Abstract Interpretation of CTL Properties

Caterina Urban(✉), Samuel Ueltschi, and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
caterina.urban@inf.ethz.ch

Abstract. CTL is a temporal logic commonly used to express program properties. Most of the existing approaches for proving CTL properties only support certain classes of programs, limit their scope to a subset of CTL, or do not directly support certain existential CTL formulas. This paper presents an abstract interpretation framework for proving CTL properties that does not suffer from these limitations. Our approach automatically infers sufficient preconditions, and thus provides useful information even when a program satisfies a property only for some inputs. We systematically derive a program semantics that precisely captures CTL properties by abstraction of the operational trace semantics of a program. We then leverage existing abstract domains based on piecewise-defined functions to derive decidable abstractions that are suitable for static program analysis. To handle existential CTL properties, we augment these abstract domains with under-approximating operators. We implemented our approach in a prototype static analyzer. Our experimental evaluation demonstrates that the analysis is effective, even for CTL formulas with non-trivial nesting of universal and existential path quantifiers, and performs well on a wide variety of benchmarks.

1 Introduction

Computation tree logic (CTL) [6] is a temporal logic introduced by Clarke and Emerson to overcome certain limitations of linear temporal logic (LTL) [33] for program specification purposes. Most of the existing approaches for proving program properties expressed in CTL have limitations that restrict their applicability: they are limited to finite-state programs [7] or to certain classes of infinite-state programs (e.g., pushdown systems [36]), they limit their scope to a subset of CTL (e.g., the universal fragment of CTL [11]), or support existential path quantifiers only indirectly by considering their universal dual [8].

In this paper, we propose a new static analysis method for proving CTL properties that does not suffer from any of these limitations. We set our work in the framework of *abstract interpretation* [16], a general theory of semantic approximation that provides a basis for various successful industrial-scale tools

```
while ^1( rand() ) {
  ^2x := 1
  ^3n := rand()
  while ^4( n > 0 ) { ^5n := n − 1 }
  ^6x := 0
}
while ^7( true ) {}^8
```

Fig. 1. Standard lock acquire/release-style program [12], where rand() is a random number generation function. Assignments x := 1 and x := 0 are acting as acquire and release, respectively. We want to prove the CTL property $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$ expressing that whenever a lock is acquired ($\mathrm{x} = 1$) it is eventually released ($\mathrm{x} = 0$). We assume that initially $\mathrm{x} = 0$.

(e.g., Astrée [3]). We generalize an existing abstract interpretation framework for proving termination [18] and other liveness properties [41].

Following the theory of abstract interpretation [14], we abstract away from irrelevant details about the execution of a program and systematically derive a program semantics that is *sound and complete* for proving a CTL property. The semantics is a function defined over the programs states that satisfy the CTL formula. The value of the semantics for a CTL formula that expresses a liveness property (e.g., $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \phi)$) gives an upper bound on the number of program execution steps needed to reach a desirable state (i.e., a state satisfying ϕ for $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \phi)$). The semantics for any other CTL formula is the constant function equal to zero over its domain. We define the semantics inductively on the structure of a CTL formula, and we express it in a constructive fixpoint form starting from the functions defined for its sub-formulas.

Further sound abstractions suitable for static program analysis are derived by *fixpoint approximation* [14]. We leverage existing numerical abstract domains based on piecewise-defined functions [39], which we augment with novel under-approximating operators to directly handle existential CTL formulas. The piecewise-defined function for a CTL formula is automatically inferred through *backward analysis* by building upon the piecewise-defined functions for its sub-formulas. It over-approximates the value of the corresponding concrete semantics and, by under-approximating its domain of definition, yields a *sufficient precondition* for the CTL property. We prove the soundness of the analysis, meaning that all program executions respecting the inferred precondition indeed satisfy the CTL property. A program execution that does not respect the precondition might or might not satisfy the property.

To briefly illustrate our approach, let us consider the acquire/release-style program shown in Fig. 1, and the CTL formula $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$. The analysis begins from the atomic propositions $\mathrm{x} = 1$ and $\mathrm{x} = 0$ and, for each program control point, it infers a piecewise-defined function that is only defined when x is one or zero, respectively. It then continues to the sub-formula

$\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$ for which, building upon the function obtained for $\mathrm{x} = 0$, it infers the following interesting function at program point 4:

$$\lambda \mathrm{x}.\lambda \mathrm{n}. \begin{cases} 0 & \mathrm{x} = 0 \\ 2 & \mathrm{x} \neq 0 \wedge \mathrm{n} \leq 0 \\ 2\mathrm{n} + 2 & \text{otherwise} \end{cases} \tag{1.1}$$

The function indicates that the sub-formula $\mathrm{x} = 0$ is either satisfied trivially (when x is already zero), or in at most 2 program execution steps when $\mathrm{n} \leq 0$ (and thus the loop at program point 4 is not entered) and $2\mathrm{n} + 2$ steps when $\mathrm{n} > 0$ (and thus the loop is entered). The analysis then proceeds to $\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$, i.e., $\mathrm{x} \neq 1 \vee \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$. The inferred function for the sub-formula $\mathrm{x} \neq 1$ is only defined over the complement of the domain of the one obtained for $\mathrm{x} = 1$. The disjunction combines this function with the one obtained for $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$ by taking the union over the function domains and the maximum over the function values. The result at program point 4 is the same function obtained for $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$. Finally, the analysis can proceed to the initial formula $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$. The function at program point 4 remains the same but its value now indicates the maximum number of steps needed until the *next state* that satisfies $\mathrm{x} = 0$. The function inferred at the beginning of the program proves that the program satisfies the CTL formula $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$ unless x has initial value one. Indeed, in such a case, the program does not satisfy the formula since the loop at program point 1 might never execute. Thus, the inferred precondition is the weakest precondition for the CTL property $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$.

We implemented our approach in the prototype static analyzer FuncTion [13]. Our experimental evaluation demonstrates that the analysis is effective, even for CTL formulas with non-trivial nesting of universal and existential path quantifiers, and performs well on a wide variety of benchmarks.

2 Trace Semantics

We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states, and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states. The set of *final states* of the program is $\Omega \overset{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s' \rangle \notin \tau\}$.

Given a transition system $\langle \Sigma, \tau \rangle$, the function $\mathrm{pre} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states X to the set of their predecessors with respect to τ:$\mathrm{pre}(X) \overset{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau\}$, and the function $\widetilde{\mathrm{pre}} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states X to the set of states whose successors with respect to τ are all in X: $\widetilde{\mathrm{pre}}(X) \overset{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s' \in X\}$.

In the following, given a set S, let $S^n \overset{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in S\}$ be the set of all sequences of exactly n elements from S. We write ε to denote the empty sequence, i.e., $S^0 \overset{\text{def}}{=} \{\varepsilon\}$. Let $S^* \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} S^n$ be the set of all finite sequences, $S^+ \overset{\text{def}}{=} S^* \setminus S^0$ be the set of all non-empty finite sequences, S^ω be the set of all infinite sequences, $S^{+\infty} \overset{\text{def}}{=} S^+ \cup S^\omega$ be the set of all non-empty finite or infinite sequences and $S^{*\infty} \overset{\text{def}}{=} S^* \cup S^\omega$ be the set of all finite or infinite sequences of elements from S. We write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in S^{*\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ if $\sigma \in S^\omega$), $T^+ \overset{\text{def}}{=} T \cap S^+$ for the selection of the non-empty finite sequences of $T \subseteq S^{*\infty}$, $T^\omega \overset{\text{def}}{=} T \cap S^\omega$ for the selection of the infinite sequences of $T \subseteq S^{*\infty}$, and $T \,;\, T' \overset{\text{def}}{=} \{\sigma s \sigma' \mid s \in S, \sigma s \in T, s\sigma' \in T'\}$ for the merging of sets of sequences $T \subseteq S^+$ and $T' \subseteq S^{+\infty}$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states described by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. The set of final states Ω and the transition relation τ can be understood as sets of traces of length one and two, respectively. The *maximal trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system is the union of all non-empty finite traces that are terminating with a final state in Ω, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ [14]:

$$\Lambda = \text{lfp}^{\sqsubseteq}\, \lambda T.\Omega \cup (\tau \,;\, T) \tag{2.1}$$

where the computational order is $T_1 \sqsubseteq T_2 \overset{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$.

The maximal trace semantics carries all information about a program and fully describes its behavior. However, reasoning about a particular property of a program is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the paper, we use *abstract interpretation* [16] to systematically derive, by abstraction of the maximal trace semantics, a sound and complete semantics that precisely captures exactly and only the needed information to reason about CTL properties.

3 Computation Tree Logic

CTL is also known as *branching* temporal logic; its semantics is based on a branching notion of time: at each moment there may be several possible successor program states and thus each moment of time might have several different possible futures. Accordingly, the interpretation of CTL formulas is defined in terms of program states, as opposed to the interpretation of LTL formulas in terms of traces. This section gives a brief introduction into the syntax and semantics of CTL. We refer to [1] for further details.

We assume a set of atomic propositions describing properties of program states. Formulas in CTL are formed according to the following grammar:

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{AX}\phi \mid \mathsf{AG}\phi \mid \mathsf{A}(\phi \;\mathsf{U}\; \phi) \mid \mathsf{EX}\phi \mid \mathsf{EG}\phi \mid \mathsf{E}(\phi \;\mathsf{U}\; \phi)$$

where a is an atomic proposition. The universal quantifier (denoted A) and the existential quantifier (denoted E) allow expressing properties of *all* or *some* traces that start in a state. In the following, we often use Q to mean either A or E. The *next* temporal operator (denoted X) allows expressing properties about the next program state in a trace. The *globally* operator (denoted G) allow expressing properties that should hold always (i.e., for all states) on a trace. The *until* temporal operator (denoted U) allows expressing properties that should hold eventually on a trace, and always until then. We omit the *finally* temporal operator (denoted F) since a formula of the form $\mathsf{QF}\phi$ can be equivalently expressed as $\mathsf{Q}(\mathit{true}\ \mathsf{U}\ \phi)$.

The semantics of formulas in CTL is formally given by a satisfaction relation $\models$ between program states and CTL formulas. In the following, we write $s \models \phi$ if and only if the formula ϕ holds in the program state $s \in \Sigma$. We assume that the satisfaction relation for atomic propositions is given. The satisfaction relation for other CTL formulas is formally defined as follows:

$$\begin{array}{lcl} s \models \neg\phi & \Leftrightarrow & \neg(s \models \phi) \\ s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \phi_1 \vee \phi_2 & \Leftrightarrow & s \models \phi_1 \vee s \models \phi_2 \\ s \models \mathsf{A}\varphi & \Leftrightarrow & \forall \sigma \in T(s) \colon \sigma \models \varphi \\ s \models \mathsf{E}\varphi & \Leftrightarrow & \exists \sigma \in T(s) \colon \sigma \models \varphi \end{array} \tag{3.1}$$

where $T(s) \in \mathcal{P}(\Sigma^{+\infty})$ denotes the set of all program traces starting in the state $s \in \Sigma$. The semantics of trace formulas φ is defined below:

$$\begin{array}{lcl} \sigma \models \mathsf{X}\phi & \Leftrightarrow & \sigma[1] \models \phi \\ \sigma \models \mathsf{G}\phi & \Leftrightarrow & \forall 0 \leq i \colon \sigma[i] \models \phi \\ \sigma \models \phi_1\ \mathsf{U}\ \phi_2 & \Leftrightarrow & \exists 0 \leq i \colon \sigma[i] \models \phi_2 \wedge \forall 0 \leq j < i \colon \sigma[j] \models \phi_1 \end{array} \tag{3.2}$$

where $\sigma[i]$ denotes the program state at position i on the trace $\sigma \in \Sigma^{+\infty}$. We refer to [1] for further details.

4 Program Semantics for CTL Properties

In the following, we derive a program semantics that is *sound and complete* for proving a CTL property. We define the semantics inductively on the structure of a CTL formula. More specifically, for each formula ϕ, we define the *CTL abstraction* $\alpha_\phi \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ which extracts a partial function $f \colon \Sigma \rightharpoonup \mathbb{O}$ from program states to ordinals from a given set of sequences $T \in \mathcal{P}(\Sigma^{+\infty})$ by building upon the CTL abstractions of the sub-formulas of ϕ. The domain of f coincides with the set of program states that satisfy ϕ. Ordinal values are needed to support programs with possibly unbounded non-determinism [18]. The definition of α_ϕ for each CTL formula is summarized in Fig. 2 and explained in more detail below. We use the CTL abstraction to define the program semantics $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ for a formula ϕ by abstraction of the maximal trace semantics Λ.

Definition 1. *Given a CTL formula ϕ and the corresponding CTL abstraction $\alpha_\phi \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$, the program semantics $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ for ϕ is defined as $\Lambda_\phi \overset{def}{=} \alpha_\phi(\Lambda)$, where Λ is the maximal trace semantics (cf. Eq. 2.1).*

ϕ	$\alpha_\phi \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to (\Sigma \rightharpoonup \mathbb{O})$
a	$\alpha_a(T) \overset{\text{def}}{=} \lambda s \in \text{st}(T). \begin{cases} 0 & s \models a \\ \text{undefined} & \text{otherwise} \end{cases}$
$\mathsf{QX}\phi$	$\alpha_{\mathsf{QX}\phi}(T) \overset{\text{def}}{=} \lambda s \in \text{st}(T). \begin{cases} 0 & s \in \text{trans}_{\mathsf{Q}}(\text{dom}(\alpha_\phi(T))) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)$	$\alpha_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}(T) \overset{\text{def}}{=} \alpha_{\mathsf{Q}}^{\text{rk}}(\alpha^{\text{sq}}_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}(T))$
$\mathsf{QG}\phi$	$\alpha_{\mathsf{QG}\phi}(T) \overset{\text{def}}{=} \text{gfp}^{\sqsubseteq}_{\alpha_\phi(T)}\ \Theta_{\mathsf{QG}\phi}$ $\Theta_{\mathsf{QG}\phi}(f) \overset{\text{def}}{=} \lambda s. \begin{cases} f(s) & s \in \text{dom}(f) \cap \text{trans}_{\mathsf{Q}}(\text{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\neg\phi$	$\alpha_{\neg\phi}(T) \overset{\text{def}}{=} \lambda s \in \text{st}(T). \begin{cases} 0 & s \notin \text{dom}(\alpha_\phi(T)) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\phi_1 \wedge \phi_2$	$\alpha_{\phi_1 \wedge \phi_2}(T) \overset{\text{def}}{=} \lambda s \in \text{st}(T). \begin{cases} \sup\{f_1(s), f_2(s)\} & s \in \text{dom}(f_1) \cap \text{dom}(f_2) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\phi_1 \vee \phi_2$	$\alpha_{\phi_1 \vee \phi_2}(T) \overset{\text{def}}{=} \lambda s \in \text{st}(T). \begin{cases} \sup\{f_1(s), f_2(s)\} & s \in \text{dom}(f_1) \cap \text{dom}(f_2) \\ f_1(s) & s \in \text{dom}(f_1) \setminus \text{dom}(f_2) \\ f_2(s) & s \in \text{dom}(f_2) \setminus \text{dom}(f_1) \\ \text{undefined} & \text{otherwise} \end{cases}$

Fig. 2. CTL abstraction $\alpha_\phi \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to (\Sigma \rightharpoonup \mathbb{O})$ for each CTL formula ϕ. The function $\text{trans}_{\mathsf{Q}}$ stands for pre, if Q is E, or $\widetilde{\text{pre}}$, if Q is A (cf. Sect. 2). The *state function* $\text{st} \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to \mathcal{P}(\Sigma)$ collects all states of a given set of sequences T: $\text{st}(T) \overset{\text{def}}{=} \{s \in \Sigma \mid \exists \sigma' \in \Sigma^*, \sigma'' \in \Sigma^{*\infty} : \sigma' s \sigma'' \in T\}$. The *ranking abstraction* $\alpha_{\mathsf{Q}}^{\text{rk}} \colon \mathcal{P}\left(\Sigma^{+}\right) \to (\Sigma \rightharpoonup \mathbb{O})$ is defined in Eq. 4.1, while the *subsequence abstraction* $\alpha^{\text{sq}}_{\mathsf{QF}\phi} \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to \mathcal{P}\left(\Sigma^{+}\right)$ is defined in Eqs. 4.2 and 4.3. In the last two rows, $f_1 \overset{\text{def}}{=} \alpha_{\phi_1}(T)$ and $f_2 \overset{\text{def}}{=} \alpha_{\phi_2}(T)$.

Remarks. It may seem unintuitive to define Λ_ϕ starting from program traces rather than program states (as in Sect. 3). The reason behind this deliberate choice is that it allows placing Λ_ϕ in the hierarchy of semantics defined by Cousot [14], which is a uniform framework that makes program semantics easily comparable and facilitates explaining the similarities and correspondences between semantic models. Specifically, this enables the comparison with existing semantics for termination [18] and other liveness properties [41] (cf. Sect. 7).

It may also seem unnecessary to define Λ_ϕ to be a function. However, this choice yields a uniform treatment of CTL formulas independently of whether they express safety or liveness properties (or a combination of these). Additionally, it allows leveraging existing abstract domains [38,39] (cf. Sect. 5) to obtain a sound static analysis for CTL properties. In particular, the proof of the soundness of the static analysis (cf. Theorem 2 and [38] for more details) requires reasoning both about the domain of Λ_ϕ as well as its value.

Atomic Propositions. For an atomic proposition a, the CTL abstraction $\alpha_a \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ simply extracts from a given set T of sequences a partial function that maps the states of the sequences in T (i.e., $s \in \mathrm{st}(T)$) that satisfy a (i.e., $s \models a$) to the constant value zero, meaning that no program execution steps are needed until a is satisfied for all executions starting in those states. Thus, the domain of the corresponding program semantics $\Lambda_a \colon \Sigma \rightharpoonup \mathbb{O}$ is (cf. Definition 1) is the set of program states that satisfy a (since $\mathrm{st}(\Lambda) = \Sigma$).

Next-Formulas. Next-formulas $\mathsf{QX}\phi$ express that the next state of all traces (if Q is A) or at least one trace (if Q is E) satisfies ϕ.

The CTL abstraction $\alpha_{\mathsf{QX}\phi} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ for a next-formula $\mathsf{QX}\phi$ (cf. Fig. 2) maps a set T of sequences to a partial function defined over the states of the sequences in T (i.e., $s \in \mathrm{st}(T)$) that are the *predecessors* of the states that satisfy ϕ, that is, the predecessors of the states in the domain of the CTL abstraction for ϕ (i.e., $s \in \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(\alpha_\phi(T)))$). The function has constant value zero over its domain, again meaning that no program execution steps are needed until $\mathsf{QX}\phi$ is satisfied for all executions starting in those states.

Thus, the domain of the program semantics $\Lambda_{\mathsf{QX}\phi} \colon \Sigma \rightharpoonup \mathbb{O}$ is the set of states inevitably (for $\Lambda_{\mathsf{AX}\phi}$) or possibly (for $\Lambda_{\mathsf{EX}\phi}$) leading to a state in the domain $\mathrm{dom}(\Lambda_\phi)$ of the program semantics of the sub-formula ϕ (cf. Definition 1).

Until-Formulas. Until-formulas $\mathsf{Q}(\phi_1 \mathbin{\mathsf{U}} \phi_2)$ express that some desired state (i.e., a state satisfying the sub-formula ϕ_2) is eventually reached during program execution, either in all traces (if Q is A) or in at least one trace (if Q is E), and the sub-formula ϕ_1 is satisfied in all program states encountered until then. Thus, we can observe that an until-formula is satisfied by *finite* subsequences of possibly *infinite* program traces. To reason about subsequences, we define the *subsequence function* $\mathrm{sq} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ which extracts all finite subsequences of a given set of sequences T: $\mathrm{sq}(T) \overset{\mathrm{def}}{=} \{\sigma \in \Sigma^{+} \mid \exists \sigma' \in \Sigma^{*}, \sigma'' \in \Sigma^{*\infty} : \sigma'\sigma\sigma'' \in T\}$. In the following, given a formula $\mathsf{Q}(\phi_1 \mathbin{\mathsf{U}} \phi_2)$, we define the corresponding *subsequence abstraction* $\alpha^{\mathrm{sq}}_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ which extracts the finite subsequences that satisfy $\mathsf{Q}(\phi_1 \mathbin{\mathsf{U}} \phi_2)$ from of a set of sequences T. We can then use $\alpha^{\mathrm{sq}}_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$ to define the CTL abstraction $\alpha_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ as shown in Fig. 2. The *ranking abstraction* $\alpha^{\mathrm{rk}}_{\mathsf{Q}} \colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ is:

$$\alpha^{\mathrm{rk}}_{\mathsf{Q}}(T) \overset{\mathrm{def}}{=} \alpha^{\mathrm{v}}_{\mathsf{Q}}(\vec{\alpha}\,(T)) \tag{4.1}$$

where $\vec{\alpha} \colon \mathcal{P}(\Sigma^{+}) \to \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma \times \Sigma)$ extracts from a given set of non-empty finite sequences T the smallest transition system $\langle S, r \rangle$ that generates T :$\vec{\alpha}\,(T) \overset{\mathrm{def}}{=} \langle \mathrm{st}(T), \{\langle s, s' \rangle \in \Sigma \times \Sigma \mid \exists \sigma \in \Sigma^{*}, \sigma' \in \Sigma^{*\infty} \colon \sigma s s' \sigma' \in T\}\rangle$ and the function $\alpha^{\mathrm{v}}_{\mathsf{Q}} \colon \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ provides the rank of the elements in the domain of the transition relation of the transition system:

$$\alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle s \overset{\text{def}}{=} \begin{cases} 0 & \forall s' \in S : \langle s, s'\rangle \notin r \\ \mathrm{bnd}_{\mathsf{Q}} \left\{ \alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle s' + 1 \;\middle|\; \begin{array}{c} s \neq s', \langle s, s'\rangle \in r, \\ s' \in \mathrm{dom}(\alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle) \end{array} \right\} & \text{otherwise} \end{cases}$$

where $\mathrm{bnd}_{\mathsf{Q}}$ stands for sup, if Q is A, or inf, if Q is E. The CTL abstraction $\alpha_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$ (resp. $\alpha_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$) maps the states $\mathrm{st}(T)$ of a given set of sequences T that satisfy $\mathsf{Q}(\phi_1 \;\mathsf{U}\; \phi_2)$ to an upper bound (resp. lower bound) on the number of program execution steps until the sub-formula ϕ_2 is satisfied, for all (resp. at least one of the) executions starting in those states.

Existential Until-Formulas. The subsequence abstraction $\alpha^{\mathrm{sq}}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$ for a formula $\mathsf{E}(\phi_1 \;\mathsf{U}\; \phi_2)$ extracts from a given a set of sequences T the finite subsequence of states that terminate in a state satisfying ϕ_2 and all predecessor states satisfy ϕ_1 (and not ϕ_2). It is defined as follows:

$$\begin{aligned} \alpha^{\mathrm{sq}}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}(T) &\overset{\text{def}}{=} \overline{\alpha}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\alpha_{\phi_1}(T))][\mathrm{dom}(\alpha_{\phi_2}(T))]T \\ \overline{\alpha}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]T &\overset{\text{def}}{=} \{\sigma s \in \mathrm{sq}(T) \mid \sigma \in (S_1 \setminus S_2)^*, s \in S_2\} \end{aligned} \tag{4.2}$$

where S_1 is the set of states that satisfy the sub-formula ϕ_1 (i.e., $\mathrm{dom}(\alpha_{\phi_1}(T))$), and S_2 is the set of desired states (i.e., $\mathrm{dom}(\alpha_{\phi_2}(T))$).

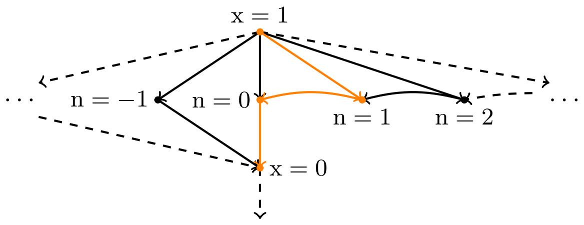
Universal Until-Formulas. A finite subsequence of states satisfies a universal until-formula $\mathsf{A}(\phi_1 \;\mathsf{U}\; \phi_2)$ if and only if it terminates in a state satisfying ϕ_2, all predecessor states satisfy ϕ_1, *and all other sequences with a common prefix also terminate in a state satisfying* ϕ_2 *(and all its predecessors satisfy* ϕ_1), i.e., the program reaches a desired state (via states that satisfy ϕ_1) independently of the non-deterministic choices made during execution. We define the *neighborhood* of a sequence of states σ in a given set T as the set of sequences $\sigma' \in T$ with a common prefix with σ: $\mathrm{nbhd}(\sigma, T) \overset{\text{def}}{=} \{\sigma' \in T \mid \mathrm{pf}(\sigma) \cap \mathrm{pf}(\sigma') \neq \emptyset\}$, where the *prefixes function* $\mathrm{pf}\colon \Sigma^{+\infty} \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of non-empty prefixes of a sequence $\sigma \in \Sigma^{+\infty}$: $\mathrm{pf}(\sigma) \overset{\text{def}}{=} \{\sigma' \in \Sigma^{+\infty} \mid \exists \sigma'' \in \Sigma^{*\infty} : \sigma = \sigma'\sigma''\}$.

We can now defined the subsequence abstraction $\alpha^{\mathrm{sq}}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$:

$$\begin{aligned} \alpha^{\mathrm{sq}}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}(T) &\overset{\text{def}}{=} \overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\alpha_{\phi_1}(T))][\mathrm{dom}(\alpha_{\phi_2}(T))]T \\ \overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]T &\overset{\text{def}}{=} \left\{ \sigma s \in \mathrm{sq}(T) \;\middle|\; \begin{array}{c} \sigma \in (S_1 \setminus S_2)^*, s \in S_2, \\ \mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset, \\ \mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap Z) = \emptyset \end{array} \right\} \end{aligned} \tag{4.3}$$

where the *suffixes function* $\mathrm{sf}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of non-empty suffixes of a set of sequences T: $\mathrm{sf}(T) \overset{\text{def}}{=} \bigcup\{\sigma \in \Sigma^{+\infty} \mid \exists \sigma' \in \Sigma^* : \sigma'\sigma \in T\}$, and $Z \overset{\text{def}}{=} \left\{\sigma s \sigma' \in \Sigma^{+\infty} \mid \sigma \in \Sigma^* \wedge s \in \overline{S_1 \cup S_2} \wedge \sigma' \in \Sigma^{+\infty}\right\}$ is the set of sequences of states in which at least one state satisfies neither ϕ_1 nor ϕ_2. The last two conjuncts in the definition of the helper function $\overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]$ ensure that a finite subsequence satisfies $\mathsf{A}(\phi_1 \ \mathsf{U} \ \phi_2)$ only if it does not have a common prefix with any subsequence of T that never reaches a desired state in S_2 (i.e., $\text{nbhd}(\sigma, \text{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset$) and with any subsequence that contains a state that does not belong to S_1 and S_2 (i.e., $\text{nbhd}(\sigma, \text{sf}(T) \cap Z) = \emptyset$).

Example 1. Let us consider again the acquire/release program of Fig. 1 and let T be the set of its traces. The suffixes starting at program point **2** of the traces in T can be visualized as follows:



Observe that these sequences form a neighborhood in the set $\text{sf}(T)$ of suffixes of T (i.e., the set of all these sequences is the neighborhood $\text{nbhd}(\sigma, \text{sf}(T))$ of any sequence σ in the set). In the following, we write x_i and n_i for the states denoted above by $\mathrm{x} = i$ and $\mathrm{n} = i$, respectively.

Let us consider the universal until-formula $\mathsf{A}(\mathrm{x} = 1 \ \mathsf{U} \ \mathrm{x} = 0)$. The set of desired states is $S_2 = \{\mathrm{x}_0\}$ and $S_1 = \{\mathrm{x}_1\} \cup \{\mathrm{n}_i \mid i \in \mathbb{Z}\}$ is the set of states that satisfy $\mathrm{x} = 1$. All sequences in the neighborhood have prefixes of the form σs where $\sigma = \mathrm{x}_1 \cdots \in (S_1 \cap \overline{S_2})^*$ and $s = \mathrm{x}_0 \in S_2$. Thus, the neighborhood of any subsequence σs does not contain sequences in $\overline{S_2}^{+\infty}$ that never reach the desired state x_0 (i.e., $\text{nbhd}(\sigma s, \text{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset$). Furthermore, the neighborhood does not contain sequences in Z in which at least one state neither satisfies $\mathrm{x} = 1$ nor $\mathrm{x} = 0$ (i.e., $\text{nbhd}(\sigma, \text{sf}(T) \cap Z) = \emptyset$). Therefore, the until-formula $\mathsf{A}(\mathrm{x} = 1 \ \mathsf{U} \ \mathrm{x} = 0)$ is satisfied at program point **2**.

Let us consider now the formula $\mathsf{A}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n} \ \mathsf{U} \ \mathrm{x} = 0)$. Again, all sequences in the neighborhood eventually reach the desired state x_0. However, in this case, the set S_1 is limited to states with non-negative values for n, i.e., $S_1 = \{\mathrm{x}_1\} \cup \{\mathrm{n}_i \mid 0 \leq i\}$. Thus, the neighborhood also contains sequences in which at least one state satisfies neither $\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}$ nor $\mathrm{x} = 0$ (e.g., the sequence $\mathrm{x}_1\mathrm{n}_{-1}\ldots$). Hence $\mathsf{A}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n} \ \mathsf{U} \ \mathrm{x} = 0)$ is not satisfied at program point **2** since $\text{nbhd}(\sigma, \text{sf}(T) \cap Z) \neq \emptyset$. Instead, the existential until-formula $\mathsf{E}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n} \ \mathsf{U} \ \mathrm{x} = 0)$ is satisfied since, for instance, the subsequence σs where $\sigma = \mathrm{x}_1\mathrm{n}_1$ and $s = \mathrm{x}_0$ satisfies $(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n} \ \mathsf{U} \ \mathrm{x} = 0)$.

Until Program Semantics. We now have all the ingredients that define the program semantics $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \colon \Sigma \rightharpoonup \mathbb{O}$ for an until-formula $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$ (cf. Definition 1). Let $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ be the partially ordered set for the computational order $f_1 \sqsubseteq f_2 \Leftrightarrow \text{dom}(f_1) \subseteq \text{dom}(f_2) \wedge \forall x \in \text{dom}(f_1) : f_1(x) \leq f_2(x)$. The program semantics $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$ can be expressed as a least fixpoint in $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ as:

$$\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} = \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}} \Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\Lambda_{\phi_1})][\mathrm{dom}(\Lambda_{\phi_2})]$$

$$\Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]f \stackrel{\text{def}}{=} \lambda s. \begin{cases} 0 & s \in S_2 \\ \mathrm{bnd}_{\mathsf{Q}} \{ f(s') + 1 \mid \langle s, s' \rangle \in \tau \} & s \in S_1 \wedge s \notin S_2 \wedge \\ & s \in \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{4.4}$$

where $\dot{\emptyset}$ is the totally undefined function. The program semantics $\Lambda_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$ (resp. $\Lambda_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$) is a well-founded function mapping each program state in $\mathrm{dom}(\Lambda_{\phi_1})$ inevitably (resp. possibly) leading to a desirable state in $\mathrm{dom}(\Lambda_{\phi_2})$ to an ordinal, which represents an upper bound (resp. lower bound) on the number of program execution steps needed until a desirable state is reached.

Globally-Formulas. Globally-formulas $\mathsf{QG}\phi$ express that ϕ holds indefinitely in all traces (if Q is A) or at least one trace (if Q is E) starting in a state.

The definition of the CTL abstraction $\alpha_{\mathsf{QG}\phi} \colon \mathcal{P}(\Sigma^{+\infty}) \rightarrow (\Sigma \rightharpoonup \mathbb{O})$ for $\mathsf{QG}\phi$ given in Fig. 2 retains the value of the CTL abstraction corresponding to the sub-formula ϕ. Intuitively, each iteration discards the states that satisfy ϕ (i.e., the states in $\mathrm{dom}(\alpha_\phi(T))$) but branch to (sub)sequences of T that do not satisfy $\mathsf{QG}\phi$. Preserving the value of α_ϕ provides more information than just mapping each state to the constant value zero. For instance, the CTL abstraction $\alpha_{\mathsf{AGAF}\phi}$ for a globally-formula $\mathsf{AGAF}\phi$ provides an upper bound on the number of program execution steps needed until the *next occurrence* of ϕ is satisfied, for all executions starting in the corresponding state.

The corresponding program semantics $\Lambda_{\mathsf{QG}\phi} \colon \Sigma \rightharpoonup \mathbb{O}$ (cf. Definition 1) preserves the value of Λ_ϕ for each state satisfying the sub-formula ϕ and inevitably (if Q is A) or possibly (if Q is E) leading only to other states that also satisfy ϕ.

Other Formulas. We are left with describing the CTL abstraction of $\neg\phi$, $\phi \wedge \phi$, and $\phi \vee \phi$ defined in Fig. 2. For a negation $\neg\phi$, the CTL abstraction $\alpha_{\neg\phi}$ maps each program state that does not satisfy ϕ to the value zero. The CTL abstraction for a binary formula $\phi_1 \wedge \phi_2$ or $\phi_1 \vee \phi_2$ retains the largest value of the functions Λ_{ϕ_1} and Λ_{ϕ_2} for each program state satisfying both ϕ_1 and ϕ_2; for a disjunction $\phi_1 \vee \phi_2$, it also retains the value of the function for each program state satisfying either sub-formula.

Theorem 1. *A program satisfies a CTL formula ϕ for all traces starting from a given set of states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\Lambda_\phi)$.*

Proof. The proof proceeds by induction over the structure of the CTL formula ϕ. The base case are atomic propositions a for which the proof is immediate.

For a next-formulas $\mathsf{QX}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda_\phi)$ coincides with the set of states that satisfy ϕ. By Definition 1 and the definition of $\alpha_{\mathsf{QX}\phi}$ in Fig. 2, the domain of $\Lambda_{\mathsf{QX}\phi}$ coincides with $\mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(\alpha_\phi(T)))$. Thus, by definition of $\mathrm{trans}_{\mathsf{Q}}$, we have that $\mathrm{dom}(\Lambda_{\mathsf{QX}\phi})$ coincides with the set of states that satisfy $\mathsf{QX}\phi$ (cf. Eqs. 3.1 and 3.2).

For an until-formula $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$, by induction hypothesis, $\mathrm{dom}(\Lambda_{\phi_1})$ and $\mathrm{dom}(\Lambda_{\phi_2})$ coincide with the set of states that satisfy ϕ_1 and ϕ_2, respectively. We have $\Lambda_{\mathsf{Q}}(\phi_1 \ \mathsf{U} \ \phi_2) = \Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\Lambda_{\phi_1})][\mathrm{dom}(\Lambda_{\phi_2})](\Lambda_{\mathsf{Q}}(\phi_1 \ \mathsf{U} \ \phi_2))$ from Eq. 4.4. Therefore, by definition of $\Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$, $\mathrm{dom}(\Lambda_{\mathsf{Q}}(\phi_1 \ \mathsf{U} \ \phi_2))$ coincides with the states that satisfy ϕ_2 and all states that satisfy ϕ_1 and inevitably (if Q is A) or possibly (if Q is E) lead to states that satisfy ϕ_2. So $\mathrm{dom}(\Lambda_{\mathsf{Q}}(\phi_1 \ \mathsf{U} \ \phi_2))$ coincides with the states that satisfy $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$ (cf. Eqs. 3.1 and 3.2).

For a globally-formula $\mathsf{QG}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda_\phi)$ coincides with the set of states that satisfy ϕ. By Definition 1 and the definition of $\alpha_{\mathsf{QG}\phi}$ in Fig. 2, we have that $\Lambda_{\mathsf{QG}\phi} = \Theta_{\mathsf{QG}\phi}(\Lambda_{\mathsf{QG}\phi})$. Therefore, by definition of $\Theta_{\mathsf{QG}\phi}$, we have that $\mathrm{dom}(\Lambda_{\mathsf{QG}\phi})$ coincides with the states that satisfy ϕ inevitably (if Q is A) or possibly (if Q is E) lead to other states that satisfy ϕ. So $\mathrm{dom}(\Lambda_{\mathsf{QG}\phi})$ coincides with the states that satisfy $\mathsf{QG}\phi$ (cf. Eqs. 3.1 and 3.2).

Finally, all other cases ($\neg\phi$, $\phi_1 \wedge \phi_2$, and $\phi_1 \vee \phi_2$) follow immediately from the induction hypothesis, the semantics of the CTL formulas (cf. Eq. 3.1) and the definition of the corresponding program semantics (cf. Definition 1 and the CTL abstractions in Fig. 2). $\square$

The program semantics for a CTL formula is not computable when the program state space is infinite. In the next section, we present decidable abstractions by means of piecewise-defined functions [38,39].

5 Static Analysis for CTL Properties

We recall here the features of the abstract domain of piecewise-defined functions [39] that are relevant for our purposes, and describe the new elements that we need to introduce to obtain a static analysis for proving CTL properties. We refer to [38] for an exhaustive presentation of the original abstract domain.

For illustration, we model a program using a control flow graph $\langle \mathcal{L}, E \rangle$, where $\mathcal{L}$ is the set of program points and $E \subseteq \mathcal{L} \times A \times \mathcal{L}$ is the set of edges in the control flow graph. Each edge is labeled by an action $s \in A$; possible actions are skip, a boolean condition b, or an assignment $x := e$. In the following, we write $out(l)$ to denote the set of outgoing edges from a program point l.

Piecewise-Defined Functions Abstract Domain. An element $t \in \mathcal{T}$ of the abstract domain is a piecewise-defined partial function represented by a *decision tree*, where the decision nodes are labeled by linear constraints over the program variables, and the leaf nodes are labeled by functions of the program variables. The decision nodes recursively partition the space of possible values of the program variables, and the leaf nodes represent the value of the function corresponding to each partition. An example of (simplified) decision tree representation of a piecewise-defined function is shown in Fig. 3.

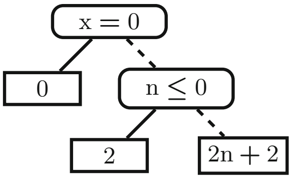


Fig. 3. Simplified decision tree representation of the piecewise-defined function inferred at program point 4 of the program of Fig. 1 (cf. Eq. 1.1). Each constraint is satisfied by the left subtree of the decision node, while the right subtree satisfies its negation. The leaves represent partial functions whose domain is determined by the constraints satisfied along the path to the leaf.

Specifically, the decision nodes are labeled by linear constraints supported by an existing underlying numerical domain, i.e., interval [15] constraints (of the form $\pm\mathrm{x} \leq \mathrm{c}$), octagonal [30] constraints (of the form $\pm\mathrm{x}_i \pm \mathrm{x}_j \leq c$), or polyhedral [19] constraints (of the form $\mathrm{c}_1 \cdot \mathrm{x}_1 + \cdots + \mathrm{c}_k \cdot \mathrm{x}_k \leq \mathrm{c}_{k+1}$). The leaf nodes are labeled by *affine functions* of the program variables (of the form $\mathrm{m}_1 \cdot \mathrm{x}_1 + \cdots + \mathrm{m}_k \cdot \mathrm{x}_k + \mathrm{q}$), or the special elements $\perp$ and $\top$, which explicitly represent undefined functions. The element $\top$ is introduced to manifest an irrecoverable precision loss of the analysis. We also support *lexicographic* affine *functions* $(\mathrm{f}_k, \ldots, \mathrm{f}_1, \mathrm{f}_0)$ in the isomorphic form of ordinals $\omega^k \cdot \mathrm{f}_k + \cdots + \omega \cdot \mathrm{f}_1 + \mathrm{f}_0$ [29,40].

The partitioning is dynamic: during the analysis of a control flow graph, partitions (i.e. decision nodes and constraints) are modified by assignments and split (i.e., added) by boolean conditions and when merging control flows. More specifically, for each action $s \in A$, we define sound over-approximating abstract transformers $[\![s]\!]_o : \mathcal{T} \to \mathcal{T}$ as well as *new under-approximating abstract transformers* $[\![s]\!]_u : \mathcal{T} \to \mathcal{T}$. These transformers always increase by one the value of the functions labeling the leaves of a given decision tree to count the number of executed program steps (i.e., actions in the control flow graph). The transformers for boolean conditions and assignments additionally modify the decision nodes by building upon the underlying numerical abstract domain. For instance, the abstract transformer $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) for a boolean condition b uses the underlying numerical domain to obtain an over-approximation (resp. an under-approximation) of b as a set of linear constraints; then it adds these constraints to the given decision tree and discards the paths that become unfeasible (because they do not satisfy the added constraints). Let $\{n \leq 0\}$ (resp. $\{n = 0\}$) be the set of constraints obtained by $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) for the boolean condition $b \equiv n \leq 0 \wedge n\%2 = 0$; then, given the right subtree in Fig. 3, $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) would discard the path leading to the leaf with value $2n+2$ by replacing it with a leaf with undefined value $\perp$ (resp. replace $n \leq 0$ with $n = 0$ and replace $2n+2$ with $\perp$). Decision trees are merged using either the approximation join $\curlyvee$ or the computational join $\sqcup$. Both join operators add missing decision nodes from either of the given trees; $\curlyvee$ retains the leaves that are labeled with an undefined function in at least one of the given trees, while $\sqcup$ preserves the leaves that are labeled with a defined function over the leaves labeled with $\perp$ (but preserves the leaves labeled with $\top$ over all other leaves). To minimize the cost of the analysis and to enforce termination, a (dual) widening operator limits the height of the decision trees and the number of maintained partitions.

$$\Lambda^\natural_a \overset{\text{def}}{=} \lambda l.\ \textsc{reset}\, [\![a]\!]\, (\bot) \tag{5.1}$$

$$\Lambda^\natural_{\mathsf{QX}\phi} \overset{\text{def}}{=} \lambda l.\ \textsc{zero} \left(\bigsqcup^{\mathsf{Q}}_{(l,s,l') \in out(l)} [\![s]\!]_{\mathsf{Q}}\, (\Lambda^\natural_\phi(l')) \right) \tag{5.2}$$

$$\Lambda^\natural_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \overset{\text{def}}{=} \text{lfp}^\natural_{\lambda l.\bot}\ \lambda m.\ \lambda l.\ \textsc{until} \left[\!\!\left[\Lambda^\natural_{\phi_1}(l), \Lambda^\natural_{\phi_2}(l) \right]\!\!\right] \left(\bigsqcup^{\mathsf{Q}}_{(l,s,l') \in out(l)} [\![s]\!]_{\mathsf{Q}}\, (m(l')) \right) \tag{5.3}$$

$$\Lambda^\natural_{\mathsf{QG}\phi} \overset{\text{def}}{=} \text{gfp}^\natural_{\Lambda^\natural_\phi}\ \lambda m.\ \lambda l.\ \textsc{mask} \left[\!\!\left[\bigsqcup^{\mathsf{Q}}_{(l,s,l') \in out(l)} [\![s]\!]_{\mathsf{Q}}\, (m(l')) \right]\!\!\right] (m(l)) \tag{5.4}$$

$$\Lambda^\natural_{\neg\phi} \overset{\text{def}}{=} \lambda l.\ \textsc{complement}(\Lambda^\natural_\phi(l)) \tag{5.5}$$

$$\Lambda^\natural_{\phi_1 \wedge \phi_2} \overset{\text{def}}{=} \lambda l.\ \Lambda^\natural_{\phi_1}(l) \curlyvee \Lambda^\natural_{\phi_2}(l) \tag{5.6}$$

$$\Lambda^\natural_{\phi_1 \vee \phi_2} \overset{\text{def}}{=} \lambda l.\ \Lambda^\natural_{\phi_1}(l) \sqcup \Lambda^\natural_{\phi_2}(l) \tag{5.7}$$

Fig. 4. Abstract program semantics $\Lambda^\natural_\phi$ for each CTL formula ϕ. The join operator $\sqcup^{\mathsf{Q}}$ and the abstract transformer $[\![s]\!]_{\mathsf{Q}}$ respectively stand for $\sqcup$ and $[\![s]\!]_u$, if Q is E, or $\curlyvee$ and $[\![s]\!]_o$, if Q is A. With abuse of notation, we use $\bot$ to denote a decision tree with a single undefined leaf node.

Abstract Program Semantics for CTL Properties. The abstract program semantics $\Lambda^\natural_\phi : \mathcal{L} \to \mathcal{T}$ for a CTL formula ϕ maps each program point $l \in \mathcal{L}$ to an element $t \in \mathcal{T}$ of the piecewise-defined functions abstract domain. The definition of $\Lambda^\natural_\phi$ for each CTL formula ϕ is summarized in Fig. 4 and explained in some detail below. More details and formal definitions can be found in [37,38].

The analysis starts with the totally undefined function (i.e., a decision tree that consists of a single leaf with undefined value $\bot$) at the final program points (i.e., nodes without outgoing edges in the control flow graph). Then it proceeds backwards through the control flow graph, taking the encountered actions into account, and joining decision trees when merging control flows. For existential CTL properties, the analysis uses the under-approximating abstract transformers $[\![s]\!]_u$ for each action s, to ensure that program states that do not satisfy the CTL property are discarded (i.e., removed from the domain of the current piecewise-defined function), and joins decision trees using the computational join $\sqcup$, to ensure that the current piecewise-defined function remains defined over states that satisfy the CTL property in at least one of the merged control flows. Dually, for universal CTL properties, the analysis uses the over-approximating abstract transformers $[\![s]\!]_o$ and joins decision trees using the approximation join $\curlyvee$, to ensure that the current piecewise-defined function remains defined only over states that satisfy the CTL property in all of the merged control flows.

At each program point, the analysis additionally performs operations that are specific to the considered CTL formula ϕ. For an atomic proposition a (cf. Eq. 5.1), the analysis performs a RESET $[\![a]\!]$ operation, which is analogous to the under-approximating transformer for boolean conditions but additionally replaces all the leaves that satisfy a with leaves labeled with the function with value zero. For example, given the atomic proposition $n = 0$ and the right subtree in Fig. 3, RESET $[\![n = 0]\!]$ would replace the constraint $n \leq 0$ with $n = 0$, the leaf $2n + 2$ with $\bot$ and the leaf 2 with 0. For a next-formula $\mathsf{QX}\phi$ (cf. Eq. 5.2), the analysis approximates the effect of the transition from each program point l to each successor program point l' and performs a ZERO operation to replace all defined functions labeling the leaves of the so obtained decision tree with the function with value zero. For an until-formula $\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)$ (cf. Eq. 5.4), the analysis performs an ascending iteration with widening [13]. At each iteration, the analysis approximates the effect of the transition from each program point l to each successor program point l' and performs an UNTIL operation to model the until temporal operator: UNTIL replaces with the function with value zero all leaves that correspond to defined leaves in the decision tree $\Lambda^\natural_{\phi_2}(l)$ obtained for ϕ_2, and retains all leaves that are labeled with an undefined function in both $\Lambda^\natural_{\phi_1}(l)$ and $\Lambda^\natural_{\phi_1}(l)$. For a globally-formula $\mathsf{QG}\phi$ (cf. Eq. 5.5), the analysis performs a descending iteration with dual widening [41], starting from the abstract semantics $\Lambda^\natural_\phi$ obtained for ϕ. At each iteration, the MASK operation models the globally temporal operator: it discards all defined partitions in $\Lambda^\natural_\phi(l)$ that become undefined as a result of the transition from each program point l to each successor program point l'; at the limit, the only defined partitions are those that remain defined across transitions and thus satisfy the globally-formula. For a negation formula $\neg\phi$ (cf. Eq. 5.5), the analysis performs a COMPLEMENT operation on the decision tree $\Lambda^\natural_\phi(l)$ obtained for ϕ at each program point l; COMPLEMENT replaces all defined functions labeling the leaves of a decision tree with $\bot$, and all $\bot$ with the function with value zero. Note that $\Lambda^\natural_\phi$ is an abstraction of Λ_ϕ and thus not all undefined partitions in $\Lambda^\natural_\phi$ necessarily correspond to undefined partitions in Λ_ϕ. Leaves that are undefined in $\Lambda^\natural_\phi$ due to this uncertainty are labeled with $\top$, and are left unchanged by COMPLEMENT to guarantee the soundness of the analysis. Finally, for binary formulas $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$, the abstract semantics $\Lambda^\natural_{\phi_1 \wedge \phi_2}$ and $\Lambda^\natural_{\phi_1 \vee \phi_2}$ (cf. Eqs. 5.6 and 5.7) merge the decision trees obtained for ϕ_1 and ϕ_2 using the approximation join $\curlyvee$ and the computational join $\sqcup$, respectively.

The abstract program semantics $\Lambda^\natural_\phi$ for each CTL formula ϕ is *sound* with respect to the approximation order $f_1 \preccurlyeq f_2 \Leftrightarrow \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x)$, which means that the abstract semantics $\Lambda^\natural_\phi$ *over-approximates* the value of the concrete semantics Λ_ϕ and *under-approximates* its domain of definition $\mathrm{dom}(\Lambda_\phi)$. In this way, the abstraction provides sufficient preconditions for the CTL property ϕ: if the abstraction is defined for a state then that state satisfies ϕ.

Theorem 2. *A program satisfies a CTL formula ϕ for all traces starting from a given set of states $\mathcal{I}$ if $\mathcal{I} \subseteq \mathrm{dom}(\gamma(\Lambda^{\natural}_{\phi}))$.*

Proof. (Sketch). The proof proceeds by induction over the structure of the formula ϕ. The base case are atomic propositions for which the proof is immediate.

For a next-formula $\mathsf{QX}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda^{\natural}_{\phi})$ is a subset of the set of states that satisfy ϕ. Using the over-approximating transformers $[\![s]\!]_o$ together with the approximation join $\curlyvee$ (resp. the under-approximating transformers $[\![s]\!]_u$ together with the computational join $\sqcup$) ensures that $\Lambda^{\natural}_{\mathsf{QX}\phi}$ soundly under-approximates the set of states that satisfy $\mathsf{QX}\phi$.

For an until-formula $\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)$, by induction hypothesis, $\mathrm{dom}(\Lambda^{\natural}_{\phi_1})$ and $\mathrm{dom}(\Lambda^{\natural}_{\phi_2})$ are a subset of the set of states that satisfy ϕ_1 and ϕ_2, respectively. By definition, $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$ is the limit of an ascending iteration sequence using widening. Again, using the over-approximating transformers $[\![s]\!]_o$ together with the approximation join $\curlyvee$ (resp. the under-approximating transformers $[\![s]\!]_u$ together with the computational join $\sqcup$) guarantees the soundness of the analysis with respect to each transition. The soundness of each iteration without widening is then guaranteed by the definition of the UNTIL operation. The iterations with widening are allowed to be unsound but the limit of the iterations (i.e., $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$) is guaranteed to soundly under-approximate the set of states that satisfy $(\phi_1 \mathsf{U} \phi_2)$. We refer to [38] for a detailed proof for formulas of the form $(\mathit{true} \mathsf{U} \phi_2)$. The generalization to $(\phi_1 \mathsf{U} \phi_2)$ is trivial.

For a globally-formula $\mathsf{QG}\phi$, $\Lambda_{\mathsf{QG}\phi}$ is the limit of a descending iteration sequence with dual widening, starting from $\Lambda^{\natural}_{\phi}$, which soundly under-approximates the set of states that satisfy ϕ. The soundness of each iteration is guaranteed by the definition of the MASK operation and the dual widening operator (see [38]).

The case of a negation $\neg\phi$ is non-trivial since, by induction hypothesis, $\mathrm{dom}(\Lambda^{\natural}_{\phi})$ is a subset of the set of states that satisfy ϕ. Specifically, $\Lambda^{\natural}_{\phi}$ maps each program point $l \in \mathcal{L}$ to a decision tree whose leaves determine this under-approximation: leaves labeled with $\bot$ represent states that do not satisfy ϕ while leaves labeled with $\top$ represent states that may or may not satisfy ϕ. The COMPLEMENT operation performed by $\Lambda^{\natural}_{\neg\phi}$ only considers leaves labeled by $\bot$ and ignores (i.e., leaves unchanged) leaves labeled by $\top$. Thus, $\Lambda^{\natural}_{\neg\phi}$ soundly under-approximates the set of states that satisfy $\neg\phi$.

Finally, for binary formulas $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$, the proof follows immediately from the definition of the approximation join $\curlyvee$ and the computational join $\sqcup$ used in the definition of $\Lambda^{\natural}_{\phi_1 \wedge \phi_2}$ and $\Lambda^{\natural}_{\phi_1 \vee \phi_2}$, respectively. □

6 Implementation

The proposed static analysis method for proving CTL properties is implemented in the prototype static analyzer FuncTion [13].

The implementation is in OCaml and consists of around 9K lines of code. The current front-end of FuncTion accepts non-deterministic programs written in a C-like syntax (without support for pointers, `struct` and `union` types). The only basic data type is mathematical integers. FuncTion accepts CTL properties written using a syntax similar to the one used in the rest of this paper, with atomic propositions written as C-like pure expressions. The abstract domain of piecewise-defined functions builds on the numerical abstract domains provided by the Apron library [24], and the under-approximating numerical operators provided by the Banal static analyzer [31].

The analysis is performed backwards on the control flow graph of a program with a standard worklist algorithm [32], using widening and dual widening at loop heads. Non-recursive function calls are inlined, while recursion is supported by augmenting the control flow graphs with call and return edges. The precision of the analysis can be tuned by choosing the underlying numerical abstract domain, by activating the extension to ordinal-value functions [40], and by adjusting the precision of the widening [13] and the widening delay. It is also possible to refine the analysis by considering only reachable states.

Experimental Evaluation. We evaluated our technique on a number of test cases obtained from various sources, and compared FuncTion against T2 [8] and Ultimate LTL Automizer [20] as well as E-HSF [4], and the prototype implementation from [10]. Figs. 5 and 6 show an excerpt of the results, which demonstrates the differences between FuncTion, T2 [8] and Ultimate LTL Automizer. The first set of test cases are programs that we used to test our implementation. The second and third set were collected from [25] and the web

No	Program	CTL Property	Result	Time
1.1	and_test.c	$\mathsf{AGAF}(n = 1) \wedge \mathsf{AF}(n = 0)$	✓	0.05s
1.2	and_test.c	$\mathsf{EGAF}(n = 1)$	✓	0.05s
1.4	global_test.c	$\mathsf{AGEF}(x \leq -10)$	✓	0.15s
1.7	or_test.c	$\mathsf{AFEG}(x < -100) \vee \mathsf{AF}(x = 20)$	✓	0.05s
1.8	may_term...	$\mathsf{EF}(\text{exit} : \mathit{true})$	✗	-
1.9	until_test.c	$\mathsf{A}(x \geq y \;\mathsf{U}\; x = y)$	✓	0.03s
1.11	fin_ex.c	$\mathsf{EGEF}(n = 1)$	✓	0.04s
1.12	until_ex.c	$\mathsf{E}(x \geq y \;\mathsf{U}\; x = y)$	✓	0.03s
2.3	win4.c	$\mathsf{AFAG}(\text{WItemsNum} \geq 1)$	✓	0.15s
2.4	toylin.c	$(c \leq 5 \wedge c > 0) \vee \mathsf{AF}(\text{resp} > 5)$	✗	-
3.9	cb5_safe.c	$\mathsf{A}(i = 0 \;\mathsf{U}\; (\mathsf{A}(i = 1 \;\mathsf{U}\; \mathsf{AG}(i = 3)) \vee \mathsf{AG}(i = 1)))$	✗	-
3.14	timer...	$\neg\mathsf{AG}(\text{timer} = 0 \Rightarrow \mathsf{AF}(\text{output} = 1))$	✗	-
3.15	togglec...	$\mathsf{AG}(\mathsf{AF}(t = 1) \wedge \mathsf{AF}(t = 0))$	✗	-
4.1	Bangalore...	$\mathsf{EF}(x < 0)$	✗	-
4.2	Ex02...	$i < 5 \Rightarrow \mathsf{AF}(\text{exit} : \mathit{true})$	✓	0.04s
4.3	Ex07...	$\mathsf{AFEG}(i = 0)$	✓	0.1s
4.4	java_Seq...	$\mathsf{EF}(\mathsf{AF}(j \geq 21) \wedge i = 100)$	✓	0.3s
4.5	Madrid...	$\mathsf{AF}(x = 7 \wedge \mathsf{EFAG}(x = 2))$	✓	0.02s

Fig. 5. Evaluation of FuncTion on selected test cases collected from various sources. All test cases were analyzed using polyhedral constraints for the decision nodes, and affine functions for the leaf nodes of the decision tree.

No	FuncTion	T2 [8]	Ultimate LTL Automizer [20]
1.1	✓	✗	✓
1.2	✓	✗	-
1.4	✓	✗	-
1.7	✓	✗ (error)	-
1.8	✗	✓	-
1.9	✓	✗	✓
1.11	✓	✗	-
1.12	✓	✗ (no implementation)	-
2.3	✓	✗	✓
2.4	✗	✗	✓
3.9	✗	-	✓
3.14	✗	-	✓
3.15	✗	-	✓
4.1	✗	✓	-
4.2	✓	✗ (out of memory)	✓
4.3	✓	✗	-
4.4	✓	✗ (error)	-
4.5	✓	✗	-

Fig. 6. Differences between FuncTion, T2, and Ultimate LTL Automizer.

interface of Ultimate LTL Automizer [20]. The fourth set are examples from the termination category of the 6th International Competition on Software Verification (SV-COMP 2017). The experiments were conducted on an Intel i7-6600U processor with 20 GB of RAM on Arch Linux with Linux 4.11 and OCaml 4.04.1.

FuncTion passes all test cases with the exception of 2.4, 3.9, 3.14, and 3.15, which fail due to imprecisions introduced by the widening, and 1.8 and 4.1, which fail due to an unfortunate interaction of the under-approximations needed for existential properties and non-deterministic assignments in the programs. However, note that for these test cases we still get some useful information. For instance, for 3.15, FuncTion infers that the CTL property is satisfied if $x < 0$.

In Fig. 6, the missing results for T2 are due to a missing conversion of the test cases to the T2 input format. The comparison with Ultimate LTL Automizer is limited to the test cases where the CTL property can be equivalently expressed in LTL (i.e., universal CTL properties). The results show that only FuncTion succeeds on numerous test cases (1.2, 1.4, 1.7, 1.11, 1.12, 4.3, 4.4, and 4.5). Ultimate LTL Automizer performs well on the supported test cases, but FuncTion still succeeds on most of the test cases provided by Ultimate LTL Automizer (not shown in Fig. 6, since there are no differences between the results of FuncTion and Ultimate LTL Automizer). Overall, none of the tools subsumes the others. In fact, we observe that their combination is more powerful than any of the tools alone, as it would succeed on all test cases.

Finally, FuncTion only succeeds on two of the industrial benchmarks from [10], while T2, E-HSF and [10] fare much better (see [8, Fig. 11]). The reason for the poor performance is that in these benchmarks the effect of function calls is modeled as a non-deterministic assignment and this heavily impacts the precision of FuncTion. We are confident that we would obtain better results on the original benchmarks, where function calls are not abstracted away.

7 Related Work

In the recent past, a large body of work has been devoted to proving CTL properties of programs. The problem has been extensively studied for finite-state programs [7,26, etc.], while most of the existing approaches for infinite-state systems have limitations that restrict their applicability. For instance, they only support certain classes of programs [36], or they limit their scope to a subset of CTL [11], or to a single CTL property such as termination [27,34, etc.] or non-termination [2,5, etc.]. Our approach does not suffer from these limitations.

Some other approaches for proving CTL properties do not reliably support CTL formulas with arbitrary nesting of universal and existential path quantifiers [23], or support existential path quantifiers only indirectly by building upon recent work for proving non-termination [22], or by considering their universal dual [8]. In particular, the latter approach is problematic: since the universal dual of an existential until formula is non-trivial to define, the current implementation of T2 does not support such formulas (see Fig. 6). Other indirect approaches [4,10] perform unnecessary computations that result in slower runtimes (see [8, Fig. 12]). In comparison to all these approaches, our approach provides strictly more information in the form of a ranking function whose domain gives a precondition for a given CTL property and whose value estimates the number of program execution steps until the property is satisfied.

In [17], Cousot and Cousot define a trace-based semantics for a very general temporal language which subsumes LTL and CTL; this is subsequently abstracted to a state-based semantics. The abstraction has been later shown to be incomplete by Giacobazzi and Ranzato [21]. In contrast to the work of Cousot and Cousot, we do not define a trace-based semantics for CTL. The semantics that we propose is close to their state-based semantics in that their state-based semantics coincides with the domain of the functions that we define. Note that Theorem 1 is not in contrast with the result of Giacobazzi and Ranzato because completeness is proven with respect to the state-based semantics of CTL.

Finally, our abstract interpretation framework generalizes an existing framework [41] for proving guarantee and recurrence properties of programs [28]. Guarantee and recurrence properties are equivalently expressed in CTL as $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \phi)$ and $\mathsf{AGA}(\mathit{true}\ \mathsf{U}\ \phi)$, respectively. In fact, we rediscover the guarantee and recurrence program semantics defined in [41] as instances of our framework: the guarantee semantics coincides with $\Lambda_{\mathsf{A}(\mathit{true}\mathsf{U}\phi)}$ (cf. Sect. 4) and the recurrence semantics coincides with $\Lambda_{\mathsf{AGA}(\mathit{true}\mathsf{U}\phi)}$ (cf. Sect. 4). The common insight with our work is the observation that CTL (sub)formulas are satisfied by finite subsequences (which can also be single states) of possibly infinite sequences. The program semantics for these (sub)formulas then counts the number of steps in these subsequences. Our work generalizes this idea to all CTL formulas and integrates the corresponding semantics in a uniform framework.

8 Conclusion and Future Work

In this paper, we have presented a new static analysis method for inferring preconditions for CTL properties of programs that overcomes the limitations of existing approaches. We have derived our static analysis within the framework of abstract interpretation by abstraction of the operational trace semantics of a program. Using experimental evidence, we have shown that our analysis is effective and performs well on a wide variety of benchmarks, and is able to prove CTL properties that are out of reach for state-of-the-art tools.

It remains for future work to investigate and improve the precision of the analysis in the presence of non-deterministic program assignments. We also plan to support LTL properties [20] or, more generally, CTL* properties [9]. This requires some form of trace partitioning [35] as the interpretation of LTL formulas is defined in terms of program executions instead of program states as CTL.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Bakhirkin, A., Piterman, N.: Finding recurrent sets with backward analysis and trace partitioning. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 17–35. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_2
3. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA, pp. 1–38 (2010)
4. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_61
5. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_11
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
8. Cook, B., Khlaaf, H., Piterman, N.: Faster temporal reasoning for infinite-state programs. In: FMCAD, pp. 75–82 (2014)
9. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
10. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: PLDI, pp. 219–230 (2013)

11. Cook, B., Koskinen, E., Vardi, M.: Temporal property verification as a program analysis task. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 333–348. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_26
12. Cook, B., Koskinen, E., Vardi, M.Y.: Temporal property verification as a program analysis task - extended version. Formal Methods Syst. Des. **41**(1), 66–82 (2012)
13. Courant, N., Urban, C.: Precise widening operators for proving termination by abstract interpretation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 136–152. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_8
14. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoret. Comput. Sci. **277**(1–2), 47–103 (2002)
15. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Symposium on Programming, pp. 106–130 (1976)
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
17. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: POPL, pp. 12–25 (2000)
18. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL, pp. 245–258(2012)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
20. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: a new approach to LTL software model checking. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 49–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_4
21. Giacobazzi, R., Ranzato, F.: Incompleteness of states w.r.t. traces in model checking. Inf. Comput. **204**(3), 376–407 (2006)
22. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.: Proving non-termination. In: POPL, pp. 147–158 (2008)
23. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: a software model-checker for verification and refutation. In: CAV, pp. 170–174 (2006)
24. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
25. Koskinen, E.: Temporal verification of programs. Ph.D. thesis, University of Cambridge, November 2012
26. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
27. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92 (2001)
28. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC, pp. 377–410 (1990)
29. Manna, Z., Pnueli, A.: The Temporal Verification of Reactive Systems: Progress (1996)
30. Miné, A.: The octagon abstract domain. High. Order Symbolic Comput. **19**(1), 31–100 (2006)
31. Miné, A.: Inferring sufficient conditions with backward polyhedral under-approximations. Electron. Notes Theor. Comput. Sci. **287**, 89–100 (2012)

32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, (1999)
33. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
34. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
35. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM TOPLAS **29**(5), 26 (2007)
36. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. Theoret. Comput. Sci. **549**, 127–145 (2014)
37. Ueltschi, S.: Proving temporal properties by abstract interpretation. Master's thesis, ETH Zurich, Zurich, Switzerland (2017)
38. Urban, C.: Static Analysis by abstract interpretation of functional temporal properties of programs. Ph.D. thesis, École Normale Supérieure, Paris, France, July 2015
39. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: SAS, pp. 302–318 (2014)
40. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: ESOP, pp. 412–431 (2014)
41. Urban, C., Miné, A.: Inference of ranking functions for proving temporal properties by abstract interpretation. Comput. Lang. Syst. Struct. **47**, 77–103 (2017)