in the proof of AX we can find an annotation $\pi' = (s_0,n_0),(s_1,n_1),\ldots$ that is an infinite path in $\textsc{Fair}(M,\Omega)$. Either $\forall i \geq 0$ we have $(s_i,n_i) \models \textsc{NTerm}(\varphi_1)$ or there is $j \geq 0$ such that $(s_j,n_j) \models \textsc{NTerm}(\varphi_2) \vee term$ and $\forall 0 \leq i < j$ we have $(s_i,n_i) \models \textsc{NTerm}(\varphi_1)$. As by assumption $\textsc{NTerm}(\varphi) \vee term$ holds for (s,n') for every n' we can show that for every one of these states and for every $n'_i > n_i$ the same property (i.e., $\textsc{NTerm}(\varphi_1)$ or $\textsc{NTerm}(\varphi_2) \vee term$) holds for (s_i,n'_i). Thus, there are infinitely many values for which these properties hold. Consider, without loss of generality the property φ_1, then by the claim above

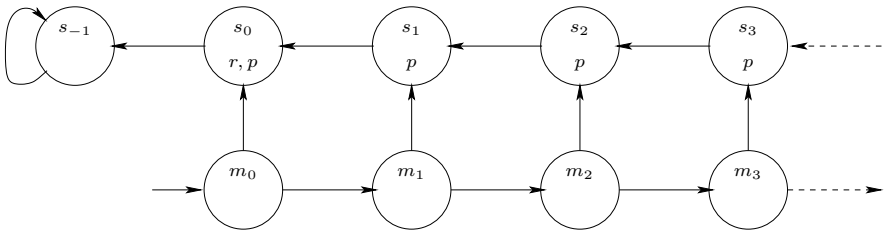


Figure 4.5: A system showing that ECTL model checking is more complicated.

NTerm(φ_1) $\vee$ *term* holds for all n'_i in the state (s'_i, n'_i). Thus, we can conclude from the induction assumption that the same pattern of satisfaction implies that π satisfies the AW property.

3. The case for AF is similar.

□

Corollary 4.3.1. *For every ACTL formula φ we have*

$$P \models_{\Omega_+} \varphi \Leftrightarrow \textsc{Fair}(\textsc{Term}(P,t),\Omega) \models \textsc{NTerm}(\textsc{Term}(\varphi,t)) \vee term$$

Proof. As Term(P,t) produces a transition system with no terminating states and Term(φ,t) converts an ACTL formula to an ACTL formula, the proof then follows from Theorem 4.1 and Theorem 4.3. □

The direct reduction presented in Theorem 4.3 works well for ACTL but does not work for existential properties. We now demonstrate why Fig. 4.1 is not sufficient to handle existential properties alone. Consider the transition system P in Figure 4.5, the fairness constraint $\Omega = \{(p,q)\}$, and the property $\mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. One can see that $P, m_0 \models_{\Omega_+} \mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. Indeed, from each state s_i there is a unique path that eventually reaches s_0, where it satisfies r, and then continues to s_{-1}, where p does not hold. As the path visits finitely many p states it is clearly fair. So, every state m_i satisfies $\mathsf{EF}r$ by considering the path $m_i, s_i, s_{i-1}, \ldots, s_0, s_{-1}, \ldots$. Then the fair path $m_0, m_1, \ldots$ satisfies $\mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. On the other hand, it is clear that no other path satisfies $\mathsf{EG}(\neg p \wedge \mathsf{EF}r)$.

Now consider the transformation Fair(P,Ω) and consider model checking of $\mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. In Fair$(P,\Omega)$ there is no path that satisfies this property. To see this, consider the transition system Fair(P,Ω) and a value $n \in \mathbb{N}$. For every value of n the path $(m_0,n),(m_1,n),(m_2,n),\ldots$ is an infinite path in Fair(P,Ω) as it never visits p. This path does not satisfy $\mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. Consider some state (m_j,n_j) reachable from (m_0,n) for $j > 2n$. The only infinite paths starting from (m_j,n_j) are paths that never visit the states s_i. Indeed, paths that visit s_i are terminated as they visit too many p states. Thus, for every $n \in \mathbb{N}$ we have $(m_0,n) \not\models \mathsf{EG}(\neg p \wedge \mathsf{EF}r)$. Finite

paths in $\textsc{Fair}(P, \Omega)$ are those of the form $(m_0, n_0), \ldots, (m_i, n_i), (s_i, n_{i+1}), \ldots$. Such paths clearly cannot satisfy the property $\mathsf{EG}(\neg p \wedge \mathsf{EF} r)$ as the states s_i do satisfy p. Allowing existential paths to ignore fairness is clearly unsound. We note also that in $\textsc{Fair}(P, \Omega)$ we have $(m_0, n) \models \textsc{NTerm}(\mathsf{AF}(p \vee \mathsf{AG}\neg r))$.

Reducing Fair Termination to Termination. Given the importance of termination as a system property, we emphasize the reduction of fair termination to termination. Note that termination can be expressed in ACTL as AFAX FALSE, thus the results in Corollary 4.3.1 allow us to reduce fair termination to model checking (without fairness). Intuitively, a state that satisfies AX false is a state with no successors. Hence, every path that reaches a state with no successors is a finite path. Here, we demonstrate that for infinite-state infinite-branching systems, fair termination can be reduced to termination.

A transition system P terminates if for every initial state $s \in S_0$ we have $\Pi_\infty(s) = \emptyset$. System P fair-terminates under fairness Ω if for every initial state $s \in S_0$ and every $\pi \in \Pi_\infty(s)$ we have $\pi \not\models \Omega$, i.e., all infinite paths are unfair.

The following corollary follows from the proof of Theorem 4.3, where we establish a correspondence between fair paths of P and infinite paths of $\textsc{Fair}(P, \Omega)$.

Corollary 4.3.2. *P fair terminates iff $\textsc{Fair}(P, \Omega)$ terminates.*

Recall that the reduction relies on transition systems having an infinite branching degree. For transition systems with finite-branching degree, we cannot reduce fair termination of finite-branching programs to termination of finite-branching programs, as the former is Σ^1_1-complete and the latter is RE-complete [Har86].

4.5 Example

Consider the example in Fig. 4.6. We will demonstrate the resulting transformations which will disprove the CTL property EG $x \leq 0$ under the weak fairness constraint GF TRUE $\rightarrow$ GF $\mathsf{y} \geq 1$ for the initial transition system introduced in (a). We begin by executing VERIFY in 9. In VERIFY the transition system in (a) is transformed according to $\textsc{Term}(P, t)$ and the CTL formula EG $x \leq 0$ is transformed according to $\textsc{Term}(P, t)$, as discussed in 4.3.2. Our main procedure FAIRCTL in 8 is then called. First, we recursively enumerate over the most inner sub-property $x \leq 0$, wherein $x \leq 0$ is returned as it is our base case. In lines 10–18, a new CTL formula φ' is then acquired by adding an appropriate termination or non-termination clause. This clause allows us to ignore finite paths that are prefixes of some unfair infinite paths, that is, those that have not been marked by $\textsc{Term}(P, t)$. We then obtain (b) in Fig. 4.6

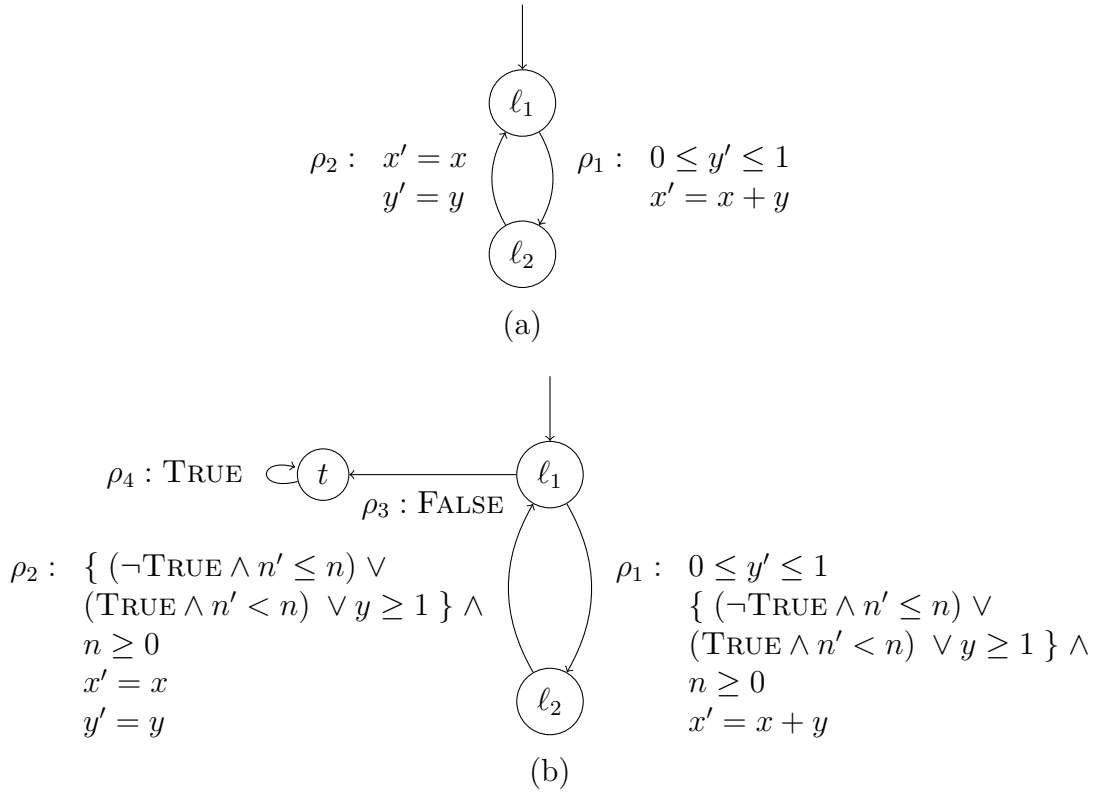


Figure 4.6: Verifying a transition system with the CTL property EG $x \le 0$ and the weak fairness constraint GF True $\rightarrow$ GF $y \ge 1$. The original transition system is represented in (a), followed by the application of our fairness reduction in (b).

by applying Fair(P, Ω) from Fig. 4.1 on line 19. Thus, we must restrict each transition such that $\{ (\neg\text{True} \wedge n' \le n) \vee (\text{True} \wedge n' < n) \vee y \ge 1 \} \wedge n \ge 0$ holds. This can be seen in transitions ρ_1 and ρ_2.

Recall that Fair(P, Ω) can include (maximal) finite paths that are prefixes of unfair infinite paths. We thus have to ensure that these paths do not interfere with the validity of our model checking procedure. We have shown how to distinguish between maximal (finite) paths that occur in P and those introduced by our transformation in Theorem 4.1. This is demonstrated by ρ_3 and ρ_4 in (b): in ρ_3 we simply take the negation of the loop invariant (in this case it is False), as it would indicate a terminating path given that no other transitions follow the loop termination. In ρ_4 we instrument a self loop and add the predicate t to eliminate all terminal states. Additionally, utilizing Term(φ, t) on EG $x \le 0$ allows us to disregard all aforementioned marked finite paths, as we only consider infinite paths which correspond to a fair path in the original system.

On line 20, a CTL model checker is then employed with the transition system in (b) and the CTL formula φ'. We then apply our methodology in Chapter 3 to the transformation introduced to verify CTL for infinite-state systems. An assertion characterizing the states in which φ' holds is returned and then further examined on lines 21–24, where it is discovered that this property

does not hold due to the restrictive fairness constraint applied to the existential CTL property. The weak fairness constraint requires that infinitely often $y \geq 1$ holds, which interferes with the existential property that EG $x \leq 0$. This shows that for the existential fragment of CTL, fairness constraints restrict the transition relations required to prove an existential property. This can be beneficial when attempting to disprove systems and their negations.

4.6 Concluding Remarks

We have introduced the first-known fair-CTL model checking procedure for infinite-state programs. Our solution reduces fair-CTL to fairness-free CTL via the use of a prophecy variable to introduce additional information into the state-space of the program under consideration. In the reduction, prophecy variables symbolically partition fair from unfair paths, thus allowing us to consider only fair paths. Our implementation seamlessly builds upon existing CTL proving techniques, one of which we introduced in Chapter 3. In addition, using a prototype implementation we demonstrate the viability of the approach in Chapter 7. As will be later discussed in Chapter 5, in future work we hope to eliminate the limitations of the strategy introduced in Chapter 5 by utilizing the technique introduced in this chapter to allow for Vardi & Wolper's automata-theoretic technique for LTL verification [VW94]. This can potentially eliminate the incompleteness introduced in Chapter 5 for CTL* verification. That is, a seamless integration between LTL and CTL reasoning may make way for an alternative relatively complete CTL* strategy.

Chapter 5

Automation of CTL* Verification for Infinite-State System

In this chapter, we introduce the first known fully automated tool for symbolically proving CTL* properties of (infinite-state) integer programs. The method uses an internal encoding which facilitates reasoning about the subtle interplay between the nesting of path and state temporal operators that occurs within CTL* proofs. A precondition synthesis strategy is then used over a program transformation that trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. We show the viability of our approach in practice using examples drawn from device drivers and various industrial examples further on in Chapter 7.

5.1 Introduction

As discussed in the previous chapters, indeed 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". As discussed in Chapter 4, some of these deficiencies can be mitigated by considering fairness for branching-time logic (fair-CTL), 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. Fair-CTL thus cannot be generalized to model all trace-based properties. 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 chapter 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 chapter 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.

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. Based on our approach, we have developed a tool capable of automatically proving properties of programs that no tool could previously fully automate. In Chapter 7, we 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.

5.1.1 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 later 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. For the sake of clarity, we will first demonstrate our technical contributions of exclusively verifying CTL*, followed by further chapters demonstrating how we can extend our algorithm to support this part of CTL^*_{lp}.

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 [MBCC07]. 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 nontermination 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 [McM06], termination [PR04c, CSZ13], nontermination [GHM+08], and CTL [BPR13, CKP14] that are not complete. Furthermore, our determinization procedure is not complete. We will address this issue in later sections.

5.1.2 Related Work

There are various algorithms for model checking CTL* for finite-state programs and other decidable settings. The approach of Emerson et al. [EL87] 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 [GV04], where a μ-calculus model checker is invoked after the translation. The approach described in [CGP99] 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. One can think of our approach as an implementation of the technique described in [CGP99], but over infinite-state programs.

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 [EJ99], one can conceive that the approach in [BCPR14] could potentially solve CTL* model checking if the latter were reduced to solving parity games by combining [GV04] and [EJ99]. However, we note that the resulting infinite-state game would integrate the (first-order to 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, [BCPR14] requires a manual instantiation of the structure of assertions, characterizing subsets of the infinite-state game, that are to be found by their tool. We generalize an approach introduced by [CK13], 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., [Bod04, CGP+07, CK11, CK13, BPR13, CKP14, ST12]. 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 [CK11] 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, [CK13] 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.

5.2 Approach Overview and Example

5.2.1 Overview

In this section, we present a quick overview of our CTL* verification procedure ProveCTL*, presented in Alg. 13 and Alg. 12 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. Note that for our CTL* algorithms, we assume that all paths are infinite for the sake of brevity and clarity. As previously discussed in Chapter 4, we assume that a transformation has been applied to distinguish between maximal (finite) paths that occur in P and those introduced by our transformations below.

Branching-Relations: We first define a key concept crucial to understanding our CTL* algorithm. 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 2.2 in Chapter 2, 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 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.

The algorithm is based on the procedures below, which are defined in later sections of this chapter:

Approximate is a procedure that performs a syntactic conversion from a path formula to its corresponding over-approximated universal CTL formula (ACTL).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 [AL91]. 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 formula. When

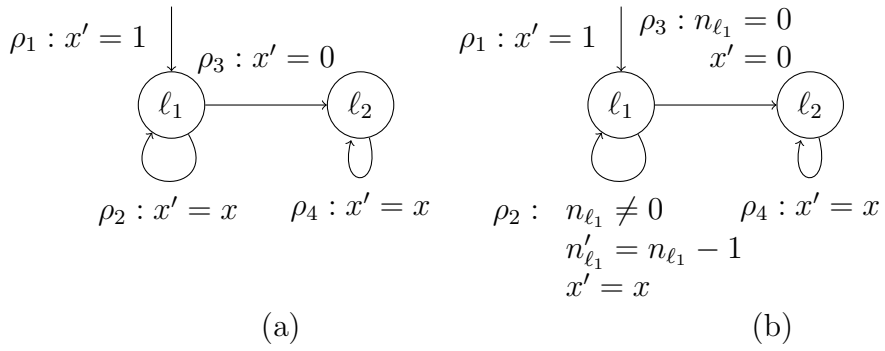


Figure 5.1: (a) The control-flow graph of a program for which we wish to prove the CTL* property EFG $x = 1$. (b) The control-flow graph after calling DETERMINIZE, it includes the prophecy variable n_{ℓ_1} corresponding to the nondeterministic branching-relation (ρ_2, ρ_3).

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.

5.2.2 Example

Consider the program in Fig. 5.1(a) and the property 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 G $x = 1$ is a path formula, and thus use APPROXIMATE to return the over-approximated state formula AG $x = 1$. We then initiate a CTL model checking task where we seek a set of states a_{G} such that EFa_{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 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 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. 5.1(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.

5.3 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.

5.3.1 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. 10, 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.

ALGORITHM 10: 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
```

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 $G = \textsc{MinSCS}(P, C, \ell)$ and $G \neq \emptyset$ on lines 7 and 8. If from some location ℓ, where $G = \textsc{MinSCS}(P, C, \ell)$, there is an edge to ℓ' such that $\textsc{MinSCS}(P, C, \ell')$ 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. Recall that there are numerous ways in which an SCS can be segmented, thus a particular choice may impact the accuracy of our determinzation algorithm. That is, some segmentations may lead to a more precise determinzation of the program, and thus a more accurate and efficient verification result. Although this may be the case theoretically, that is not the case in practice. Due to the nature of the procedural programs that we analyze, only one choice will ever be identified. Furthermore, the choice of the SCS would not affect the correctness of our verification algorithm.

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 $\ell \in \textsc{Synth}$, 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 $\textsc{MinSCS}(P, C, \ell)$, we constrain ρ by requiring that $n_\ell > 0$, and then decrement n_ℓ. If ℓ' is not contained within $\textsc{MinSCS}(P, C, \ell)$, 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 5.1. *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 $G_j = \textsc{MinSCS}(P, C, \ell_j)$, 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.

□

5.3.2 Approximation

In Alg. 11, 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.

ALGORITHM 11: Approximate produces a syntactic conversion from a path formula to its corresponding over-approximation in ACTL.

```
1 Let Approximate(ψ, a_θ'1, a_θ'2) : φ =
2     match (ψ) with
3         Fθ'1 → return AFa_θ'1
4         Gθ'1 → return AGa_θ'1
5         Xθ'1 → return AXa_θ'1
6         θ'1 W θ'2 → return Aa_θ'1 W a_θ'2
7         θ'1 U θ'2 → return Aa_θ'1 U a_θ'2
8         θ'1 ∧ θ'2 → return a_θ'1 ∧ a_θ'2
9         θ'1 ∨ θ'2 → return a_θ'1 ∨ a_θ'2
```

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.

Theorem 5.2. *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 predicates and Boolean combinations of simpler formulae, the proof is immediate.

- Suppose that $\psi = G\theta_1$. Then, Approximate$(\psi, a_{\theta_1}, a_{\theta_2})$ is $AG(a_{\theta_1})$. Suppose that $s \models$ Approximate$(\psi, a_{\theta_1}, a_{\theta_2})$ but $s \not\models A\psi$. Then, there is a path π starting in s such that π does not satisfy $G\theta_1$. 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, $\textsc{Approximate}(\psi, a_{\theta_1}, a_{\theta_2})$ is $\mathsf{AF}(a_{\theta_1})$. Suppose that $s \models \textsc{Approximate}(\psi, a_{\theta_1}, a_{\theta_2})$ 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 $s \models \textsc{Approximate}(\psi, a_{\theta_1}, a_{\theta_2})$, 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 5.2 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.

5.3.3 CTL* Verification Procedure

ALGORITHM 12: 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
```

In this section, we present our main CTL* verification procedure, ProveCTL*. Alg. 12 depicts Verify, which wraps the main procedure ProveCTL*, shown in Alg. 13. 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, 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. 13. 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

ALGORITHM 13: 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)
```

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 predicate α is the predicate itself, as shown on line 13. The precondition is then utilized in the remaining actions of the algorithm.

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 [BPR13] and [CKP14]. 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.

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 predicate α is the predicate 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.

Quantifier Elimination for Satisfying Preconditions

ALGORITHM 14: 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)
```

The procedure QUANTELIM, presented in Alg. 14, 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. 14. 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. Recall that for our CTL* algorithms, we assumed that all paths are infinite for the sake of brevity and clarity. Quantification thus must be restricted to states for which there does exist a prophecy value leading to infinite paths. For example, consider Fig. 5.1(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. Note that this solution is simpler than that proposed in Chapter 4, as terminating paths necessitate fairness, requiring further consideration of finite paths.

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 5.3. *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 sub-formulae of ψ. By induction on the structure of ψ and repeated use of Theorem 5.2 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 5.1 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 sub-formulae of ψ. By induction on the structure of ψ and repeated use of Theorem 5.2 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 5.3 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.

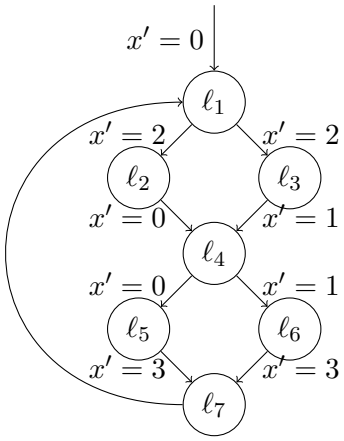


Figure 5.2: Program for which determinization is insufficient.

5.4 (In)completeness of Determinization

Given that our determinization technique has been adopted and repurposed from a similar symbolic determinization technique introduced in [CK11] 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. 5.2 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 sub-formulae 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. 5.2. 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.

5.4.1 Towards completeness of CTL*

This above example 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 [VW86] 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.

Recall that our technique introduced in Chapter 4 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. As an example, we refer back to Chapter 4 Fig. 4.2. If we were to apply our CTL* methodology, as fair-CTL is indeed a subset of CTL*, the performance of our model-checker would suffer. Consider the transformation applied in Fig. 4.2(b), now consider if we were to instead apply our CTL* transformation:

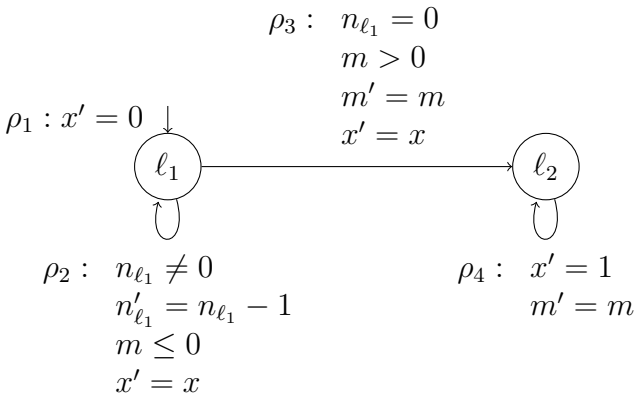


Figure 5.3: Verifying a control-flow graph of a program with the fair-CTL property $\mathsf{AG}(x = 0 \Rightarrow \mathsf{AF}(x = 1))$ and the fairness constraint $\mathsf{GF}\ \rho_2 \Rightarrow \mathsf{GF}\ m > 0$ utilizing the ProveCTL* methodology.

These transformations significantly differ, as Fig. 4.2(b) demonstrates a property-dependent instrumentation, whilst Fig. 5.3 a property-independent one. That is, the transformation in ProveCTL* only attempts to determinize the paths in a given program, regardless of the CTL* property or the fairness constraint at hand. The transformation in Chapter 4 aims to identify non-deterministic decisions which repeat infinitely often based on a given fairness constraint. Thus at each transition in Fig. 4.2(b), the choice $\{\ (\neg\rho_2\ \wedge\ n' \leq n)\ \vee\ (\rho_2\ \wedge\ n' < n)\ \vee\ m > 0\ \}\ \wedge\ n \geq 0$ is given. The transformation in ProveCTL* only attempts to distinguish all possible non-deterministic paths within a generally identified SCS, thus in Fig. 5.3, the path partitioning solely relies on whether $n_{\ell_1} \neq 0$ or $n_{\ell_1} = 0$. The counterexamples and preconditions produced from the former transformation are clearly more conducive to the verification of a fair-CTL property relative to a general SCS transformation, as they are guided by a specific property.

If a fairness constraint corresponds to the structure of a program's SCS, then ProveCTL* could be used to verify a fair-CTL property with a likelihood of performance loss. However, the incompleteness of ProveCTL*, as discussed in 5.4, could pose further issues. Indeed, ProveCTL* may not always be able to verify a fair-CTL property given an incompatible pro-

gram structure relative to the given fairness constraint. Consider the structure of the following program:

```
1  bool b;
2  while (nondet()) {
3      if (nondet()) {
4          b = true;
5          ...;
6      } else {
7          b = false;
8          ...;
9     }
10    ...;
11 }
```

For the fairness property $\Omega = \{(b = \textsc{True}, b = \textsc{False})\}$, and a given CTL formula, our determinization in ProveCTL* may not be able to sufficiently isolate the paths that satisfy Ω in order to prove the fair-CTL property. This is a result of the fairness constraint not corresponding explicitly to the program's SCS or structure. Our technique in Chapter 4 should thus be considered as part of a prospective automata theoretic approach where determinization is indeed tailored for the formula to be verified. We hope to further examine both the viability and practicality of such an extension.

5.5 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.

As previously discussed, we hope to eliminate the limitations of our determinization procedure by potentially utilizing the technique introduced in Chapter 4 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* alone cannot express this without the support of first-order quantification. There does exist one automated model checking tool that supports first-order CTL [BBR14]. 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* to include first-order logic.

Chapter 6

Remembrance of Things Past

In this chapter, 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 over our CTL^* technique introduced in Chapter 5.

6.1 Introduction

Past-time connectives are known to make the formulation of temporal logic specifications exponentially more intuitive and succinct [LPZ85], thus in this chapter we choose to support a past-connective extension to CTL^*, maintaining the effectiveness of the intuitiveness of our specification and verification platform. As noted earlier, 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 [GPSS80, KPV12] [1]. One thus may speculate that no "extension" per se is necessary, as an automated translation from CTL^*_{lp} to CTL^* is a more suitable strategy than embedding the additional history variables to be proposed, 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 [LS95, KPV12]. The known lower bounds for conversion from temporal logic with past to temporal logic without past are expressible in the fragment we consider [LS95]. 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.

[1] Note that 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 chapter.

We are not aware of any implementations of the (non-elementary) translation from LTL with past, to LTL (and clearly none for CTL^*_{lp}).

6.1.1 Approach and Contribution

In this chapter we thus 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.

Based on our approach, we have extended our tool to automatically prove CTL^*_{lp} properties of programs that no tool could previously fully automate, as will be later demonstrating in Chapter 7.

6.2 $\mathbf{CTL}^*_{lp}$– Adding Past to $\mathbf{CTL}^*$

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 in Chapter 2 Section 2.3.2. 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 6.3. We additionally discuss the challenges of extending to full CTL^*_{lp} in Section 6.2.2, and further exhibit the usefulness of our CTL^*_{lp} extension via a case study provided in Section 6.3.1.

6.2.1 Checking $\mathbf{CTL}^*_{lp}$ Formulae

In this section, we describe the details of our CTL^*_{lp} model checking procedure ProveCTL$^*_{lp}$ presented in Alg. 15. 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 explosions, even with limited temporal height [LS95, KPV12]. We stress again that the lower

ALGORITHM 15: Extending our recursive CTL^* verification procedure to support CTL^*_{lp}.

Let rec $\textsc{ProveCTL}^*_{lp}$ *(θ, P, P_D) : (formula, bool, program, program)* $=$
  $(\mathcal{L}, E, \mathsf{Vars}) = P$
  match *(θ)* **with**
    $\varphi : stateformula \rightarrow$
      match *(φ)* **with**
        $\theta'_1 \vee \theta'_2 \mid \theta'_1 \wedge \theta'_2 \mid \mathsf{A}\theta'_1 \circ^{\pm} \theta'_2 \mid \mathsf{E}\theta'_1 \circ^{\pm} \theta'_2 \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta'_1, P, P_D)$
          $(a_{\theta'_2}, \textsc{Path}_2, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta'_2, P, P_D)$
        $\mathsf{A}\circ^{\pm}\theta' \mid \mathsf{E}\circ^{\pm}\theta' \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta', P, P_D)$
          $(a_{\theta'_2}, \textsc{Path}_2) = (\textsc{False}, \textsc{False})$
      match *(φ)* **with**
        $\alpha \rightarrow a_\theta = \alpha$;
        $\theta'_1 \vee \theta'_2 \mid \theta'_1 \wedge \theta'_2 \mid \mathsf{A}\circ\theta' \mid \mathsf{E}\circ\theta' \mid \mathsf{A}\theta'_1 \circ \theta'_2 \mid \mathsf{E}\theta'_1 \circ \theta'_2 \rightarrow$
          $\varphi' = \textsc{Replace}(\varphi, a_{\theta'_1}, a_{\theta'_2})$
          if $\textsc{Path}_1 \vee \textsc{Path}_2$ **then**
            $a_\theta = \textsc{QuantElim}(\mathrm{CTL}(P_D, \varphi'), \varphi)$
          else
            $a_\theta = \mathrm{CTL}(P, \varphi')$
        $\mathsf{A}\circ^{-1}\theta' \mid \mathsf{E}\circ^{-1}\theta' \mid \mathsf{A}\theta'_1 \circ^{-1} \theta'_2 \mid \mathsf{E}\theta'_1 \circ^{-1} \theta'_2 \rightarrow$
          $(a_\theta, P, P_D) = \textsc{AddHistory}(\varphi, \circ^{-1}, a_{\theta'_1}, a_{\theta'_2}, P, P_D)$
      $\textsc{Path} = \textsc{False}$
    $\psi : pathformula \rightarrow$
      match *(ψ)* **with**
        $\theta'_1 \vee \theta'_2 \mid \theta'_1 \wedge \theta'_2 \mid \theta'_1 \circ^{\pm} \theta'_2 \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta'_1, P, P_D)$
          $(a_{\theta'_2}, \textsc{Path}_2, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta'_2, P, P_D)$
        $\circ^{\pm}\theta' \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1, P, P_D) = \textsc{ProveCTL}^*_{lp}(\theta', P, P_D)$
          $(a_{\theta'_2}, \textsc{Path}_2) = (\textsc{False}, \textsc{False})$
      match *(ψ)* **with**
        $\theta'_1 \vee \theta'_2 \mid \theta'_1 \wedge \theta'_2 \rightarrow$
          $a_\theta = \textsc{Approximate}(\psi, a_{\theta'_1}, a_{\theta'_2})$
          $\textsc{Path} = \textsc{Path}_1 \vee \textsc{Path}_2$
        $\theta'_1 \circ \theta'_2 \mid \circ\theta' \rightarrow$
          $\psi' = \textsc{Approximate}(\psi, a_{\theta'_1}, a_{\theta'_2})$
          $a_\theta = \mathrm{CTL}(P_D, \psi')$
          $\textsc{Path} = \textsc{True}$
        $\theta'_1 \circ^{-1} \theta'_2 \mid \circ^{-1}\theta' \rightarrow$
          $(a_\theta, P, P_D) = \textsc{AddHistory}(\psi, \circ^{-1}, a_{\theta'_1}, a_{\theta'_2}, P, P_D)$
          $\textsc{Path} = \textsc{False}$
  return $(a_\theta, \textsc{Path}, P, P_D)$

ALGORITHM 16: AddHistory produces history variables corresponding to past-connectives in CTL^*_{lp}.

```
1  Let AddHistory(ψ, ∘^{-1}, a_{θ1}, a_{θ2}, P, P_D) : formula, past-operator, program, program =
2      a_θ = H_ψ
3      match (∘^{-1}) with
4          F^{-1} →
5              ι = (H'_ψ = a'_{θ1}); ρ_h = (H'_ψ = H_ψ ∨ a'_{θ1})
6          G^{-1} →
7              ι = (H'_ψ = a'_{θ1}); ρ_h = (H'_ψ = H_ψ ∧ a'_{θ1})
8          X^{-1} →
9              ι = (H'_ψ = False); ρ_h = (H'_ψ = a_{θ1})
10         W^{-1} →
11             ι = (H'_ψ = a'_{θ1} ∨ a'_{θ2}); ρ_h = (H'_ψ = (H_ψ ∧ a'_{θ1}) ∨ a'_{θ2}))
12         U^{-1} →
13             ι = (H'_ψ = a'_{θ2}); ρ_h = (H'_ψ = (H_ψ ∧ a'_{θ1}) ∨ a'_{θ2})
14     P = InstrumentHistory(H_ψ, ι, ρ_h, P)
15     P_D = InstrumentHistory(H_ψ, ι, ρ_h, P_D)
16     return (a_θ, P, P_D)
```

ALGORITHM 17: InstrumentHistory embeds conditions over history variables within a transition system P.

```
1  Let InstrumentHistory(H, ι, ρ_h, P) : program =
2      (L, E, Vars) = P
3      foreach (ℓ, ρ, ℓ') ∈ E do
4          if ℓ = ℓ_I then
5              ρ = ρ ∧ ι
6          else
7              ρ = ρ ∧ ρ_h
8      Vars = Vars ∪ {H}
9      return P
```

bound in [LS95] can be expressed in our fragment. We thus introduce a procedure that extends ProveCTL* in Chapter 5 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.

AddHistory & InstrumentHistory

In Alg. 16, 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.[2] 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.

- 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

[2]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.

in terms of variables of P and can be instrumented into both P and P_D. As demonstrated in Alg. 17, 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. 16, we return the transformed programs P and P_D alongside the history variable H_ψ as an atomic predicate, 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 $(\_, \hat{P}', \_) =$ 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. 16.

  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.* 16. 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. 16). 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.

ProveCTL$^*_{lp}$

We return to our main algorithm in Alg. 15. 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 $\textsc{Path}_1$ and $\textsc{Path}_2$. Accordingly we set Path to their disjunction on line 34.

Theorem 6.3. *If* VERIFY(θ, P) *returns true for a program* $P = (\mathcal{L}, E, \mathit{Vars})$ *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 5.2 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 5.1 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 5.2 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 5.1 π' is a computation of P. By Theorems 5.2 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.

□

6.2.2 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. 5.1(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 [LPZ85]. 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. 5.1(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.

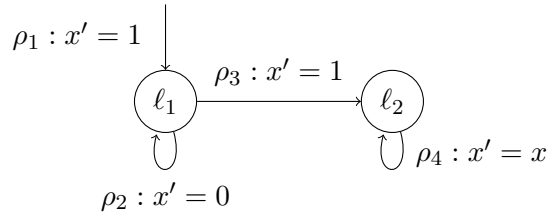
6.3 Demonstrating CTL^*_{lp}

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. 6.1(a) and the property $\mathsf{EGFG}^{-1}\ x = 1$ stating that there exists some path such that infinitely often there is a state in which $x = 1$ has always held in the past (note that this formula is equivalent to $\mathsf{EG}\ x = 1$ in the initial state). 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 $\mathsf{AFG}^{-1}\ x = 1$ does not hold.

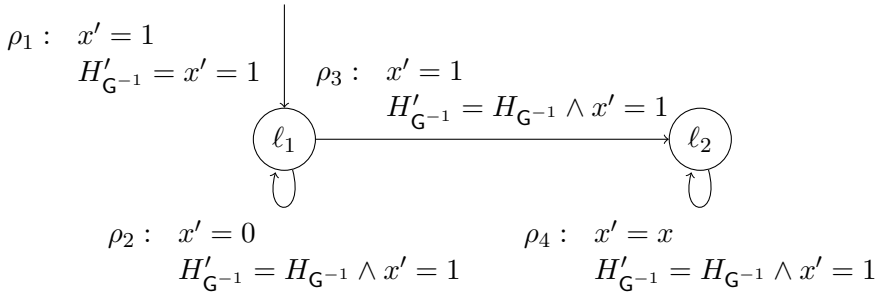
As discussed, in order to check this property we recursively handle its sub-formulae. The most inner sub-formula is a past sub-formula, namely $\mathsf{G}^{-1}\ x = 1$. We call the function AddHistory with the precondition $x = 1$, given that the precondition of an atomic predicate is the atomic predicate 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. 6.1(b), we show the history variable $H_{\mathsf{G}^{-1}}$ introduced in P, and in Fig. 6.1(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.

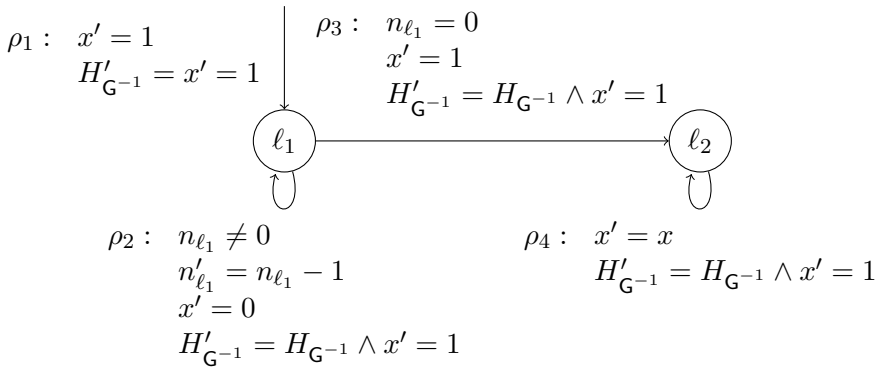
As before, P_D is a new symbolically partially-determinized program in which a newly introduced prophecy variable, namely n_{ℓ_1} in Fig. 6.1(c), is associated with the branching-relation (ρ_2, ρ_3), and is used to make predictions about the occurrences of relations ρ_2 and ρ_3. As can be



(a)



(b)



(c)

Figure 6.1: (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, and then AddHistory, it includes the prophecy variable n_{ℓ_1}, corresponding to the nondeterministic branching-relation (ρ_2, ρ_3)

seen in Fig. 6.1(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 F $(H_{\mathsf{G}^{-1}})$ holds by verifying the determinized program, denoted by P_D, using the over-approximated CTL formula 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 F $H_{\mathsf{G}^{-1}}$ by a_{F} and finally arrive at our outermost CTL^*_{lp} formula EG a_{F}. As dictated by our ProveCTL$^*_{lp}$ algorithm, our final step is to verify EG a_{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 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. 6.1(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 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 satisfy the path formula then clearly, these paths give us a sufficient proof to conclude that 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.

6.3.1 Case Study

We report on a case study that requires the application of our extended ProveCTL$^*_{lp}$ algorithm presented in Alg. 15. 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. 6.2, 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:

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

```
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;
```

Figure 6.2: A Windows Device Driver driver setting a *Cancel* routine for an I/O request packet.

We thus call ProveCTL$^*_{lp}$ with the property above, the program in Fig. 6.2, which we will denote as P, and a determinized variation (P_D) attained from the Determinize algorithm previously discussed in Alg. 10. Supplementary information regarding how we interpret and parse a program's commands to attain P can be found in [BCI+16]. 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 AddHis-

TORY algorithm in Alg. 16. Given that `InsertTailList()` and `IoMarkIrpPending()` are call sites, they serve as the atomic predicates a_{θ_1} and a_{θ_2}, respectively. Our sub-formula is then matched on line 12 in Alg. 16 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 instrumented 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()` $\Rightarrow$ 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 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 [BPR13] and our technique introduced in Chapter 3. 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. 6.2.

6.4 Concluding Remarks

We have introduced the first-known fully automatic method capable of proving CTL^*_{lp} properties for infinite-state (integer) programs. We provide a novel methodology which extends our CTL^* procedure in Chapter 5 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.

As discussed in Chapter 5, eliminating the limitations of our determinization procedure in the future perhaps through techniques introduced in Chapter 4 would also allow us to extend our algorithm to handle full CTL^*_{lp}. Indeed, in our current technique it is not possible to quantify prophecy variables nested within history variables. However, this would not be a requirement if our determinization is not utilized, providing potential for a full CTL^*_{lp} technique.

Chapter 7

Implementation and Benchmarks

In this Chapter, we present the open-source tool T2, upon which all of our implementations of Chapters 3–6 have been made. We demonstrate T2 in order to support the automatic verification of CTL, fair-CTL, CTL*, and CTL$^*_{lp}$ for software systems. As input can be provided in a native format and in C, via the support of the LLVM compiler framework, we use benchmarks derived from the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the SoftUpdates patch system. We briefly discuss T2's architecture, its underlying techniques, and conclude with an experimental illustration of its competitiveness and directions for future extensions.

7.1 Introduction

In this chapter, we describe T2 (*a.k.a* TERMINATOR 2), an open-source framework that implements, combines, and extends research techniques developed in Chapters 3–6, in addition to other research techniques [BCF13, CSZ13] aimed towards the unification of a verification system of temporal properties for software systems. T2 is a fully automated tool that operates on an input format that can be automatically extracted from the LLVM compiler framework's intermediate representation. This in turn allows T2 to analyze programs in a wide range of programming languages including C programs, and potentially any broader set of languages based on LLVM (e.g., C++, Objective C, and Swift), against a user-given temporal property. As T2 includes contributions from other research methodologies, in this chapter we will indeed demonstrate all features of T2, yet explicitly highlight features corresponding to this dissertation. T2 allows users to:

1. Utilize state-of-the-art termination techniques, shown to be highly competitive with existing termination tools, in addition to various and recent techniques for proving safety

and nontermination of infinite-state C programs. Details with regards to these features can be found in [BCF13, CSZ13].

2. Verify CTL properties through synthesizing preconditions asserting the satisfaction of CTL input property. CTL subsumes reasoning about safety, termination, and nontermination, in addition to expressive state-based properties via the system's interaction with inputs and nondeterminism, a capability in which linear-time temporal logics like LTL are inadequate to express. This feature is the implementation of our technique from Chapter 3.

3. Verify fair-CTL, which allows one to model trace-based assumptions about the environment both in a sequential setting, and when reasoning about concurrent environments, where fairness is used to abstract away the scheduler. This feature is the implementation of our technique from Chapter 4.

4. Utilize the first known fully automated tool for symbolically proving CTL^*_{lp} properties of infinite-state C programs. CTL^*_{lp} is capable of expressing CTL, LTL, and properties necessitating their interplay. This feature is the implementation of our techniques from Chapters 5 and 6.

We describe T2's capabilities and demonstrate its effectiveness by an experimental evaluation against competing tools. We include several key optimizations conducive to T2's performance and how they allowed the aforementioned verification techniques to come to fruition. T2's architecture is based on the combination of a reachability engine, ranking functions and recurrence-sets synthesis, and our precondition synthesis strategy introduced in Chapter 3. We close this chapter with an experimental comparison to competing tools, as well as an evaluation that demonstrates the performance improvements achieved with the various optimisations implemented. We note that T2 is implemented in F# and makes heavy use of the Z3 SMT solver [DMB08]. T2 runs on Windows, MacOS, and Linux. It is available under the free MIT license at `https://github.com/hkhlaaf/T2`.

7.1.1 Related work

We focus on tool features of T2 and consider only related publicly released tools. Generally speaking, we note that with the exception of KITTeL [FKS11], T2 is the only open-source termination prover and is the first open-source temporal property prover. When considering T2's features outside the scope of this dissertation, ARMC [PR07] and CProver [KSTW], implement a TERMINATOR-style [CPR06] incremental reduction to safety proving. T2 is distinguished from these tools by its use of lexicographic ranking functions instead of disjunctive termination arguments [CSZ13]. Other termination proving tools include FuncTion [Urb13], KITTeL [FKS11],

and Ultimate [HHP14], which synthesize termination arguments, but have weak support for inferring supporting invariants in long programs with many loops. AProVE [GBE+14] is a closed-source portfolio solver implementing many successful techniques, including T2's methods. With regards to all contributions to T2 within the scope of this dissertation, we know of only one other tool able to automatically prove CTL properties of infinite-state programs:[1] Q'ARMC [BPR13], however Q'ARMC does not provide an automated front-end to its native input and requires a manual instantiation of the structure of the invariants. We are not aware of tools other than T2 that can verify Fair-CTL and CTL^*_{lp} for such programs.

As extensively discussed in Chapter 2, T2 only supports linear integer arithmetic fragments of C. An extension of T2 that handles heap program directly is presented in [ABCK15].[2] As in many other tools, numbers are treated as mathematical integers, not machine integers. However, our C front-end provides a transformation [FKS12] that handles machine integers correctly by inserting explicit normalization steps at possible overflows.

7.2 Background

We have discussed numerous systems that have been proposed to automate the temporal verification of C programs [CK13, CK11, BPR13]. Yet It is unclear to what extent real C programs are handled given that each verification system utilizes its own intermediate language. A verification system's front-end preprocessing, or lack-thereof, can thus significantly impact its capability to be fully automated and modular. T2 indeed similarly allows input in its internal program representation to facilitate use from other tools. T2 internally represents programs as control flow graphs of program locations $\mathcal{L}$ connected by transition rules with conditions and assignments to a set of integer variables Vars, just as defined by Control Flow Graphs in Chapter 2 Section 2.2. However, it additionally supports the direct operation on C programs, through the conversion from the LLVM-IR allowing an automatic conversion of, e.g., C programs to our native input format.

In recent years, LLVM has become the standard basis of program analysis tools for C, for a multitude of reasons: A rapidly-growing number of programming languages are being supported by LLVM, T2 thus profits from this development and can potentially be used to also analyze programs written in Objective-C, Swift, ... Additionally, when compiling the LLVM-IR, LLVM implements a number of compiler optimizations that are readily applied to simplify verification analysis significantly. LLVM-IR uses Static Single Assignment (SSA) based representation that includes type information, explicit control flow graphs, and an explicit data-flow representation

[1] We do not discuss tools that only support finite-state systems or pushdown automata.

[2] Alternatively, the heap-to-integer abstractions implemented in Thor [MTLT10] for C or the one implemented in AProVE [GBE+14] for C and Java can be used as a pre-processing step.

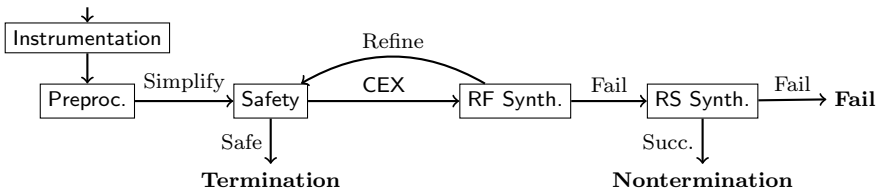


Figure 7.1: Flowchart of the T2 termination proving procedure

using an infinite typed register set, all features conducive to the generation of T2's native input format. We have thus chosen to extend llvm2kittel [FKS11], which automatically translates C programs into integer term rewriting systems using LLVM, to also generate T2's native format. Our implementation uses the existing dead code elimination, constant propagation, and control-flow simplifications to simplify the input program. Further details on how we generate the T2 native input from a C program from can be found in [BCI+16].

7.2.1 Back-end

In T2, we implement efficient safety, termination, and non-termination procedures allowing for which our techniques from Chapters 3–6 are built upon, providing scalable automated temporal logic model checking.

Proving Safety: To prove temporal properties, T2 repeatedly calls to a safety proving procedure on instrumented programs. For this, T2 implements the Impact [McM06] safety proving algorithm, and furthermore can use safety proving techniques implemented in Z3, *e.g.* generalized property directed reachability (GPDR) [HB12b] and Spacer [KGC14]. Thus, we convert our transition systems into sets of linear Horn clauses with constraints in linear arithmetic, in which one predicate p_ℓ is introduced per program location ℓ. For example, the transition from ℓ_2 to ℓ_2 in Fig. 2.1(right) is represented as $\forall \mathtt{x}, \mathtt{k}, \mathtt{x}' : \mathsf{p}_{\ell_2}(\mathtt{x}', \mathtt{k}) \leftarrow \mathsf{p}_{\ell_2}(\mathtt{x}, \mathtt{k}) \wedge \mathtt{x}' = \mathtt{x} - \mathtt{k}$.

Proving Termination: A schematic overview of the termination proving procedure is displayed in Fig. 7.1. In the initial Instrumentation phase (described in [BCF13]), the input program is modified so that a termination proof can be constructed by a sequence of alternating safety queries and rank function synthesis steps. This reduces the check of a speculated (possibly lexicographic) rank function f for a loop to an assertion that the value of f after one loop iteration is smaller than before that iteration. If the speculated termination argument is insufficient, our Safety check fails, and the returned counterexample is used to refine the termination argument in step RF Synth. Here, we follow the strategy presented in [CSZ13] to construct a lexicographic termination argument, extending a standard linear rank function synthesis procedure [PR04a].[3] The synthesis procedure is implemented as constraint solving via Z3. Note

[3] T2 can optionally also synthesize disjunctive termination arguments [PR04b] as implemented in the original TERMINATOR [CPR06].

that the overall procedure is independent of the used safety prover and rank function synthesis.

In our Preprocessing phase, a number of standard program analysis techniques are used to simplify the remaining proof. Most prominently, this includes the termination proving preprocessing technique presented in [BCF13] to remove loop transitions that we can directly prove terminating, without needing further supporting invariants. In our termination benchmarks, about 80% of program loops (*e.g.* encodings of `for i in 1 .. n do`-style loops) are eliminated at this stage.

Disproving Termination: When T2 cannot refine a termination argument based on a given counterexample, it tries to prove existence of a recurrence set [GHM+08] witnessing nontermination in the RS Synth. step. T2 uses a variation of the techniques from [BSOG], restricted to only take a counterexample execution into account and implemented as constraint solving via Z3.

7.3 Experimental Evaluation

We conclude with evaluations underlining T2's effectiveness compared to competing tools. There are currently no other known tools supporting fair-CTL and CTL* for infinite-state systems, thus we are not able to make experimental comparisons with other tools with regards to these benchmarks. We note that T2's performance has significantly improved through advancements in our back-end (e.g. by using Spacer instead of Impact), relative to the initial publications of Chapters 3–6.

7.3.1 CTL Experiments

We evaluate T2's CTL verification techniques against the only other available tool, Q'ARMC [BPR13] on the 56 benchmarks from its evaluation in Fig. 7.2 . These benchmarks are drawn from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the SoftUpdates patch system. They can be found at `http://www.cims.nyu.edu/~ejk/ctl/`. The tools were executed on a Core i7 950 CPU with a timeout of 100 seconds. Both tools are able to successfully verify all examples. T2 needs 2.7 seconds on average, whereas Q'ARMC takes 3.6 seconds. The scatterplot on the right shows how proof times compare on the individual examples. Detailed benchmarks and experimentations against [CK13] can be found in [BPR13].

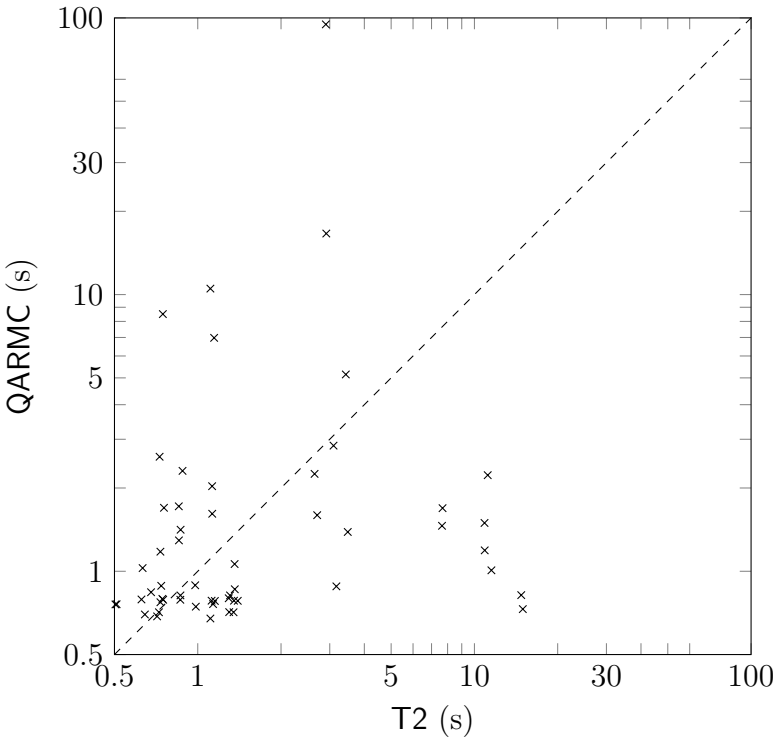


Figure 7.2: Experimental evaluations of CTL on infinite-state programs drawn from the Windows OS, and PgSQL against Q'ARMC.

7.3.2 Fair-CTL Experiments

In addition to benchmarks drawn from`http://www.cims.nyu.edu/~ejk/ctl/`, we applied our tool to several small programs: a classical mutual exclusion algorithm as well as code fragments drawn from device drivers. As previously discussed, there are currently no known tools supporting fair-CTL for infinite-state systems, thus we are not able to make experimental comparisons.

Fig. 7.3 shows experimental evaluations of sequential Windows device drivers (WDD) and various concurrent systems [4]. WDD1 uses the fairness constraint
$\mathsf{GF}(\texttt{IoCreateDevice.exit}\{1\}) \Rightarrow \mathsf{GF}(\texttt{status} = \texttt{SUCCESS})$, while WDD2 and 3 utilize the same fairness constraint in relation to checking the acquisition and release of spin locks and the entrance and exit of critical regions, respectively. WDD4 requires a weak fairness constraint indicating that `STATUS_OK` will hold a value of `true` infinitely often, that is, whenever sockets are successfully opened, the server will eventually return a successful status infinitely often.

Note that the initially concurrent programs are reduced to sequential programs via [GCPV09], which uses rely-guarantee reasoning to reduce multi-threaded verification to liveness. We verify the traditional Bakery algorithm, requiring that any thread requesting access to the critical region will eventually be granted the right to do so. The producer-consumer algorithm requires that any amount of input data produced, must be eventually consumed. The Chain benchmark

[4]Benchmarks can be found at `http://heidyk.com/experiments.html`

Program	LoC	Property	FC	Time(s)	Result
WDD1	20	AG(BlockInits() ⇒ AF UnblockInits())	Yes	14.4	✓
WDD1	20	AG(BlockInits() ⇒ AF UnblockInits())	No	2.1	χ
WDD2	374	AG(AcqSpinLock() ⇒ AF RelSpinLock())	Yes	18.8	✓
WDD2	374	AG(AcqSpinLock() ⇒ AF RelSpinLock())	No	14.1	χ
WDD3	58	AF(EnCritRegion() ⇒ EG ExCritRegion())	Yes	12.5	χ
WDD3	58	AF(EnCritRegion() ⇒ EG ExCritRegion())	No	9.6	✓
WDD4	302	AG(added_socket > 0 ⇒ AFEG STATUS_OK)	Yes	30.2	✓
WDD4	302	AG(added_socket > 0 ⇒ AFEG STATUS_OK)	No	72.4	χ
Bakery	37	AG(Noncritical ⇒ AF Critical)	Yes	2.9	✓
Bakery	37	AG(Noncritical ⇒ AF Critical)	No	16.4	χ
Prod-Cons	30	AG($p_i > 0 \Rightarrow$ AF $q_i <= 0$)	Yes	18.5	✓
Prod-Cons	30	AG($p_i > 0 \Rightarrow$ AF $q_i <= 0$)	No	5.5	χ
Chain	48	AG($x \geq 8 \Rightarrow$ AF $x = 0$)	Yes	1.8	✓
Chain	48	AG($x \geq 8 \Rightarrow$ AF $x = 0$)	No	4.7	χ

Figure 7.3: Experimental evaluations of infinite-state programs such as Windows device drivers (WDD) and concurrent systems, which were reduced to non-deterministic sequential programs via [GCPV09]. Each program is tested for both the success of a branching-time liveness property with a fairness constraint and its failure due to a lack of fairness. A ✓ represents the existence of a validity proof, while χ represents the existence of a counterexample. We denote the lines of code in our program by LoC and the fairness constraint by FC. There exist no competing tools available for comparison.

consists of a chain of threads, where each thread decreases its own counter, but the next thread in the chain can counteract, and increase the counter of the previous thread, thus only the last thread in the chain can be be decremented unconditionally. These algorithms are verified on 2, 4, and 8 threads, respectively.

For the the existential fragment of CTL, fairness constraints can often restrict the transition relations required to prove an existential property, as demonstrated by WDD3. For universal CTL properties, fairness policies can assist in enforcing properties to hold that previously did not. Thus, our tool allows us to both prove and disprove the negation of each of the properties.

7.3.3 CTL^*_{lp} Experiments

As with CTL and fair-CTL, we have drawn out a set of CTL^* problems from industrial code bases (I/O subsystems of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server). We note that these are the same set of industrial examples drawn from our CTL benchmarks, with CTL^* properties applied to them. These benchmarks were executed on an Intel x64-based 2.8 GHz single-core processor. Recall that 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",

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	✓

Figure 7.4: Experimental evaluations of infinite-state programs drawn from the Windows OS, PgSQL, and 8 toy examples. There are no competing tools available for comparison.

"OS frag. 3", "PgSQL arch 1", and "Bench 2" are verified using said properties, described in detail in Section 1.1.1. Our case study's results in Section 6.3.1, demonstrating our CTL^*_{lp} extension, is 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 [GV04] is not applicable. In Figure 7.4 we display the results of our benchmarks. As with our fair-CTL 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 property did not hold, thus 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 [CGP+07, CK11, CK13, BPR13], which can only verify significantly less expressive properties over infinite-state programs. Our tool has successfully both verified and invalidated CTL^*_{lp} 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.

7.4 Concluding Remarks

T2 is a mature platform for safety, termination, and temporal property verification. In future developments, we wish to integrate and extend its input language with native support for the heap, recursion, and concurrency. This would allow the further support LLVM-IR generated for languages such as C++, Objective-C, and Swift, which can in-turn be mapped onto its input language. Alternatively, an extension to the input language would allow us to directly parse languages like C or C++, or possibly directly supporting binaries. Indeed, such features would allow us to further verify the expressive temporal logics of fair-CTL, CTL^*, and CTL^*_{lp} beyond C programs.

Thesis Summary

In this dissertation, we automatically verified increasingly expressive temporal specifications for the undecidable general class of infinite-state programs supporting both control-sensitive and integer properties. This was achieved through introducing the first known unifying, fully automated verification system culminating to the verification of a superset logic, known as CTL^*_{lp}, of the widely accepted specification language of temporal logic.

We built our framework by introducing a novel scalable, automated, and high-performance CTL verification technique that utilizes a counterexample-guided precondition synthesis strategy. This methodology is unique to competing strategies beyond its scalability in that it allows us to implement internal encodings conducive to the verification of more expressive logics such as fair-CTL, CTL^*, and CTL^*_{lp}. We supported the verification of fair-CTL through a reduction to our CTL model checking technique via a program transformation that used infinite nondeterministic branching to symbolically partition fair from unfair executions.

For CTL^*, we proposed a method that used an internal encoding which facilitated reasoning about the subtle interplay between the nesting of path and state temporal operators that occurs within CTL^* proofs. A precondition synthesis strategy was then used over a program transformation which trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. We then proposed a linear-past extension to CTL^*, that being CTL^*_{lp}, in which the past is linear and each moment in time has a unique past. We supported this extension through the use of history variables over our CTL^* technique.

Finally, we demonstrated the fully automated implementation of our techniques, and reported our benchmarks carried out on code fragments from the PostgreSQL database server, Apache web server, Windows OS kernel, as well as smaller programs demonstrating the expressiveness of fair-CTL, CTL^*, and CTL^*_{lp} specifications. Together, these novel methodologies lead to a new class of fully automated tools capable of proving crucial properties that no tool could previously prove in the infinite-state setting.

Bibliography

[ABCK15] Aws Albargouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. *Spatial Interpolants*, pages 634–660. Springer Berlin Heidelberg, 2015.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[AO88] Krzysztof Apt and Ernst-Rüdiger Olderog. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10, 1988.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical report, Ithaca, NY, USA, 1986.

[BBC+00] Nikolaj S. Bjørner, Anca Browne, Michael A. Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Form. Methods Syst. Des.*, 16(3):227–270, 2000.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Operating Systems Review*, 40:73–85, April 2006.

[BBR14] Tewodros A. Beyene, Marc Brockschmidt, and Andrey Rybalchenko. Ctl+fo verification as constraint solving. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 101–104, New York, NY, USA, 2014. ACM.

[BCF13] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. CAV 2013, pages 413–429, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[BCI+16] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: Temporal property verification. In *Tools and Algorithms for the Construction*

and Analysis of Systems, pages 387–393, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[BCPR14] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL '14*, pages 221–233. ACM, 2014.

[Bod04] Eric Bodden. A lightweight ltl runtime verification tool for java. In *OOPSLA '04*, pages 306–307. ACM, 2004.

[Boz08] Laura Bozzelli. 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, Berlin, Heidelberg, 2008. Springer-Verlag.

[BPR13] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In *CAV'13*, pages 869–882. Springer, 2013.

[BSOG] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *FOVEOOS'11*.

[CCF+14] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. *Proving Nontermination via Safety*, pages 156–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV'02*, pages 359–364. Springer, 2002.

[CCG+05] Sagar Chaki, Edmund M. Clarke, Orna Grumberg, Joël Ouaknine, Natasha Sharygina, Tayssir Touili, and Helmut Veith. State/event software verification for branching-time specifications. In *IFM'05*, pages 53–69, 2005.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS, pages 52–71. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, pages 244–263, April 1986.

[CFKP11] Byron Cook, Jasmin Fisher, Elzbieta Krepska, and Nir Piterman. Proving stabilization of biological systems. In *VMCAI'11*, pages 134–149. Springer, 2011.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855, pages 154–169. Springer, 2000.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. 1999.

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

[CK11] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *POPL'11*, pages 399–410. ACM, 2011.

[CK13] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *PLDI'13*, pages 219–230. ACM, 2013.

[CKP14] Byron Cook, Heidy Khlaaf, and Nir Piterman. Faster temporal reasoning for infinite-state programs. In *FMCAD '14*, pages 16:75–16:82. FMCAD Inc, 2014.

[CKP15a] Byron Cook, Heidy Khlaaf, and Nir Piterman. Fairness for infinite-state systems. In *TACAS '15*, pages 384–398. Springer Berlin Heidelberg, 2015.

[CKP15b] Byron Cook, Heidy Khlaaf, and Nir Piterman. *On Automation of CTL* Verification for Infinite-State Systems*, pages 13–29. Springer, Cham, 2015.

[CKP17] Byron Cook, Heidy Khlaaf, and Nir Piterman. Verifying increasingly expressive temporal logics for infinite-state systems. *J. ACM*, 64(2):15:1–15:39, 2017.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI'06*, pages 415–426. ACM, 2006.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CSZ13] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS'13*, LNCS, pages 47–61. Springer, 2013.

[DHLP06] Alexandre David, John Håkansson, Kim G. Larsen, and Paul Pettersson. Model checking timed automata with priorities using dbm subtraction. In *FORMATS'06*, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never"; revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[EJ99] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, 1999.

[EKS03] Javier Esparza, Antonín Kucera, and Stefan Schwoon. Model checking ltl with regular valuations for pushdown systems. *Information and Computation*, 186:355–376, November 2003.

[EL86] E.A. Emerson and C.-L. Lei. Temporal reasoning under generalized fairness constraints. In *3rd Annual Symposium on Theoretical Aspects of Computer Science*, volume 210 of *LNCS*, pages 21–36. Springer, 1986.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[ES84] E. Allen Emerson and A. Prasad Sistla. Deciding branching time logic. In *STOC '84*, pages 14–24. ACM, 1984.

[FKS11] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, pages 41–50, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FKS12] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 261–277, Berlin, Heidelberg, 2012. Springer-Verlag.

[GBE+14] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. *Proving Termination of Programs Automatically with AProVE*, pages 184–191. Springer International Publishing, Cham, 2014.

[GCPV09] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL'09*, pages 16–28. ACM, 2009.

[GHM+08] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. *SIGPLAN Not.*, 43:147–158, January 2008.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM.

[GT00] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *Recent Advances in AI Planning*, volume 1809 of *LNCS*, pages 1–20. Springer, 2000.

[GV04] Alain Griffault and Aymeric Vincent. The Mec 5 model-checker. In *CAV'04*, LNCS, pages 488–491. springer, 2004.

[GWC06] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *CAV'06*, pages 170–174. Springer, 2006.

[Har86] David Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes and fairness. *Journal of the ACM*, 33:224–248, 1986.

[HB12a] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[HB12b] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 36–52, Berlin, Heidelberg, 2013. Springer-Verlag.

[HHP14] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. *Termination Analysis by Learning Terminating Programs*, pages 797–813. Springer International Publishing, Cham, 2014.

[Kam68] Johan Kamp. Tense logic and the theory of linear order. 1968.

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Conference on Computer Aided Verification*, pages 17–34. Springer-Verlag New York, Inc., 2014.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Form. Methods Syst. Des.*, 48(3):175–205, 2016.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 52–66, London, UK, UK, 2002. Springer-Verlag.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 585–591, Berlin, Heidelberg, 2011. Springer-Verlag.

[KP05] Yonit Kesten and Amir Pnueli. A compositional approach to ctl* verification. *Theor. Comput. Sci.*, 331(2-3):397–428, 2005.

[KPV12] Orna Kupferman, Amir Pnueli, and Moshe Y. Vardi. Once and for all. *J. Comput. Syst. Sci.*, 78:981–996, 2012.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[KSTW] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *22nd International Conference on Computer Aided Verification*, pages 89–103, Berlin, Heidelberg. Springer-Verlag.

[Lam80] Leslie Lamport. "sometime" is sometimes "not never": On the temporal logic of programs. In *POPL '80*, pages 174–185. ACM, 1980.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, UK, 1985. Springer-Verlag.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. In *Selected Papers of the Eleventh Symposium on Theoretical Aspects of Computer Science*, STACS '94, pages 303–324, Amsterdam, The Netherlands, The Netherlands, 1995. Elsevier Science Publishers B. V.

[MBCC07] Stephen Magill, Josh Berdine, Edmund M. Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *SAS'07*, pages 419–436. Springer, 2007.

[McM06] Kenneth Lauchlin McMillan. Lazy abstraction with interpolants. In *CAV'06*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[Mon10] David Monniaux. Quantifier elimination by lazy model enumeration. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 585–599. Springer-Verlag, 2010.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*, volume 2. Springer, 1995.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL'10*, pages 211–222. ACM, 2010.

[PMT02] H. Peng, Y. Mokhtari, and S. Tahar. Environment synthesis for compositional model checking. In *Computer Design: VLSI in Computers and Processors*, pages 70–75, 2002.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PPR05] Amir Pnueli, Andreas Podelski, and Andrey Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 124–139, Berlin, Heidelberg, 2005. Springer-Verlag.

[PR04a] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937, pages 239–251. Springer, 2004.

[PR04b] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS'04*, pages 32–41. IEEE, 2004.

[PR04c] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[PR07] Andreas Podelski and Andrey Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, pages 245–259, Berlin, Heidelberg, 2007. Springer-Verlag.

[Pri57] Arthur Prior. Time and modality. 1957.

[PS08] Amir Pnueli and Yaniv Sa'ar. All you need is compassion. In *9th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.

[Rey01] Mark Reynolds. An axiomatization of full computation tree logic. *The Journal of Symbolic Logic*, 66(3):pp. 1011–1057, 2001.

[ST12] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In *TACAS'12*, pages 607–610. ACM, 2012.

[Urb13] Caterina Urban. *The Abstract Domain of Segmented Ranking Functions*, pages 43–62. Springer Berlin Heidelberg, 2013.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[Wah] Thomas Wahl. Notions of fairness and liveness.