Proving Temporal Properties by Abstract Interpretation

Samuel Marco Ueltschi

September 6, 2017

Contents

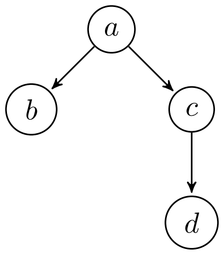


Figure 1: A basic state transition system

1 Introduction

Motivation etc.

2 State Transition Systems

To be able to analyze the behavior of a program, it is necessary to express said behavior through a mathematical model. We model the operational semantics of programs using transition systems. This is based on the definitions presented in [1].

Definition 2.1. A transition system is a tuple $\langle \Sigma, \tau \rangle$ where Σ is the set of all states in the system and $\tau \in \Sigma \times \Sigma$ is the so called transition relations that defines how one can transition from one state to the other.

Transition systems allow us to model the semantics of a program independently of the programming language in which it was written. By expressing the possible transition between states in terms of a relation, it is also possible to capture nondeterminism. Figure 1 shows a simple transition system represented as directed graphs. States are represented as nodes and state transitions as directed edges.

We introduce the following auxiliary functions over states of a transition systems which will become useful in section 5 where we defined the semantics of CTL operators in terms of transition systems.

Definition 2.2. Given a transition system $\langle \Sigma, \tau \rangle$. $\text{pre}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \in \mathcal{P}(\Sigma)$ to the set of their predecessors with respect to the program transition relation τ:

$$\text{pre}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X \colon \langle s, s' \rangle \in \tau\} \tag{1}$$

Definition 2.3. Given a transition system $\langle \Sigma, \tau \rangle$. $\widetilde{\text{pre}}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \in \mathcal{P}(\Sigma)$ to the set of their predecessors with respect to the program transition relation τ with the limitation that only those predecessor states are selected which exclusively transition to states in X:

$$\widetilde{\mathrm{pre}}(X) \stackrel{\mathsf{def}}{=} \{s \in \Sigma \mid \forall s' \in X \colon \langle s, s' \rangle \in \tau \Rightarrow s' \in X\} \tag{2}$$

To get an intuition for the difference between $\widetilde{\mathrm{pre}}$ and pre, consider the state transition system depicted in figure 1. There it holds that $\mathrm{pre}(\{b, d\}) = \{a, c\}$ because a is the predecessor of b and c the predecessor of d. However note that $\widetilde{\mathrm{pre}}(\{b, d\}) = \{c\}$ since only c has transitions that exclusively end up in either b or d. Consequently it holds that $\widetilde{\mathrm{pre}}(\{b, c\}) = \{a\}$ because a transitions exclusively to either b or c.

The next section introduces Computation Tree Logic, a logic for stating properties about transition systems.

3 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) is a logic which allows stating properties about execution traces of state transition systems. In the context of this thesis, CTL is used to express temporal properties about the runtime behavior of programs. This section gives a brief introduction into the syntax and semantic of CTL. Further information about CTL can be found in [2].

We assume the existence of some atomic proposition logic over the set of states. Atomic propositions $a \in \mathtt{AP}$ state properties of states $\sigma \in \Sigma$. The satisfaction relation $\models \subseteq \Sigma \times \mathtt{AP}$ determines if a state satisfies an atomic proposition.

3.1 Syntax

The syntax of a CTL formula is given by the following grammar.

$$\begin{aligned}
\Phi ::= & \\
& a \mid \\
& \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \\
& \forall \bigcirc \Phi \mid \exists \bigcirc \Phi \mid \\
& \forall\Diamond\Phi \mid \exists\Diamond\Phi \mid \\
& \forall\Box\Phi \mid \exists\Box\Phi \mid \\
& \forall(\Phi\ U\ \Phi) \mid \exists(\Phi\ U\ \Phi)
\end{aligned}$$

The term a is an atomic propositions. Formulae with quantifiers $\exists$ or $\forall$ are called path-dependent, formulae without are called path-independent.

3.2 Semantic

We now define the satisfaction relation $\models$ between states $\sigma \in \Sigma$ and CTL formulae. The satisfaction relation for atomic propositions $a \in \mathtt{AP}$ is assumed to be given by the underlying logic for atomic propositions.

$$
\begin{aligned}
\sigma \models a &\iff \sigma \models a \\
\sigma \models \neg\Phi &\iff \text{not } \sigma \models \Phi \\
\sigma \models \Phi_1 \wedge \Phi_2 &\iff (\sigma \models \Phi_1) \text{ and } (\sigma \models \Phi_1) \\
\sigma \models \Phi_1 \vee \Phi_2 &\iff (\sigma \models \Phi_1) \text{ or } (\sigma \models \Phi_1) \\
\sigma \models \forall \bigcirc \Phi &\iff \forall \pi \in \mathit{Paths}(\sigma)\colon (\pi[1] \models \Phi) \\
\sigma \models \exists \bigcirc \Phi &\iff \exists \pi \in \mathit{Paths}(\sigma)\colon (\pi[1] \models \Phi) \\
\sigma \models \forall(\Phi_1 \ U \ \Phi_2) &\iff \forall \pi \in \mathit{Paths}(\sigma)\colon \\
&\qquad (\exists j \geq 0\colon \pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j\colon \pi[k] \models \Phi_1)) \\
\sigma \models \exists(\Phi_1 \ U \ \Phi_2) &\iff \exists \pi \in \mathit{Paths}(\sigma)\colon \\
&\qquad (\exists j \geq 0\colon \pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j\colon \pi[k] \models \Phi_1)) \\
\sigma \models \forall\Box\Phi &\iff \forall \pi \in \mathit{Paths}(\sigma)\colon (\forall j \geq 0\colon \pi[j] \models \Phi) \\
\sigma \models \exists\Box\Phi &\iff \exists \pi \in \mathit{Paths}(\sigma)\colon (\forall j \geq 0\colon \pi[j] \models \Phi)
\end{aligned}
$$

The states $\sigma \in \Sigma$ are part of a state transition system $\langle \Sigma, \tau \rangle$ and $\mathit{Paths}(\sigma_0)$ is the set of all paths $\pi = \sigma_0\sigma_1\sigma_2\ldots$ starting from σ_0 with $\pi[j] = \sigma_j$. The CTL formulae $\forall\Diamond\Phi$ and $\exists\Diamond\Phi$ are not explicitly defined. They are equivalent to $\forall(\mathit{true} \ U \ \Phi)$ and $\exists(\mathit{true} \ U \ \Phi)$.

Properties $\forall(\Phi_1 \ U \ \Phi_2)$ and $\exists(\Phi_1 \ U \ \Phi_2)$ are called until properties. These properties require that a state satisfying Φ_2 is reached eventually among all (some, respectively) paths, and until then Φ_1 holds.

Properties $\forall\Box\Phi$ and $\exists\Box\Phi$ are called global properties. They state that all states among all (some, respectively) reachable paths satisfy the property Φ.

Properties $\forall \bigcirc \Phi$ and $\exists \bigcirc \Phi$ are called next properties. They state that all (some, respectively) immediate next states satisfy the property Φ.

The following useful equivalence relations exists which can be used to relate existential to universal CTL formulae.

$$\exists \bigcirc \Phi \equiv \neg\forall \bigcirc (\neg\Phi)$$
$$\exists\Diamond\Phi \equiv \neg\forall\Box(\neg\Phi)$$
$$\exists\Box\Phi \equiv \neg\forall\Diamond(\neg\Phi)$$

The next section introduces the concept of ranking functions, a proof method for liveness properties. This proof method will then be extended to CTL in Section 5.

4 Ranking Functions

The traditional method for proving termination are *ranking functions* [3] [4]. A ranking function is a partial function from program states to a well-ordered set like natural numbers or ordinals. To prove termination, the value of the ranking function must decrease during program execution. Cousot and Cousot prove the existence of a *most precise ranking function* and that it can be derived by abstract interpretation [5]. We will call this *most precise ranking function*, as defined by Cousot and Cousot, the *termination semantics* from now on.

Definition 4.1. The *termination semantics* is a ranking function $\tau^t \in \Sigma \rightarrow \mathbb{O}$. A program starting from some initial state $\sigma \in \Sigma$ terminates if and only if $\sigma \in dom(\tau^t)$.

The *termination semantics* assigns an upper bound on the number of steps until termination to each state. Therefore, a program terminates if its initial state is in the domain of the *termination semantics*.

Based on the work of Cousot and Cousot [5], Urban and Miné [1] extended the *termination semantics* to other liveness properties. More precisely to guarantee and recurrence properties. A guarantee property states that some state satisfying a given property is guaranteed to be reached eventually. Termination is therefore just a guarantee property stating that some final state will be eventually reached. As with termination, the *guarantee semantics* is a ranking function that assigns each state an upper bound on the number of steps until a state satisfying said property is reached. Guarantee properties can be expressed using the CTL formula $\forall\Diamond(a)$

Definition 4.2. The *guarantee semantics* is a ranking function $\tau^g_{[S]} \in \Sigma \rightarrow \mathbb{O}$ where $S \subseteq \Sigma$ is a set of states satisfying a desired property. A program starting from some state $\sigma \in \Sigma$ will reach a state $s \in S$ if and only if $\sigma \in dom(\tau^g_{[S]})$.

In addition to guarantee properties, Urban and Miné [1] also introduced the *recurrence semantics*. A recurrence property guarantees that a program starting from some state $\sigma \in \Sigma$ will reach some state satisfying a given property infinitely often. The value assigned to a state by the *recurrence semantics* is an upper bound on the number of executions steps until the next state satisfying the property is reached. Recurrence properties can be expressed using the CTL formula $\forall\Box\forall\Diamond(a)$.

Definition 4.3. The *recurrence semantics* is a ranking function $\tau^r_{[S]} \in \Sigma \rightarrow \mathbb{N}$ where $S \subseteq \Sigma$ is a set of states satisfying a desired property. A program starting from some state $\sigma \in \Sigma$ will reach a state $s \in S$ infinitely often if and only if $\sigma \in dom(\tau^r_{[S]})$.

Figure 2 shows an example for the semantics discussed in this section. We illustrate ranking functions by labeling the states with the corresponding value assigned to them by the function. The first example (a) shows the *termination semantics* for a state transition system that always terminates. Therefore the initial state has the value 2 assigned to it stating that this program terminates in at most two steps.

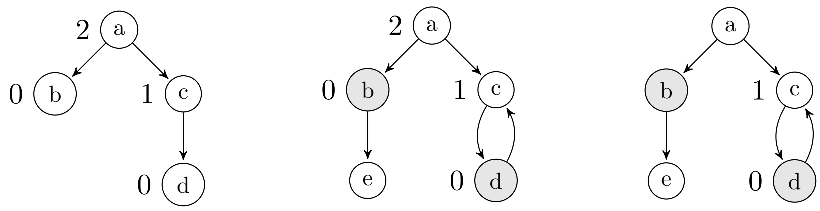
The second example (b) shows the *guarantee semantics* for the guarantee property that states that a gray state will be reached eventually ($\forall\Diamond$(gray)). This holds for example (b), therefore the initial state has the value 2 assigned to it. The program reaches a gray state in at most two steps.

The last example (c) shows the *recurrence semantics* for the recurrence property that states that a gray state will be reached infinitely often ($\forall\Box\forall\Diamond$(gray)). As one can see in the transition system, this is not true when starting from the initial state. Therefore the *recurrence semantics* is undefined for the initial state. However the property holds when starting from state c or d. Accordingly these two states have the values 1 and 0 assigned to them.

We refer to [6] for a detailed discussion of the various semantics presented in this section. The next section extends on the concepts of this section and presents a proof method for CTL properties.

5 Concrete Semantics for CTL

In Section 4 we introduced the concept of ranking functions and explained how they express the semantics of termination and also liveness properties in general. We recall that ranking functions are a proof method for liveness properties. The ranking function assigns well-ordered values to program states. Liveness properties state that some goal state will be reached eventually. The ranking function assigns 0 to those goal states and increased the



(a) τ^t (b) $\tau^g_{[\{b,d\}]}$ (c) $\tau^r_{[\{b,d\}]}$

Figure 2: Example *termination semantics* (a), *guarantee semantics* (b) and *recurrence semantics* (c)

values for states leading up to said goal state through backtracking. One can determine if a state satisfies a liveness property by checking if the corresponding ranking function assigns a value to said state. The assigned value is an upper bound on the number of steps until a goal state is reached. In this section we extend this proof method presented in [6] to CTL.

CTL can define both liveness and safety properties. For liveness properties, ranking functions can be used as proof method. However, ranking functions are not suitable in the case of safety properties since there is no goal state to be reached eventually. Because a CTL formula can arbitrarily combine liveness and safety properties we will use a *counting function* $\tau_\Phi \in \Sigma \to \mathbb{O}$ as proof method. The function τ_Φ is similar to a ranking function, it assigns well-ordered values to states. A state satisfies a CTL property Φ if it is in the domain of τ_Φ. However, as opposed to ranking functions, the value of the *counting function* τ_Φ is not guaranteed to decrease. It will only decrease if Φ is a liveness property. In that case, the value of the function is an upper bound on the number of steps until some goal states is reached. Otherwise, in the case of safety properties, the value may be constant and is irrelevant. We call τ_Φ the *CTL semantics* from now on.

Theorem 5.1. The *CTL semantics* for a given CTL formula Φ is a counting function $\tau_\Phi \in \Sigma \to \mathbb{N}$. It encodes the semantics of Φ for a given state transition system $\langle \Sigma, \tau \rangle$ such that $\sigma \models \Phi \iff \sigma \in dom(\tau_\Phi)$

We will define the *CTL semantics* inductively for each CTL operator such that arbitrary combinations of CTL properties can be expressed. Furthermore we split the definition into path-independent and path-dependent CTL operators.

5.1 Path Independent Operators

We start by defining the *CTL semantics* for atomic propositions and logic operators. These CTL properties are path independent and can be defined individually for each state $\sigma \in \Sigma$. The definitions are formed according to

Theorem 5.1 and the satisfiability relation for CTL properties defined in Section 3.

Definition 5.1. Equations for path independent CTL operators

$$\tau_a \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models a \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3}$$

$$\tau_{\neg\Phi} \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \notin \mathit{dom}(\tau_\Phi) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{4}$$

$$\tau_{\Phi_1 \wedge \Phi_2} \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} \sup\{\tau_{\Phi_1}(s), \tau_{\Phi_2}(\sigma)\} & \text{if } \sigma \in \mathit{dom}(\tau_{\Phi_1}) \cap \mathit{dom}(\tau_{\Phi_2}) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5}$$

$$\tau_{\Phi_1 \vee \Phi_2} \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} \sup\{\tau_{\Phi_1}(\sigma), \tau_{\Phi_2}(\sigma)\} & \text{if } \sigma \in \mathit{dom}(\tau_{\Phi_1}) \cap \mathit{dom}(\tau_{\Phi_2}) \\ \tau_{\Phi_1}(\sigma) & \text{if } \sigma \in \mathit{dom}(\tau_{\Phi_1}) \setminus \mathit{dom}(\tau_{\Phi_2}) \\ \tau_{\Phi_2}(\sigma) & \text{if } \sigma \in \mathit{dom}(\tau_{\Phi_2}) \setminus \mathit{dom}(\tau_{\Phi_1}) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6}$$

Equation 3 assigns 0 to all states that satisfy the atomic proposition a. In case of liveness properties, such as $\forall\Diamond a$, the value 0 marks that the goal of the property has been reached. For safety properties, the value 0 simply states that the states satisfies the property.

The logic $\neg$ operator in equation 4 follows the same approach as for atomic propositions. It interprets $\neg\Phi$ as an atomic proposition and uses the fact that $\sigma \notin \mathit{dom}(\tau_\Phi) \implies \sigma \models \neg\Phi$ which follows from Theorem 5.1.

The logical $\wedge$ and $\vee$ connectives in equations 5 and 6 reuse the values of the underlying functions τ_{Φ_1} and τ_{Φ_2} according to the semantics of these operators. If a state is assigned a value by both functions, then the supremum of the two values is used. That way, if the two underlying properties Φ_1 and Φ_2 are liveness-properties, we preserve the notion of increasing the values of the function when backtracking from the goal states of underlying properties.

Lemma 5.2. The *CTL semantics* for path-independent CTL operators are sound and complete. Let $\sigma \in \Sigma$ and Φ, Φ_1 and Φ_2 be arbitrary CTL properties.

$$\sigma \models a \iff \sigma \in dom(\tau_a) \tag{7}$$
$$\sigma \models \neg\Phi \iff \sigma \in dom(\tau_{\neg\Phi}) \tag{8}$$
$$\sigma \models \Phi_1 \wedge \Phi_2 \iff \sigma \in dom(\tau_{\Phi_1 \wedge \Phi_2}) \tag{9}$$
$$\sigma \models \Phi_1 \vee \Phi_2 \iff \sigma \in dom(\tau_{\Phi_1 \vee \Phi_2}) \tag{10}$$

5.2 Path Dependent Operators

CTL semantics for path-dependent operators until and global are defined in terms of fixed-points. These fixed-points are defined over the partially ordered set of functions $\langle \Sigma \to \mathbb{N}, \sqsubseteq \rangle$. Fixed-point iterates are related to each other using the *computational order* $\sqsubseteq$. This partial order relates functions in terms of expressiveness, i.e., for how many states can a function prove that the CTL property holds.

Definition 5.2. Let $f, g \in \Sigma \to \mathbb{O}$. The *computational order* $\sqsubseteq$ is defined as follows.

$$f \sqsubseteq g \iff \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge \forall x \in \mathrm{dom}(f) : f(x) \leq g(x)$$

The *CTL semantics* are not computable in general. In Section 8 we will present a sound decidable abstraction of the *CTL semantics*. Soundness is related to the *approximation order* $\preceq$.

Definition 5.3. Let $f, g \in \Sigma \to \mathbb{O}$. The *approximation order* $\preceq$ is defined as follows.

$$f \preceq g \iff \mathrm{dom}(f) \supseteq \mathrm{dom}(g) \wedge \forall x \in \mathrm{dom}(g) : f(x) \leq g(x)$$

The *approximation order* ranks counting functions in terms of precision. A function f is more precise than a function g if it is defined over more states than g and if the value is smaller for all states in the domain of g. Intuitively, a more precise counting function is able to prove for more states that a CTL Φ holds or does not hold.

Until

Recall that for the CTL property $\forall(\Phi_1 U \Phi_2)$ to hold for some state $\sigma \in \Sigma$, all paths starting from said state must form a chain of states satisfying Φ_1 ending in a state satisfying Φ_2. In case of $\exists(\Phi_1 U \Phi_2)$ at least on such path must exists. The *CTL semantics* for universal and existential until properties are defined as least fixed-points of the abstract transformers

$$\phi_{\forall(\Phi_1 U \Phi_2)} \in (\Sigma \to \mathbb{N}) \to (\Sigma \to \mathbb{N}))$$
$$\phi_{\exists(\Phi_1 U \Phi_2)} \in (\Sigma \to \mathbb{N}) \to (\Sigma \to \mathbb{N}))$$

starting from the totally undefined counting function $\dot{\emptyset}$.

Definition 5.4. *CTL semantics* for existential and universal until properties

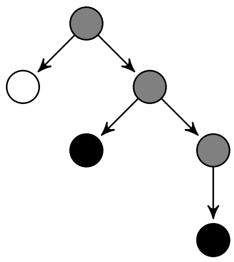
$$\tau_{\forall(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} \text{lfp}_{\dot{\emptyset}}^{\dot{\sqsubseteq}}\ \phi_{\forall(\Phi_1 U \Phi_2)} \tag{11}$$

$$\phi_{\forall(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in dom(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle\sigma, \sigma'\rangle \in \tau\} & \text{if } \sigma \notin dom(\tau_{\Phi_2}) \wedge \\ & \sigma \in dom(\tau_{\Phi_1}) \wedge \\ & \sigma \in \widetilde{pre}(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{12}$$

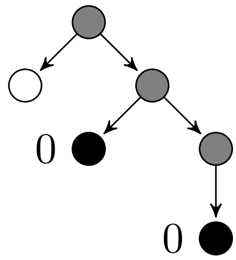
$$\tau_{\exists(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} \text{lfp}_{\dot{\emptyset}}^{\dot{\sqsubseteq}}\ \phi_{\exists(\Phi_1 U \Phi_2)} \tag{13}$$

$$\phi_{\exists(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in dom(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle\sigma, \sigma'\rangle \in \tau\} & \text{if } \sigma \notin dom(\tau_{\Phi_2}) \wedge \\ & \sigma \in dom(\tau_{\Phi_1}) \wedge \\ & \sigma \in pre(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{14}$$

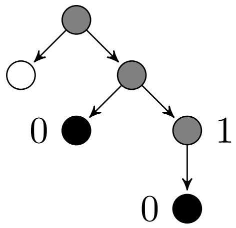
This definition is a generalization of the *guarantee semantics* presented in [1]. The fixed-point iteration starts by assigning the value 0 to all states that satisfy Φ_2. In subsequent iterations we consider all states that satisfy Φ_1 and from which one can only transition to states that already satisfy $\forall(\Phi_1 U \Phi_2)$. These states are then assigned the largest ranking value of all reachable states plus one. By performing iterations this way, we backtrack paths in the state transition systems that end in a state satisfying Φ_2 and which are preceded by an unbroken chain of states satisfying Φ_1. Every state on such a path is guaranteed to satisfy $\forall(\Phi_1 U \Phi_2)$. Furthermore, by starting from 0 at states that satisfy Φ_2 and incrementing the value of the function while backtracking, we construct a counting function such that the value assigned to each state is an upper bound on the number of steps until a state satisfying Φ_2 is reached.



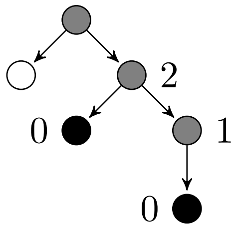
(a) Iteration 0



(b) Iteration 1



(c) Iteration 2



(d) Iteration 3

Figure 3: Iterative computation of $\tau_{\forall(gray\ U\ black)}$.

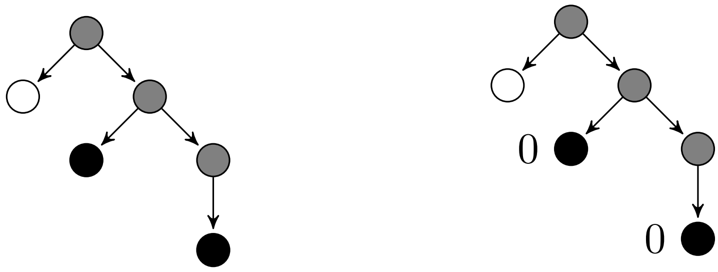
The $\widetilde{pre}$ relation guarantees that during the backtracking, only those states that exclusively transition to states satisfying $\forall(\Phi_1 U \Phi_2)$, are considered. This condition can be relaxed for existential 'until' properties by using the *pre* relation instead. That way, states that have at least one reachable state satisfying $\exists(\Phi_1 U \Phi_2)$ are also considered during the backtracking (see Definitions 2.2 and 2.3).

Lemma 5.3. The *CTL semantics* for until properties are sound and complete. Let $\sigma \in \Sigma$ and Φ_1, Φ_2 be arbitrary CTL properties.

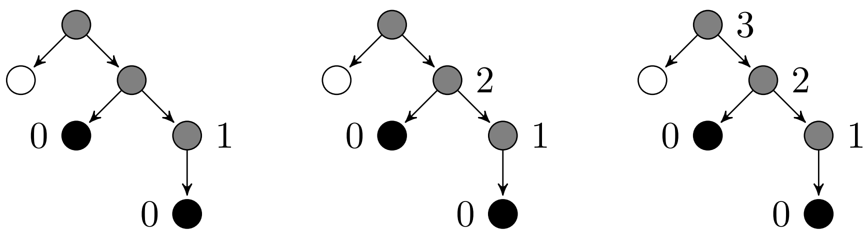
$$\sigma \models \forall(\Phi_1 U \Phi_2) \iff \sigma \in dom(\tau_{\forall(\Phi_1 U \Phi_2)}) \tag{15}$$

$$\sigma \models \exists(\Phi_1 U \Phi_2) \iff \sigma \in dom(\tau_{\exists(\Phi_1 U \Phi_2)}) \tag{16}$$

Figures 3 and 4 give an example on how the iterative computation for until properties works for universal and existential quantifiers. Note how Figure 4 has one additional iteration because of the existential quantifier. The initial state is added to the function in the last iteration because there exists one edge that leads to a state satisfying the property. For the universal property, the iteration stops after three iterations because not all successor states of the initial state satisfy the property.



(a) Iteration 0 (b) Iteration 1



(c) Iteration 2 (d) Iteration 3 (e) Iteration 4

Figure 4: Iterative computation of $\tau_{\exists(gray\ U\ black)}$.

Global

Recall that the CTL global operator states that some property must hold globally, i.e., indefinitely for *all* paths starting from some state in the case of the universal quantifier ($\forall\Box\Phi$) or for *some* paths in case of the existential quantifier ($\exists\Box\Phi$). The CTL semantics for global properties are defined as greatest fixed-point of the abstract transformers

$$\phi_{\forall(\Phi_1 U \Phi_2)} \in (\Sigma \to \mathbb{N}) \to (\Sigma \to \mathbb{N}))$$
$$\phi_{\exists(\Phi_1 U \Phi_2)} \in (\Sigma \to \mathbb{N}) \to (\Sigma \to \mathbb{N}))$$

starting from the CTL semantics τ_Φ of the inner CTL property.

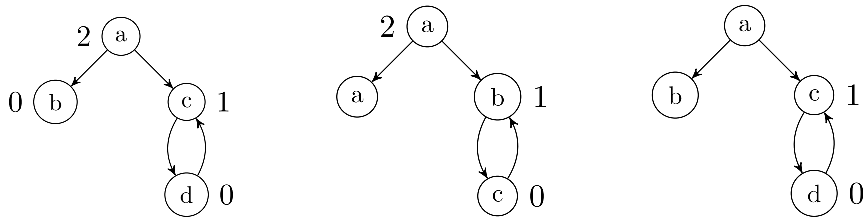
Definition 5.5. Equations for CTL global operator

$$\tau_{\forall\Box\Phi} \stackrel{\text{def}}{=} \text{gfp}^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\forall\Box\Phi} \tag{17}$$

$$\phi_{\forall\Box\Phi} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} f(x) & \text{if } \sigma \in \widetilde{pre}(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{18}$$

$$\tau_{\exists\Box\Phi} \stackrel{\text{def}}{=} \text{gfp}^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\exists\Box\Phi} \tag{19}$$

$$\phi_{\exists\Box\Phi} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} f(x) & \text{if } \sigma \in pre(dom(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{20}$$



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 5: Iterative computation of $\tau_{\forall\Box\Phi}$.

This definition is based on the *recurrence semantics* presented in [1]. As with the until operator, we distinguish between universal and existential properties by using either $\widetilde{pre}$ or pre. The fixed-point iteration starts with the counting function τ_Φ of the inner property Φ. At each iteration, every state that is still part of the domain of the function is inspected. The inspected state is kept in the domain of the function if *all* its successor states (or *some* for the existential case) are also part of the domain of the function, otherwise it is removed. That way, only states which are part of an infinite path consisting exclusively of states satisfying Φ are kept in the domain of the function.

Lemma 5.4. The *CTL semantics* for global properties are sound and complete. Let $\sigma \in \Sigma$ and Φ be an arbitrary CTL property.

$$\sigma \models \forall\Box\Phi \iff \sigma \in dom(\tau_{\forall\Box\Phi}) \tag{21}$$

$$\sigma \models \exists\Box\Phi \iff \sigma \in dom(\tau_{\exists\Box\Phi}) \tag{22}$$

Figures 5 and 6 show this for $\tau_{\exists\Box\Phi}$ and $\tau_{\forall\Box\Phi}$. Both iterations start with some initial counting function τ_Φ. In the first iteration state b is removed because it has no outgoing edges. For the existential case, the iteration stops here because all remaining states a, c and d have at least one edge to a node thats part of the function. In the universal case we get an additional iteration that removes state a because not all of its successor nodes (namely b) are part of the function. Note that only infinite paths are considered to hold globally.

Next

The next operator is path dependent but does not require fixed-point iterations. A state satisfies $\forall \bigcirc \Phi$ if all its immediate successors satisfy the property Φ, correspondingly $\exists \bigcirc \Phi$ is satisfied if at least one immediate successor satisfies the property Φ. This corresponds to the definition of the $\widetilde{pre}$ and pre relations. Zero is assigned to each state that satisfies the property to construct a valid counting function according to Theorem 5.1.

(a) Iteration 0 (b) Iteration 1

Figure 6: Iterative computation of $\tau_{\exists\Box\Phi}$.

Definition 5.6. Equations for CTL next operator

$$\tau_{\forall\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in \widetilde{pre}(dom(\tau_\Phi)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{23}$$

$$\tau_{\exists\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \in pre(dom(\tau_\Phi)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{24}$$

Lemma 5.5. The *CTL semantics* for `next` properties are sound and complete. Let $\sigma \in \Sigma$ and Φ be an arbitrary CTL property.

$$\sigma \models \forall \bigcirc \Phi \iff \sigma \in dom(\tau_{\forall\bigcirc\Phi}) \tag{25}$$
$$\sigma \models \exists \bigcirc \Phi \iff \sigma \in dom(\tau_{\exists\bigcirc\Phi}) \tag{26}$$

6 Imperative Language

In this section we briefly introduce a minimal imperative programming language. It will be used in section 8 to define the abstract CTL semantics. The language has no procedures, pointers or recursion and is non-deterministic. Variables are integer valued ($\mathbb{Z}$) and statically allocated.

First we define the syntax for arithmetic and boolean expressions. The syntax definitions are based on chapter 3 of [6].

Definition 6.1. Syntax for arithmetic and boolean expressions.

Arithmetic expressions are defined over a set of variables $\mathcal{X}$.

$$
\begin{aligned}
aexp \ ::= \ & X && X \in \mathcal{X} \\
\mid \ & [i_1, i_2] && i_1 \in \mathbb{Z} \cup \{-\infty\}, \ i_1 \in \mathbb{Z} \cup \{\infty\}, \ i_1 \leq i_2 \\
\mid \ & -aexp && \\
\mid \ & aexp \diamond aexp && \diamond \in \{+, -, *, /\}
\end{aligned}
$$

$$
\begin{aligned}
bexp \ ::= \ & ? && \text{non-deterministic choice} \\
\mid \ & \neg bexp && \\
\mid \ & bexp \wedge bexp && \\
\mid \ & bexp \vee bexp && \\
\mid \ & aexp \diamond aexp && \diamond \in \{<, \leq, >, \geq\}
\end{aligned}
$$

The semantics for expressions are defined as expected. Please refer to [6] for a formal definition. Note that the symbol ? stands for non-deterministic choice.

Programs are defined in terms as control-flow-graphs. The control-flow-graph models all possible program executions as paths in the graph.

Definition 6.2. Program representation as control-flow-graph. Let $\mathcal{L}$ be the set of program labels.

$$
\begin{aligned}
stmt \ ::= \ & \texttt{skip} && \\
\mid \ & bexp && \\
\mid \ & X := aexp && X \in \mathcal{X}
\end{aligned}
$$

$$
\begin{aligned}
edge &\stackrel{\mathsf{def}}{=} \mathcal{L} \times stmt \times \mathcal{L} \\
cfg &\stackrel{\mathsf{def}}{=} \mathcal{P}(\mathcal{L}) \times \mathcal{P}(edge) \\
prog &\stackrel{\mathsf{def}}{=} cfg \times \mathcal{L} \times \mathcal{L}
\end{aligned}
$$

A control-flow-graph $(V, E) \in cfg$ consists of a set of nodes V and edges E. Every control point of a program is assigned a label $l \in \mathcal{L}$. The nodes in the control-flow-graph correspond to those labels. An edge $(u, s, v) \in E$ states that one can transition from node u to v by executing statement s. The `skip` statement transitions from one node to another without doing anything, the boolean expression *bexp* limits the set of states that are allowed to transition to the next node and the assignment $X := aexp$ assigns the value of the arithmetic expression *aexp* to the variable X. A program $((V, E), l_{entry}, l_{exit}) \in prog$ consists of a control-flow-graph and two special nodes that defined the entry and exit point of the program.

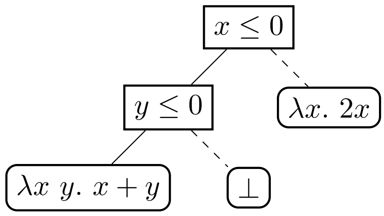


Figure 7: Example for decision tree

We introduce the following auxiliary functions on nodes of a control-flow-graph to refer to the incoming and outgoing edges of a node.

Definition 6.3. Given a control-flow-graph $(V, E) \in cfg$ and some node $l \in V$

$$\begin{aligned} in(l) &\stackrel{\text{def}}{=} \{(u, s, v) \in E \mid v = l\} \\ out(l) &\stackrel{\text{def}}{=} \{(u, s, v) \in E \mid u = l\} \end{aligned}$$

7 Decision Tree Abstract Domain

This section briefly recaps the decision tree abstract domain [6]. Decision trees encode piecewise-defined linear functions which are used as an abstraction of the counting functions introduced in Section 4. First we give a description of the decision tree abstract domain. Then we introduce ordering relations between the elements of the domain and relevant operations on the elements of the domain. An in-depth description of the topics covered in this section can be found in [6].

7.1 Domain

The elements of the abstract domain are binary decision trees. The nodes of the trees are linear constraints and the leafs are linear functions of the program variables. Decision trees partition the state space, given by a set of variables $\mathcal{X}$, into partitions. Each partition is defined through the conjunction of linear constraints on the path from root to leaf in the decision tree. The linear function at the leaf determines the value of the function for the corresponding partition of the state space.

Figure 7 gives an example for such a decision tree. It consists of two nodes with linear constraints $x \leq 0$ and $y \leq 0$. The left most leaf is the function $\lambda x\ y.\ x + y$. It is defined for all states satisfying $x \leq 0 \wedge y \leq 0$ according to the constraints from root to leaf. The right most leaf is the function $\lambda x.\ 2x$, it is defined for all states satisfying $\neg(x \leq 0)$ (following the right child of a node negates the linear constraint). The leaf in the middle is a bottom node, signifying that the function for the corresponding partition is undefined. In summary the decision tree in Figure 7 is equivalent to the following partial function:

$$f(x, y) = \begin{cases} x + y & \text{if } x \leq 0 \land y \leq 0 \\ 2x & \text{if } x > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We will now formalize the decision tree abstract domain.

Constraints

The constrains at the inner nodes of the decision tree are elements of the *linear constraints auxiliary abstract domain* $\mathcal{C}$.

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ c_1 X_1 + \cdots + c_n X_n + c_{n+1} \geq 0 \;\middle|\; \begin{array}{l} \mathcal{X} = \{X_1, \ldots, X_n\} \\ c_1, \ldots, c_n, c_{n+1} \in \mathbb{Z} \\ gcd(|c_1|, \ldots, |c_n|, |c_{n+1}|) = 1 \end{array} \right\}$$

Elements of $\mathcal{C}$ can be instances of the *interval abstract domain* [7], the *octagon abstract domain* [8] or the *polyhedra abstract domain* [9].

Functions

Leafs of the decision trees are elements of the *functions auxiliary abstract domain* $\mathcal{F}$. Elements of $\mathcal{F}$ are either natural valued linear functions or one of the two special elements $\top_{\mathcal{F}}$ or $\bot_{\mathcal{F}}$. The element $\bot_{\mathcal{F}}$ indicates that the function is undefined on the given partition. The element $\top_{\mathcal{F}}$ indicates that the value of the function is unknown for the given partition.

$$\mathcal{F} \stackrel{\text{def}}{=} \{\mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N}\} \cup \{\top_{\mathcal{F}},\ \bot_{\mathcal{F}}\}$$

We will write constant functions that return a value $n \in \mathbb{N}$ by just stating the constant value, e.g., $0 \in \mathcal{F}$ denotes the constant function that returns 0 for every state.

In the following sections we will distinguish between so called *defined* and *undefined* leafs. A leaf $f \in \mathcal{F}$ is called *defined* if f is neither $\top_{\mathcal{F}}$ nor $\bot_{\mathcal{F}}$ and *undefined* otherwise. Defined leafs assign an actual value to its partition, therefore the function that the decision tree represents is defined for that partition. Otherwise the function is undefined.

Decision Trees

We now define the decision tree abstract domain $\mathcal{T}$. An element $t \in \mathcal{T}$ is either a *leaf node* $\texttt{LEAF}\colon f$ consisting of a function $f \in \mathcal{F}$ (denoted $t.f$), or a *decision node* $\texttt{NODE}\{c\} : l; r$ consists of a linear constraint $c \in \mathcal{C}$ (denoted $t.c$) and a left and a right sub tree $l, r \in \mathcal{T}$ (denoted $t.l$ and $t.r$).

$$\mathcal{T} \stackrel{\mathsf{def}}{=} \{\texttt{LEAF}\colon f \mid f \in \mathcal{F}\} \cup \{\texttt{NODE}\{c\} : l; r \mid c \in \mathcal{C}, l, r \in \mathcal{T}\}$$

For algorithmic purposes we also define $\mathcal{T}_{\texttt{NIL}}$. This adds an additional leaf element $\texttt{NIL}$ to $\mathcal{T}$ to represent the absence of information about a partition. $\texttt{NIL}$ leafs usually appear if a partition in a decision tree can be excluded because it is infeasible w.r.t. the program execution.

$$\mathcal{T}_{\texttt{NIL}} \stackrel{\mathsf{def}}{=} \{\texttt{NIL}\} \ \cup \ \{\texttt{LEAF}\colon f \mid f \in \mathcal{F}\} \ \cup \ \{\texttt{NODE}\{c\} : l; r \mid c \in \mathcal{C}, l, r \in \mathcal{T}_{\texttt{NIL}}\}$$

Sound Abstractions

Decision trees are an abstraction of counting functions. A sound abstraction is defined in terms of the *approximation order* $\preceq$ (see Definition 5.3).

Definition 7.1. Let $\gamma \in \mathcal{T} \to (\Sigma \to \mathbb{O})$ be the concretization function from decision trees to counting functions. A decision tree $t \in \mathcal{T}$ is a sound abstraction of a counting function $f \in \Sigma \to \mathbb{N}$ if $f \preceq \gamma(t)$.

This definitions allows us to soundly define computable abstractions of counting functions. Every decision tree computed in such a way captures the behavior of the concrete counting function for all defined partitions of the tree.

Orders

We now define the *computational order* $\sqsubseteq_{\mathcal{T}}$ and *approximation order* $\preceq_{\mathcal{T}}$ over elements of $\mathcal{T}$. These orders are an approximation of the corresponding orders presented in Section 5 (see Definitions 5.2 and 5.3).

Both orders are defined by leaf-wise comparison of two decision trees. To that end we define the *computational order* $\sqsubseteq_{\mathcal{F}}$ and *approximation order* $\preceq_{\mathcal{F}}$ for elements of $\mathcal{F}$. Two trees are related to each other if all leafs are pairwise related w.r.t. $\sqsubseteq_{\mathcal{F}}$ ($\preceq_{\mathcal{F}}$, respectively). Please refer to [6] for a detailed explanation.

The two orders $\sqsubseteq_{\mathcal{F}}$ and $\preceq_{\mathcal{F}}$ are identical for function values $f \in \mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N}$. Pairings with the special elements $\top_{\mathcal{F}}$ and $\bot_{\mathcal{F}}$ are related to each other according to the Hasse diagrams in Figure 8.

(a) *computational order* $\sqsubseteq_{\mathcal{F}}$ (b) *approximation order* $\preceq_{\mathcal{F}}$

Figure 8: Hasse diagrams for $\sqsubseteq_{\mathcal{F}}$ and $\preceq_{\mathcal{F}}$

(a) t_1 (b) t_2

(c) $t_1 \sqcup_{\mathcal{T}} t_2$ (d) $t_1 \curlyvee_{\mathcal{T}} t_2$

Figure 9: Decision Tree Join Example

Definition 7.2. The *computation order* $\sqsubseteq_{\mathcal{F}}$ and *approximation order* $\preceq_{\mathcal{F}}$ for elements $f_1, f_2 \in \mathcal{F} \setminus \{\top_{\mathcal{F}},\ \bot_{\mathcal{F}}\}$ are defined as follows.

$$f_1 \sqsubseteq_{\mathcal{F}} f_2 \iff f_1 \preceq_{\mathcal{F}} f_2 \iff \forall x \in \mathbb{Z}^{|\mathcal{X}|}\colon f_1(x) \leq f_2(x)$$

7.2 Join

Two trees can be joined to form the union of all partitions represented by the two trees. When joining two trees, they are first reshaped such that both trees consist of the same partitions. They only differ in the values of the leafs. Then the two trees can be joined leaf-wise. There are two join variations. The *computational join* $\sqcup_{\mathcal{T}}\colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \to \mathcal{T}_{NIL}$ and the *approximation join* $\curlyvee_{\mathcal{T}}\colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \to \mathcal{T}_{NIL}$. Two leafs l_1 and l_2 are joined by taking the least upper bound of the two functions $l_1.f$ and $l_2.f$ w.r.t. $\sqsubseteq_{\mathcal{F}}$ ($\preceq_{\mathcal{F}}$, respectively). Figure 9 demonstrates the difference between the two join types. When joining two trees where one leaf is defined and one is undefined (see left leaf in t_1 and t_2), the *computational join* will preserve the defined leaf and the *approximation join* will make the leaf undefined. If one of the two leafs is `NIL` then they are joined by taking the one that is not `NIL`. When both are `NIL` then the result of joining the leafs is `NIL`.

7.3 Meet

The meet operator intersects the partitions of two decision trees. As with the join, both trees are first brought into the same shape such that they can be combined leaf-wise. There are two meet variations. The *computational meet* $\sqcap_{\mathcal{T}} : (\mathcal{T}_{\mathtt{NIL}} \times \mathcal{T}_{\mathtt{NIL}}) \rightarrow \mathcal{T}_{\mathtt{NIL}}$ and the *approximation meet* $\curlywedge_{\mathcal{T}} : (\mathcal{T}_{\mathtt{NIL}} \times \mathcal{T}_{\mathtt{NIL}}) \rightarrow \mathcal{T}_{\mathtt{NIL}}$. Both combine defined leafs using the least upper bound w.r.t. $\preceq_{\mathcal{F}}$. If at least one of the two leafs is `NIL`, then the result is $\bot_{\mathcal{F}}$ in case of the *computational meet* and `NIL` in case of the *approximation meet*.

7.4 Widening and Dual Widening

The *widening* operator $\nabla_{\mathcal{T}} : (\mathcal{T}_{\mathtt{NIL}} \times \mathcal{T}_{\mathtt{NIL}}) \rightarrow \mathcal{T}_{\mathtt{NIL}}$ is used to enforce convergence when computing increasing sequences of values in the decision tree abstract domain. Once the sequence is stable for the *computational order* $\sqsubseteq_{\mathcal{T}}$, the limit of this sequence is guaranteed to be a sound approximation of the corresponding concrete counting function w.r.t. the *approximation order* $\preceq_{\mathcal{F}}$. The dual of this concept for decreasing sequences is called *dual widening* $\bar{\nabla}_{\mathcal{T}} : (\mathcal{T}_{\mathtt{NIL}} \times \mathcal{T}_{\mathtt{NIL}}) \rightarrow \mathcal{T}_{\mathtt{NIL}}$. Intuitively, the widening operator tries to extrapolate the function values of decision trees to linear functions. This extrapolation is initially a guess. If the guess proves to be wrong, it is corrected by setting the corresponding partition to $\top_{\mathcal{T}}$ in future iterations. A thorough discussion about widening for elements of the decision tree abstract domain can be found in Section 5.2.4 of [6].

7.5 Filter

The filter operator $\mathtt{FILTER}_o[\![bexp]\!] : \mathcal{T}_{\mathtt{NIL}} \rightarrow \mathcal{T}_{\mathtt{NIL}}$ prunes all partitions of a decision tree that do not satisfy the given boolean expression *bexp*. Leafs are pruned by replacing them with `NIL`. Partitions are pruned using over-approximation. That means that the resulting tree can still be defined for states that do not satisfy the boolean expression. It is however guaranteed that all states that satisfy the boolean expression remain defined in the tree. In addition to the over-approximating version there is also the under-approximating version $\mathtt{FILTER}_u[\![bexp]\!] : \mathcal{T}_{\mathtt{NIL}} \rightarrow \mathcal{T}_{\mathtt{NIL}}$. Here the resulting decision tree is guaranteed to not contain any partitions that do not satisfy the boolean expressions. However states that do satisfy it might be removed if the underlying numerical domain is not expressive enough.

7.6 Backward Assign

The operator $\mathtt{ASSIGN}_o[\![X := \mathtt{aexp}]\!] : \mathcal{T}_{\mathtt{NIL}} \rightarrow \mathcal{T}_{\mathtt{NIL}}$ handles the backward assignment of the arithmetic expression `aexp` to variable X. The linear constraints of the decision tree nodes and the functions at the leafs are adjusted accordingly. $\mathtt{ASSIGN}_o$ uses over-approximation on the underlying numerical domains. As with the filter operator, there also exists an under-approximating version $\mathtt{ASSIGN}_u[\![X := \mathtt{aexp}]\!] \colon \mathcal{T}_{\mathtt{NIL}} \to \mathcal{T}_{\mathtt{NIL}}$.

8 Abstract Semantics for CTL

The counting function τ_Φ is in general not computable. In this section we present a sound and computable approximation of the *CTL semantics* τ_Φ defined in Section 5. We approximate τ_Φ by using the decision tree abstract domain (Section 7) to approximate counting functions in terms of piecewise defined ranking functions.

Theorem 8.1. The abstract CTL semantics $\tau_\Phi^\sharp \in \mathcal{L} \to \mathcal{T}$ is a sound approximation of the CTL semantics τ_Φ with regards to the *approximation order* $\preceq_\mathcal{T}$ (see Definition 7.1).

Recall that the CTL semantics $\tau_\Phi \in \Sigma \to \mathbb{N}$ is a partial function that assigns numerical values to program states $\sigma \in \Sigma$. In the abstract version, program states are grouped by program labels $l \in \mathcal{L}$ and partitioned by decision trees $t \in \mathcal{T}$. A program satisfies a given CTL property Φ if the decision tree of the initial program label $\tau_\Phi^\sharp(t_{init})$ is defined over all partitions of program states, i.e., all leafs of the decision tree are defined.

The following sections present how to compute $\tau_\Phi^\sharp$ for each CTL operator. We start with the basic operators $\wedge, \vee, \neg$ and atomic propositions. These can be computed directly for each program label. Then we present how to compute the universal $\forall(\cdot U \cdot)$, $\forall\bigcirc$ and $\forall\Box$ operators through fixed-point iteration. Followed by a discussion on how to adapt the universal operators to their existential version. Note that the abstract CTL semantics are computed recursively. The recursion stops at atomic propositions.

8.1 Path Independent Operators

Atomic propositions are path independent, therefore $\tau_a^\sharp$ assigns the same decision tree to each program label $l \in \mathcal{L}$. This decision tree assigns the constant function 0 to all partitions that satisfy the atomic proposition a. We compute this tree by using the $\mathtt{RESET}[\![a]\!]$ operator on the totally undefined decision tree $\bot_\mathcal{T}$. The $\mathtt{RESET}[\![a]\!] \colon \mathcal{T} \to \mathcal{T}$ operator takes as input a decision tree and returns a copy of said tree where every partition that satisfies a is replaced with the constant function 0. We refer the reader to [6] for a detailed description of the RESET operator. Note however, that the implementation of RESET in [6] has a small error that can lead to unsoundness. This is discussed in the excursion below titled 'RESET and over-approximation'.

Definition 8.1. Abstract CTL semantics for atomic propositions

$$\tau_a^\sharp \stackrel{\text{def}}{=} \lambda l.\ \mathtt{RESET}[\![a]\!]\bot_\mathcal{T} \tag{27}$$

Lemma 8.2. The abstract CTL semantics $\tau_a^\sharp$ is a sound approximation of the CTL semantics τ_a with regards to the *approximation order* $\preceq_\mathcal{T}$ (see Definition 7.1).

`RESET` and over-approximation

The $\texttt{RESET}[\![a]\!]$ operator was originally introduced in [6] in the context of abstract guarantee semantics and would over-approximate the set of partitions that satisfy the atomic proposition a. During the work on this thesis, we discovered that this original definition is actually unsound and leads to incorrect analysis results.

The problem is best described using an example. Consider the abstract CTL semantics $\tau^\sharp_{x^2<y^3+1}$. The non-linear constraint $x^2 < y^3 + 1$ can usually not be represented by any of the commonly used numerical domains. An over-approximating implementation of $\texttt{RESET}[\![x^2 < y^3 + 1]\!]$ will therefore reset some pairs (x, y) for which $x^2 < y^3 + 1$ does not hold which is unsound as to the definition of the CTL semantics $\tau_{x^2<y^3+1}$. Note that this problem propagates to more complex temporal properties that depend on atomic propositions.

We resolve this problem by using under-approximating on the underlying numerical domains to soundly determine which partitions satisfy the atomic proposition a.

Now we define the abstract CTL semantics for the logical operators $\wedge$, $\vee$ and $\neg$.

Definition 8.2. Abstract CTL semantics for logic operators

$$\tau^\sharp_{\Phi_1\wedge\Phi_2} \overset{\text{def}}{=} \lambda l.\ (\tau^\sharp_{\neg\Phi_1} l)\ \sqcup_\mathcal{T}\ (\tau^\sharp_{\neg\Phi_2} l) \tag{28}$$

$$\tau^\sharp_{\Phi_1\vee\Phi_2} \overset{\text{def}}{=} \lambda l.\ (\tau^\sharp_{\neg\Phi_1} l)\ \sqcap_\mathcal{T}\ (\tau^\sharp_{\neg\Phi_2} l) \tag{29}$$

$$\tau^\sharp_{\neg\Phi} \overset{\text{def}}{=} \lambda l.\ \texttt{COMPLEMENT}\ (\tau^\sharp_{\neg\Phi} l) \tag{30}$$

The abstract CTL semantics for $\wedge$ and $\vee$ (Equations 28 and 29) combine the decision trees of the nested properties piecewise for each program label.

The *computational join* $\sqcup_\mathcal{T}$ is used to combine the two trees (see Section 7.2) in case of the logical $\vee$. This operator forms the union of the two decision trees. Note that we use the computational version of the join operator to include partitions that are defined in at least one of the two trees. If a partition is defined in both trees, the least upper bound of the two functions assigned to that partition is used w.r.t. to the partial order $\sqsubseteq_\mathcal{F}$ given in Definition 7.2.

Algorithm 1 Tree Complement

```
function COMPLEMENT(t)                                  ▷ t ∈ 𝒯_NIL
    if (isNode(t) ∧ t.f = ⊤) ∨ isNil(t) then
        return t                                        ▷ ignore ⊤ and NIL
    else if isLeaf(t) ∧ t.f = ⊥ then
        return LEAF : 0                                 ▷ undefined becomes defined
    else if isLeaf(t) then
        return LEAF : ⊥                                 ▷ defined becomes undefined
    else
        l ← COMPLEMENT(t.l)
        r ← COMPLEMENT(t.r)
        return NODE{t.c} : l; r
    end if
end function
```

The corresponding definition for the logical $\wedge$ operator forms the intersection of the two trees by using the *computational meet* $\sqcap_{\mathcal{T}}$ (see Section 7.3). By using the computational version of the meet, we ensure that no `NIL` leafs are introduced when forming the intersection. As with the logical $\vee$, the least upper bound w.r.t. to $\sqsubseteq_{\mathcal{F}}$ is used if a partition is defined in both trees.

For the logical $\neg$ operator we introduce the `COMPLEMENT`: $\mathcal{T}_{\texttt{NIL}} \rightarrow \mathcal{T}_{\texttt{NIL}}$ operator. This operator replaces all defined leafs with a $\perp$-leaf and all $\perp$-leafs with the constant function 0. By doing so, all states that originally satisfied the property do not satisfy it any more and vice versa. However one has to be careful when changing a partition from undefined to defined. Decision trees are an approximation of the concrete CTL semantics. Therefore not all states that are undefined in the abstract decision tree are actually undefined in the concrete ranking function. Partitions that are undefined because of this uncertainty are marked with a $\top$-leaf. To ensure soundness, these leafs have to be ignored when forming the complement of a decision tree. The `COMPLEMENT` operator is implemented in Algorithm 1.

Lemma 8.3. The abstract CTL semantics $\tau^{\sharp}_{\Phi_1 \wedge \Phi_2}$, $\tau^{\sharp}_{\Phi_1 \vee \Phi_2}$ and $\tau^{\sharp}_{\neg\Phi}$ are a sound approximation of the CTL semantics $\tau_{\Phi_1 \wedge \Phi_2}$, $\tau_{\Phi_1 \vee \Phi_2}$ and $\tau_{\neg\Phi}$ with regards to the *approximation order* $\preceq_{\mathcal{T}}$ (see Definition 7.1).

8.2 Path Dependent Operators

In this section we describe how the abstract CTL semantics for the path-dependent operators ($\forall \bigcirc \Phi$, $\exists \bigcirc \Phi$, $\forall(\Phi_1 U \Phi_2)$, $\exists(\Phi_1 U \Phi_2)$, $\forall \Box \Phi$, $\exists \Box \Phi$) are defined.

First, we define the two functions $[\![stmt]\!]_o \in \mathcal{T}_{NIL} \to \mathcal{T}_{NIL}$ and $[\![stmt]\!]_u \in \mathcal{T}_{NIL} \to \mathcal{T}_{NIL}$. The first one uses over-approximation on the underlying numerical domains, the second one under-approximation.

Both functions implement the effect of backward propagating an edge in the control-flow-graph, i.e., the effect of executing a statement. Assume that we have computed a decision tree for the target node of some edge. This decision tree represents the value of the counting function for this node. By applying $[\![\cdot]\!]$ to this tree, we compute the decision tree that holds before executing the statement, i.e. the value of the function at the source node of this edge.

Definition 8.3. Abstract semantics for basic statements

$$\begin{aligned}
[\![\texttt{skip}]\!]_o &\stackrel{\text{def}}{=} \lambda t.\ \texttt{STEP}(t) \\
[\![bexp]\!]_o &\stackrel{\text{def}}{=} \lambda t.\ \texttt{ASSIGN}_o(t) \\
[\![X := aexp]\!]_o &\stackrel{\text{def}}{=} \lambda t.\ \texttt{FILTER}_o(t) \\
\\
[\![\texttt{skip}]\!]_u &\stackrel{\text{def}}{=} \lambda t.\ \texttt{STEP}(t) \\
[\![bexp]\!]_u &\stackrel{\text{def}}{=} \lambda t.\ \texttt{ASSIGN}_u(t) \\
[\![X := aexp]\!]_u &\stackrel{\text{def}}{=} \lambda t.\ \texttt{FILTER}_u(t)
\end{aligned}$$

The `skip` statement is handled by the `STEP` operator. This operator increases the value of all defined partitions in the decision tree by one. Recall that a defined partition in a decision tree represents a set of states that satisfies some CTL property. The associated value is an upper bound on the number of steps until some condition is reached. By executing `skip` this number is incremented by one. For assignments and boolean conditions we use the corresponding *assign* and *filter* operators that were introduced in Sections 7.6 and 7.5. The definitions for the remaining path dependent operators all depend on these two functions.

Until

The abstract CTL semantics for universal and existential until properties are defined as the least fixed-point of the abstract transformers

$$\begin{aligned}
\phi^\sharp_{\forall(\Phi_1 U \Phi_2)} &\in (\mathcal{L} \to \mathcal{T}_{NIL}) \to (\mathcal{L} \to \mathcal{T}_{NIL}) \\
\phi^\sharp_{\exists(\Phi_1 U \Phi_2)} &\in (\mathcal{L} \to \mathcal{T}_{NIL}) \to (\mathcal{L} \to \mathcal{T}_{NIL})
\end{aligned}$$

starting from the function $\tau_\bot$ that assigns the decision tree $\bot_\mathcal{T}$ to every node in the control-flow-graph. $\forall l \in \mathcal{L}\colon \tau_\bot(l) = \bot_\mathcal{T}$.

Definition 8.4. Abstract semantics for until operator.

$$\begin{aligned}
\tau^\sharp_{\forall(\Phi_1 U \Phi_2)} &\stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq_\mathcal{T}}_{\mathcal{T}_\perp} \phi^\sharp_{\forall(\Phi_1 U \Phi_2)} \\
t_\curlyvee(m,l) &\stackrel{\text{def}}{=} \bigcurlyvee_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_o(m(l')) \\
\phi^\sharp_{\forall(\Phi_1 U \Phi_2)}(m)l &\stackrel{\text{def}}{=} \texttt{UNTIL}[\![\tau^\sharp_{\Phi_1}(l), \tau^\sharp_{\Phi_2}(l)]\!](t_\curlyvee(m,l))
\end{aligned}$$

$$\begin{aligned}
\tau^\sharp_{\exists(\Phi_1 U \Phi_2)} &\stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq_\mathcal{T}}_{\perp_\mathcal{T}} \phi^\sharp_{\exists(\Phi_1 U \Phi_2)} \\
t_\sqcup(m,l) &\stackrel{\text{def}}{=} \bigsqcup_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_u(m(l')) \\
\phi^\sharp_{\exists(\Phi_1 U \Phi_2)}(m)l &\stackrel{\text{def}}{=} \texttt{UNTIL}[\![\tau^\sharp_{\Phi_1}(l), \tau^\sharp_{\Phi_2}(l)]\!](t_\sqcup(m,l))
\end{aligned}$$

We will first discuss the universal version and then explain what changes for the existential case. The value $m \in \mathcal{L} \to \mathcal{T}_{NIL}$ is the current iteration value of the fixed-point iteration, $\phi^\sharp_{\forall(\Phi_1 U \Phi_2)}(m)$ describes the value of the next iteration. Recall that $out(l)$ denotes all outgoing edges of node l leading to its immediate successors nodes. Every edge is labeled with a statement. The abstract transformer $\phi^\sharp_{\forall(\Phi_1 U \Phi_2)}$ computes decision trees point-wise for each node l in the control-flow-graph, based on the decision trees of its successor nodes.

First, the decision tree of each successor node l' is passed to the $[\![stmt]\!]_o$ function. This approximates the effect of transitioning from l to l'. The resulting decision tree approximates the value of the ranking function before executing the statement.

If a node has multiple successor nodes then the resulting decision trees are combined using the *approximation join* $\curlyvee$. The *approximation join* discards all partitions (i.e., makes them undefined) of decision trees that are not defined for all successor nodes. By doing so, we approximate the semantic of the universal path quantifier $\forall$. Note that if a node has no successors then $t_\curlyvee(m,l) = \perp_\mathcal{T}$ since the least upper bound (join) of the empty set is $\perp_\mathcal{T}$.

We use the over-approximating version of the $[\![\cdot]\!]_o$ function. This might temporarily lead to unsound decision trees due to over-approximation. Decision trees produced by $[\![\cdot]\!]_o$ can contain defined partitions for states that are unfeasible among that path in the control-flow-graph. For the universal case however, this is not a problem since the *approximation join* only keeps those partitions which are feasible among all paths. Partitions that are unfeasible among some paths are discarded.

Finally the result of joining the decision trees of the immediate predecessors are applied to he $\texttt{UNTIL}[\![\tau^\sharp_{\Phi_1}, \tau^\sharp_{\Phi_2}]\!]$ operator. The purpose of his operator is to implement the semantics of the until CTL operator. All partitions that satisfy Φ_1 are set to zero and all partitions that neither satisfy Φ_1 nor Φ_2 are discarded (see Algorithm 3). That way we end up with a decision tree that is only defined for the partitions that satisfy $\forall(\Phi_1 U \Phi_2)$.

The abstract transformer for the existential case follows the same structure as in the universal case. However instead of using the *approximation join* it uses the *computational join* $\sqcup_\mathcal{T}$ to approximate the semantics of the $\exists$ path quantifier. The *computational join* preserves all partitions that are defined for at least one decision tree. Note however, that all decision trees passed to the *computation join* must be sound since we can no longer rely on the join operator to discard unfeasible partitions. Therefore we apply the under-approximating $[\![stmt]\!]_u$ function when processing statements to guarantee soundness.

Lemma 8.4. The abstract CTL semantics $\tau^\sharp_{\forall(\Phi_1 U \Phi_2)}$ and $\tau^\sharp_{\exists(\Phi_1 U \Phi_2)}$ are a sound approximation of the CTL semantics $\tau_{\forall(\Phi_1 U \Phi_2)}$ and $\tau_{\exists(\Phi_1 U \Phi_2)}$ with regards to the *approximation order* $\preceq_\mathcal{T}$ (see Definition 7.1).

Convergence of the least fixed-point iteration is guaranteed after a finite amount of iterations by using the widening operator $\triangledown_\mathcal{T}$ (see Section 7.4). The decision tree computed for every node $l \in \mathcal{L}$ is guaranteed to converge by applying the folowing widening scheme (Φ_U is a placeholder for $\forall(\Phi_1 U \Phi_2)$ and $\exists(\Phi_1 U \Phi_2)$):

$$
\begin{aligned}
y_0 &\stackrel{\mathsf{def}}{=} \bot_\mathcal{T} \\
y_{n+1} &\stackrel{\mathsf{def}}{=} \begin{cases} y_n & \text{if } \phi^\sharp_{\Phi_U}(y_n) \sqsubseteq_\mathcal{T} y_n \wedge \phi^\sharp_{\Phi_U}(y_n) \preceq_\mathcal{T} y_n \\ y_n \ \triangledown_\mathcal{T}\ \phi^\sharp_{\Phi_U}(y_n) & \text{otherwise} \end{cases}
\end{aligned}
$$

Algorithm 2 Tree Until Filter

```
function FILTER_UNTIL(t, t_valid)
    if isNil(t) ∨ isNil(t_valid) then
        ▷ ignore NIL nodes
        return t
    else if isLeaf(t) ∧ isLeaf(t_valid) ∧ isDefined(t) then
        ▷ t is defined in t_valid
        return t
    else if isLeaf(t) ∧ isLeaf(t_valid) ∧ ¬isDefined(t) then
        ▷ t is not defined in t_valid, make undefined
        return LEAF : ⊥
    else
        l ← FILTER_UNTIL(t.l, t_valid.l)
        r ← FILTER_UNTIL(t.r, t_valid.r)
        return NODE{t.c} : l; r
    end if
end function
```

Global

The abstract CTL semantics for universal and existential global properties are defined as the greatest fixed-point of the abstract transformers

$$\phi^\sharp_{\forall\Box\Phi} \in (\mathcal{L} \to \mathcal{T}_{NIL}) \to (\mathcal{L} \to \mathcal{T}_{NIL})$$
$$\phi^\sharp_{\exists\Box\Phi} \in (\mathcal{L} \to \mathcal{T}_{NIL}) \to (\mathcal{L} \to \mathcal{T}_{NIL})$$

starting from the abstract CTL semantics $\tau^\sharp_\Phi$ of the inner CTL property Φ.

Algorithm 3 Tree Until

function RESET_UNTIL(t, t_{reset})
  if $isNil(t) \vee isNil(t_{\text{reset}})$ **then**
    ▷ ignore NIL nodes
    return t
  else if $isLeaf(t) \wedge isLeaf(t_{\text{reset}}) \wedge isDefined(t)$ **then**
    ▷ t is defined in t_{valid}, reset leaf
    return $LEAF : 0$
  else if $isLeaf(t) \wedge isLeaf(t_{\text{valid}}) \wedge \neg isDefined(t)$ **then**
    ▷ t is undefined in t_{valid}, keep as is
    return t
  else
    $l \leftarrow$ RESET_UNTIL($t.l, t_{\text{reset}}.l$)
    $r \leftarrow$ RESET_UNTIL($t.r, t_{\text{reset}}.r$)
    return $NODE\{t.c\} : l; r$
  end if
end function
function UNTIL$[\![t_{\Phi_1}, t_{\Phi_2}]\!](t)$ ▷ $t, t_{\Phi_1}, t_{\Phi_2} \in \mathcal{T}_{NIL}$
  $(t_1, t_2) \leftarrow$ TREE_UNIFICATION($t, t_{\Phi_1} \sqcup t_{\Phi_2}$)
  $t_{\text{filtered}} \leftarrow$ FILTER_UNTIL(t_1, t_2)
  $(t_1, t_2) \leftarrow$ TREE_UNIFICATION($t_{\text{filtered}}, t_{\Phi_2}$)
  return RESET_UNTIL(t_1, t_2)
end function

Definition 8.5. Abstract semantics for global operator.

$$\tau^{\sharp}_{\forall\Box\Phi} \overset{\text{def}}{=} \text{gfp}^{\sqsubseteq_{\mathcal{T}}}_{\tau^{\sharp}_{\Phi}} \phi^{\sharp}_{\forall\Box\Phi}$$

$$t_{\curlyvee}(m, l) \overset{\text{def}}{=} \bigcurlyvee_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_o(m(l'))$$

$$\phi^{\sharp}_{\forall\Box\Phi}(m)l \overset{\text{def}}{=} \mathtt{MASK}[\![t_{\curlyvee}(l, m)]\!](m(l))$$

$$\tau^{\sharp}_{\exists\Box\Phi} \overset{\text{def}}{=} \text{gfp}^{\sqsubseteq_{\mathcal{T}}}_{\tau^{\sharp}_{\Phi}} \phi^{\sharp}_{\exists\Box\Phi}$$

$$t_{\sqcup}(m, l) \overset{\text{def}}{=} \bigsqcup_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_u(m(l'))$$

$$\phi^{\sharp}_{\exists\Box\Phi}(m)l \overset{\text{def}}{=} \mathtt{MASK}[\![t_{\sqcup}(l, m)]\!](m(l))$$

The value $m \in \mathcal{L} \rightarrow \mathcal{T}_{NIL}$ is the current iteration value of the fixed-point iteration. The definition for $\phi^{\sharp}_{\forall\Box\Phi}(m)$ ($\phi^{\sharp}_{\exists\Box\Phi}(m)$, respectively) describes the value of the next iteration. We use the same approach as the until operator to join decision trees of successor nodes (see $t_{\curlyvee}(m,l)$ and $t_{\sqcup}(m,l)$).

In the final step however, the decision tree $m(l)$ of each node l is *masked* with the result of $t_{\curlyvee}(l,m)$ ($t_{\sqcup}(l,m)$ respectively). Masking is implemented by the `MASK` operator. This operator sets all partitions of $m(l)$ to $\bot_{\mathcal{T}}$, that are not defined in $t_{\curlyvee}(l,m)$ ($t_{\sqcup}(l,m)$, respectively). That way all states that do not satisfy Φ indefinitely among all (some, respectively) paths are iteratively removed from the decision tree until a fixed-point is reached. The `MASK` operator is defined in Algorithm 4.

Lemma 8.5. The abstract CTL semantics $\tau^{\sharp}_{\forall\Box\Phi}$ and $\tau^{\sharp}_{\exists\Box\Phi}$ are a sound approximation of the CTL semantics $\tau_{\forall\Box\Phi}$ and $\tau_{\exists\Box\Phi}$ with regards to the *approximation order* $\preceq_{\mathcal{T}}$ (see Definition 7.1).

Convergence of the greatest fixed-point iteration is guaranteed after a finite amount of iterations by using the dual widening operator $\bar{\nabla}_{\mathcal{T}}$ (see Section 7.4). The decision tree computed for every node $l \in \mathcal{L}$ is guaranteed to converge by applying the folowing widening scheme (Φ_G is a placeholder for $\forall\Box\Phi$ and $\exists\Box\Phi$):

$$y_0 \stackrel{\mathsf{def}}{=} \tau^{\sharp}_{\Phi_U}(l)$$

$$y_{n+1} \stackrel{\mathsf{def}}{=} \begin{cases} y_n & \text{if } y_n \sqsubseteq_{\mathcal{T}} \phi^{\sharp}_{\Phi_G}(y_n) \wedge y_n \preceq_{\mathcal{T}} \phi^{\sharp}_{\Phi_G}(y_n) \\ y_n \ \bar{\nabla}_{\mathcal{T}} \ \phi^{\sharp}_{\Phi_G}(y_n) & \text{otherwise} \end{cases}$$

Next

The abstract CTL semantics for the next operator are given in Definition 8.6:

Algorithm 4 Tree Mask

> **function** $\textsc{mask}[\![t_{\textsc{mask}}]\!](t)$
>   **function** $\textsc{mask\_aux}(t, t_{\text{mask}})$
>   **if** $isNil(t) \vee isNil(t_{\text{reset}})$ **then**
>   ▷ ignore NIL nodes
>   **return** t
>   **else if** $isLeaf(t) \wedge isDefined(t) \wedge isLeaf(t_{\text{mask}})$ **then**
>   **if** $isDefined(t) \wedge \neg isDefined(t_{\text{mask}})$ **then**
>   ▷ t is defined and t_{mask} is undefined, discard leaf
>   **return** $LEAF : \bot$
>   **else**
>   **return** t
>   **end if**
>   **else**
> $l \leftarrow \textsc{mask}(t.l, t_{\text{mask}}.l)$
> $r \leftarrow \textsc{mask}(t.r, t_{\text{mask}}.r)$
>   **return** $NODE\{t.c\} : l; r$
>   **end if**
>   **end function**
> $(t_1, t_2) \leftarrow \textsc{tree\_unification}(t, t_{\text{MASK}})$
>   **return** $\textsc{mask\_aux}(t_1, t_2)$
> **end function**

Definition 8.6. Abstract semantics for next operator.

$$t_{\curlyvee}(l) \stackrel{\text{def}}{=} \bigcurlyvee_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_o(\tau^\sharp_\Phi(l))$$

$$\tau^\sharp_{\forall\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda l.\ \texttt{ZERO}(t_{\curlyvee}(l))$$

$$t_{\sqcup}(l) \stackrel{\text{def}}{=} \bigsqcup_{(l,stmt,l') \,\in\, out(l)} [\![stmt]\!]_u(\tau^\sharp_\Phi(l))$$

$$\tau^\sharp_{\exists\bigcirc\Phi} \stackrel{\text{def}}{=} \lambda l.\ \texttt{ZERO}(t_{\sqcup}(l))$$

As opposed to the until and global operator, the decision trees for each label only depends on the immediate successor nodes. Therefore no fixed-point iteration is necessary. Each node is computed in one step based on the immediate successor nodes. Outgoing edges are joined as describe for the until and global operators. The resulting value is then passed to the `ZERO` operator which sets all defined partitions to zero (see Algorithm 5).

Lemma 8.6. The abstract CTL semantics $\tau^{\sharp}_{\forall\bigcirc\Phi}$ and $\tau^{\sharp}_{\exists\bigcirc\Phi}$ are a sound approximation of the CTL semantics $\tau_{\forall\bigcirc\Phi}$ and $\tau_{\exists\bigcirc\Phi}$ with regards to the *approximation order* $\preceq_{\mathcal{T}}$ (see Definition 7.1).

Algorithm 5 Tree Zero

function ZERO(t)
  if $isLeaf(t) \wedge isDefined(t)$ **then**
    return $LEAF : 0$
  else if $isNode(t)$ **then**
    $l \leftarrow$ ZERO($t.l$)
    $r \leftarrow$ ZERO($t.r$)
    return $NODE\{t.c\} : l; r$
  else
    return t
  end if
end function

9 Implementation

This section describes how the CTL analysis was implemented for the FuncTion Static Analyzer[1]. First, we briefly discuss the initial feature set of FuncTion, followed by a description of the various extensions that were added to the analyzer.

9.1 Previous Work

FuncTion is an abstract interpretation based static analyzer that was written by Caterina Urban in the context of her PhD thesis [6]. When starting this thesis, FuncTion had support for analyzing termination, guarantee properties and recurrence properties (see Section 4). To that end, it featured an implementation of the decision tree abstract domain (see Section 7) with the corresponding *join*, *meet*, *widen* (but not *dual-widen*), *filter*, *backward assign* and *reset* operators. Note that the *filter*, backward assign and reset operators were all implemented using over-approximation. The Apron[2] library was used as basis for the numerical domains. FuncTion can analyze programs using the *interval*, *octagon* and *polyhedra* abstract domain. Finally,

[1] https://github.com/caterinaurban/function
[2] http://apron.cri.ensmp.fr/library

FuncTion was equipped with a front-end for parsing programs written in a C-like language. The termination analysis could only tail-recursive function calls. The analysis of guarantee and recurrence properties could could not handle any function calls.

9.2 CTL Analysis

The CTL analysis was implemented based on the existing components of FuncTion. We reused the implementation of the decision tree abstract domain and added implementations for the *complement*, *until* and *mask* operators that we presented in Section 8. We also wrote an implementation of the *dual-widen* operator which was defined in [6] but not actually implemented.

The under-approximating implementations of the *filter* and *backward assign* required an under-approximating implementation of said operators on the underlying numerical abstract domains. Such an implementation for the polyhedra abstract domain was proposed by Antoine Miné [10] and implemented by him in the Banal Static Analyzer[3]. We were able to reuse his implementation in FuncTion which allowed us to provide under-approximating version of *filter* and *backward assign* with minimal effort. In addition to that, the under-approximating version of *filter* also allowed us to fix the unsoundness problem of the *reset* operator which we discussed in Section 8.

The inital implementation of the CTL analysis in FuncTion reuses the existing front-end and performs the analysis through a backward analysis on the abstract-syntax-tree. This had the drawback that the analyzer did not support control-flow statements such as `goto`, `continue` or `break`. Furthermore there was no support for function calls. These problems were later resolved by converting the analysis to a control-flow-graph based implementation. This will be discussed in the next subsection.

9.3 Improved Front-end

As mentioned previously the abstract-syntax-tree (AST) based CTL analysis had several drawbacks. Performing the analysis on the AST made it difficult to support `goto` statements and function calls. Therefore we changed the analysis to a control-flow-graph based analysis. To that end we reused an existing front-end[4] for a university course by Antoine Miné which features an abstract-syntax-tree to control-flow-graph conversion.

[3]https://www-apr.lip6.fr/~mine/banal/

[4]https://www-apr.lip6.fr/~mine/enseignement/l3/2015-2016/project.html

The new analysis implementation is based on a standard worklist algorithm [11]. Nodes to be proccessed are added to a worklist and processed in FIFO order. If the value of a node changes, all its predecessors are added to the worklist. To improve performance, nodes are never added twice to the worklist.

Widening is only performed at loop heads. To that end, we implemented a simple loop-detection algorithm. First we check if the control-flow-graph is reducible. Then we detect loop heads by finding edges in the control-flow-graph where the head dominates its tail. The dominator tree is computed with a simple $O(mn)$ algorithm. Better algorithms exists [12], however at the time of implementing the static analyzer, performance was not an issue. More information about loop-detection can be found in the corresponding literature [13].

Non-recursive function calls are handled by inlining. Recursive calls are resolved by adding edges to the control-flow-graph from every call site to the entry node of the function and from the exit node of the function back to the call sites. This approach leads to infeasible paths in the control-flow-graph. However the result of the analysis is still sound. We did not pursue more involved intra-procedural analysis approaches since this was not the focus the thesis.

Existential CTL properties can be analyzed using the abstract semantics presented in Section 8. In addition to that, it is also possible to convert existential properties to equivalent universal properties (see Section 3). This can yield more precise results in some cases.

10 Evaluation

This section presents the result of our evaluation of the CTL analysis capabilities of FuncTion. First we present a few examples that demonstrate the kinds of properties FuncTion is able to analyze. Then we present how FuncTion fares against other static analyzers.

```
while (x > y) {
    x--;
}
```

Figure 10: Program test_until

Let us consider the program test_until. If we assume that initially $x \geq y$

No	Program	Property
1.1	and_test.c	$\forall\Box\forall\Diamond(n=1) \wedge \forall\Diamond(n=0)$
1.2	and_test.c	$\forall\Box\forall\Diamond(n=1)$
1.3	and_test.c	$\forall\Diamond(n=0)$
1.4	global_test_simple.	$\forall\Box\forall\Diamond(x \leq -10)$
1.5	multi_branch_choice.c	$\forall\Diamond(x=4 \vee x=-4)$
1.6	multi_branch_choice.c	$\exists\Diamond(x=-4)$
1.7	or_test.	$\forall\Diamond\forall\Box(x<-100) \vee \forall\Diamond(x=20)$
1.8	potential_termination.c	$\exists\Diamond(exit : \texttt{true})$
1.9	until_test.c	$\forall(x \geq y\ U\ x=y)$
1.10	existential_test_1	$\exists\Diamond(r=1)$
1.11	existential_test_3	$\exists\Diamond(r=1)$
1.12	existential_test_4	$\exists\Diamond(r=1)$
2.1	acqrel.c	$\forall\Box((a=1) \Rightarrow \forall\Diamond(r=1))$
2.2	acqrel_mod.c	$\forall\Box((a=1) \Rightarrow \forall\Diamond(r=1))$
2.3	fig8-2007.c	$(set=1) \Rightarrow \forall\Diamond(unset=1)$
2.4	fig8-2007_mod.c	$(set=1) \Rightarrow \forall\Diamond(unset=1)$
2.5	win4.c	$\forall\Diamond\forall\Box(WItemsNum \geq 1)$
2.6	toylin.c	$(c \leq 5 \wedge c>0) \vee \forall\Diamond(resp>5)$
3.1	coolant_basis_1_safe_sfty.c	$\forall\Box((chainBroken=1) \Rightarrow \forall\Box(chainBroken=1))$
3.2	coolant_basis_1_unsafe_sfty.c	$\neg\forall\Box((chainBroken=1) \Rightarrow \forall\Box(chainBroken=1))$
3.3	coolant_basis_2_safe_lifeness.c	$\forall\Box\forall\Diamond(otime<time)$
3.4	coolant_basis_2_unsafe_lifeness.c	$\neg\forall\Box\forall\Diamond(otime<time)$
3.5	coolant_basis_3_safe_sfty.c	$\forall\Box((init=3) \Rightarrow \forall\Box\forall\Diamond(time>otime))$
3.6	coolant_basis_3_unsafe_sfty.c	$\neg\forall\Box((init=3) \Rightarrow \forall\Box\forall\Diamond(time>otime))$
3.7	coolant_basis_4_safe_sfty.c	$\forall\Box(init \neq 3 \vee tem \leq limit \vee \forall\Diamond\forall\Box(chainBroken=1))$
3.8	coolant_basis_4_unsafe_sfty.c	$\neg\forall\Box(init \neq 3 \vee tem \leq limit \vee \forall\Diamond\forall\Box(chainBroken=1))$
3.9	coolant_basis_5_safe_sfty.c	$\forall(init=0\ U\ (\forall(init=1\ U\ \forall\Box(init=3)) \vee \forall\Box(init=1)))$
3.10	coolant_basis_5_safe_cheat.c	$\forall(init=0\ U\ (\forall(init=1\ U\ \forall\Box(init=3)) \vee \forall\Box(init=1)))$
3.11	coolant_basis_5_unsafe_sfty.c	$\neg\forall(init=0\ U\ (\forall(init=1\ U\ \forall\Box(init=3)) \vee \forall\Box(init=1)))$
3.12	coolant_basis_6_safe_sfty.c	$\forall\Box(limit \leq -273 \vee limit \geq 10 \vee tempIn \geq 0 \vee \forall\Diamond(warndLED=1))$
3.13	coolant_basis_6_unsafe_sfty.c	$\neg\forall\Box(limit \leq -273 \vee limit \geq 10 \vee tempIn \geq 0 \vee \forall\Diamond(warndLED=1))$
4.1	Bangalore_false-no-overflow.c	$\exists\Diamond(x<0)$
4.2	Ex02...	$i<5 \Rightarrow \forall\Diamond(exit : \texttt{true})$
4.3	Ex07...	$\forall\Diamond\forall\Box(i=0)$
4.4	java_Sequence...	$\forall\Diamond(\forall\Diamond(j \geq 21) \wedge i=100)$
4.4	Madrid...	$\forall\Diamond(x=7 \wedge \forall\Diamond\forall\Box(x=2))$
4.4	NO_02...	$\forall\Diamond\forall\Box(j=0)$

Table 1: Table of test cases. Lists the number, program name and CTL property to be analyzed.

holds, one can show that the program satisfies the property $\forall(x \geq y\ U\ x = y)$. FuncTion is able to infer the following counting function for the entry point of the program:

$$f(x,y) = \begin{cases} 2(x-y) & \text{if } x \geq y \\ \text{undefined} & \text{otherwise} \end{cases}$$

It states that the program satisfies the CTL properties if $x \geq y$ holds initially. Furthermore, the goal of the *until* property, $x = y$, will be reached in at most $2(x - y)$ program execution steps.

11 Conclusion

No	FuncTion	T2	Ultimate LTL
1.1	success	fail	success
1.2	success	fail	success
1.3	success	success	success
1.4	success	error	success
1.5	success	success	success
1.6	success	success	-
1.7	success	error	success
1.8	success	success	-
1.9	success	fail	success
1.10	success	success	-
1.11	success	success	-
1.12	success	success	-
2.1	fail	success	success
2.2	success	success	success
2.1	fail	success	success
2.2	success	success	success
2.2	success	-	success
2.2	fail	-	success
3.1	success	-	success
3.2	success	-	success
3.3	success	-	success
3.4	success	-	success
3.5	success	-	success
3.6	success	-	success
3.7	success	-	success
3.8	success	-	success
3.9	fail	-	success
3.10	success	-	success
3.11	success	-	success
3.12	success	-	success
3.13	success	-	success
4.1	success	success	-
4.2	success	out of memory	-
4.3	success	fail	-
4.4	success	error	-
4.5	success	error	-
4.6	success	success	-

Table 2: Table with results of test cases for FuncTion, T2 and Ultimate LTL Automizer.

References

[1] Caterina Urban and Antoine Miné, "Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation," in *VMCAI*, 2015, pp. 190–208.

[2] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen, *Principles of model checking*, MIT press, 2008.

[3] Alan Touring, "Checking a large routing," *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69, 1949.

[4] Robert W. Floyd, "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, pp. 19:19–32, 1967.

[5] P. Cousot and R. Cousot, "An abstract interpretation framework for termination," in *Conference Record of the 39*$^{\mathrm{th}}$ *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, PA, Jan. 25-27 2012, pp. 245–258, ACM Press, New York.

[6] Caterina Urban, *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.*, Ph.D. thesis, École Normale Supérieure, Paris, France, 2015.

[7] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming.* 1976, pp. 106–130, Dunod, Paris, France.

[8] Antoine Miné, "The octagon abstract domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, Mar. 2006.

[9] Patrick Cousot and Nicolas Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 1978, POPL '78, pp. 84–96, ACM.

[10] Antoine Miné, "Inferring sufficient conditions with backward polyhedral under-approximations," *Electr. Notes Theor. Comput. Sci.*, vol. 287, pp. 89–100, 2012.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, *Principles of program analysis*, Springer, 1999.

[12] Thomas Lengauer and Robert Endre Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, Jan. 1979.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.