Proving Temporal Properties by Abstract Interpretation

Samuel Marco Ueltschi

September 6, 2017

Contents

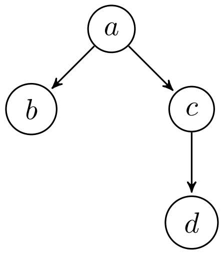


Figure 1: A basic state transition system

1 Introduction

Motivation etc.

2 State Transition Systems

To be able to analyse the behaviour 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 $\langle \Sigma, \tau \rangle$. $pre\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \in \mathcal{P}(\Sigma)$ to the set of their predecessors with respect to the program transition relation τ:*

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

Definition 2.3 *Given a transition system $\langle \Sigma, \tau \rangle$. $\widetilde{pre}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ 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:*

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

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.

3 Computation Tree Logic (CTL)

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

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 a placeholder for arbitrary atomic propositions. Formulae with quantifiers $\exists$ or $\forall$ are called path-dependent, formulae without 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 depends on the semantics of the underlying logic for atomic propositions.

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

The states $\sigma \in \Sigma$ are part of a state transition system $\langle \Sigma, \tau \rangle$ and $Paths(\sigma_0)$ 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 defined for $\models$ as they are equivalent to $\forall(true \; U \; \Phi)$ and $\exists(true \; U \; \Phi)$. Furthermore the following useful equivalence relations exists which can be used to relate existential to universal CTL formulae.

$$
\begin{aligned}
\exists \bigcirc \Phi &\equiv \neg\forall \bigcirc (\neg\Phi)\\
\exists\Diamond\Phi &\equiv \neg\forall\Box(\neg\Phi)\\
\exists\Box\Phi &\equiv \neg\forall\Diamond(\neg\Phi)
\end{aligned}
$$

3.3 Recurrence and Guarantee Properties

TODO

4 Ranking Functions

The traditional approach for proving termination is based on inferring *ranking functions* [3] [4]. A ranking function is a partial function from program states to a well-ordered set. For simplicity reasons we will use natural numbers as example. To prove termination, the values of the ranking functions must decrease during program execution. The value that a *ranking function* assigns to a state is an upper bound on the number of steps until the program terminates. Cousot and Cousot prove the existence of a *most precise ranking function* then can be derived by abstract interpretation [5]. The theory of abstract interpretation makes it possible to express various aspects of the semantics of a program. In that context the *most precise ranking function* for termination is called the *termination semantics*.

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

By definition of the *termination semantics*, a program will terminate if its initial state is in the domain of the ranking function. In other words, if the termination semantics assigns an upper bound on the number of steps until termination starting from the initial state.

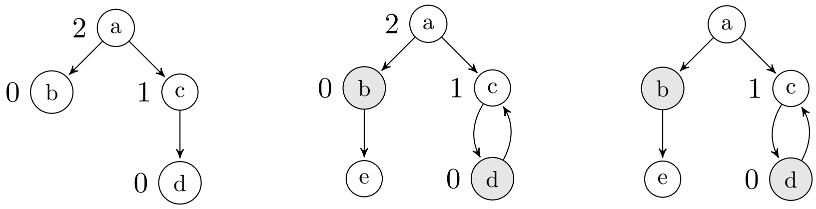
Based on the work of Cousot and Cousot [5]. Urban and Miné [1] extended the *termination semantics* to the more general notion of guarantee properties. A guarantee property states that some state satisfying a given property is guaranteed to be reached eventually. Termination is therefore just a guarantee property stating that some final state will be reached eventually. 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* $\tau^g_{[S]} \in \Sigma \to \mathbb{N}$ *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* $s \in S$ *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 a state satisfying the property is reached the next time. Recurrence properties can be expressed using the CTL formula $\forall\Box\forall\Diamond(a)$.

Definition 4.3 *The recurrence semantics is a ranking function* $\tau^r_{[S]} \in \Sigma \to \mathbb{N}$ *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 the ranking functions by labeling the states in the transition systems with the corresponding value assigned to them by the ranking functions. 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.



(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)

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 would hold 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.

5 Concrete Semantics for CTL

In section 4 we introduced the concept of ranking functions and explained how they express the semantics of recurrence and guarantee properties. To be able to analyse CTL properties, we extend the notion of ranking functions to CTL. Urban et al. [1] define ranking functions for guarantee and recurrence properties. The *CTL semantics* presented here are an extension of this work. We will define the *CTL semantics* inductively for each CTL operator such that arbitrary combinations of CTL properties can be expressed.

Definition 5.1 *The CTL semantics for a given CTL formula Φ is a ranking 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 $\sigma \models \Phi \iff \sigma \in dom(\tau_\Phi)$ holds.*

The definition of the concerete CTL semantics will be split into path dependent and independent CTL operators.

5.1 Path Independent Operators

We start by defining the *CTL semantics* for atomic propositions and logic operators (see definition 5.2). These CTL properties are path independent and can be defined individually for each state $\sigma \in \Sigma$. The definitions follow directly from the satisfiability relation for CTL properties in section 3.

Definition 5.2 *Equations for path independent CTL operators*

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

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

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

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

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 ranking functions $\langle \Sigma \rightarrow \mathbb{N}, \sqsubseteq \rangle$. Ranking functions are related to each other using the *computational order* $\sqsubseteq$. This partial order relates ranking functions in terms of expressiveness i.e. for how many states can a ranking function prove that the CTL property holds.

Definition 5.3 *Let $f, g \in \Sigma \rightarrow \mathbb{N}$. The computational order $\sqsubseteq$ is defined as follows.*

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

Until

The *CTL semantics* for universal and existential 'until' properties are defined as least 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 totally undefined ranking function $\dot{\emptyset}$ (see definition 5.4). This definition is an generalization of the *guarantee semantics* presented in [1].

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

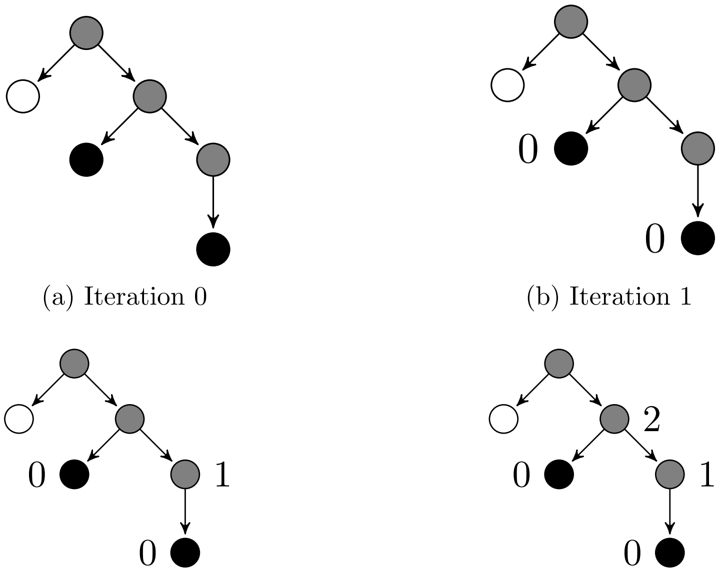
$$\tau_{\forall(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} lfp^{\sqsubseteq}_{\dot{\emptyset}} \ \phi_{\forall(\Phi_1 U \Phi_2)} \tag{7}$$

$$\phi_{\forall(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \textit{if } \sigma \in \textit{dom}(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle \sigma, \sigma' \rangle \in \tau\} & \textit{if } \sigma \notin \textit{dom}(\tau_{\Phi_2}) \ \wedge \\ & \sigma \in \textit{dom}(\tau_{\Phi_1}) \ \wedge \\ & \sigma \in \widetilde{\textit{pre}}(\textit{dom}(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{8}$$

$$\tau_{\exists(\Phi_1 U \Phi_2)} \stackrel{\text{def}}{=} lfp^{\sqsubseteq}_{\dot{\emptyset}} \ \phi_{\exists(\Phi_1 U \Phi_2)} \tag{9}$$

$$\phi_{\exists(\Phi_1 U \Phi_2)} f \stackrel{\text{def}}{=} \lambda\sigma. \begin{cases} 0 & \textit{if } \sigma \in \textit{dom}(\tau_{\Phi_2}) \\ \sup\{f(\sigma') + 1 \mid \langle \sigma, \sigma' \rangle \in \tau\} & \textit{if } \sigma \notin \textit{dom}(\tau_{\Phi_2}) \ \wedge \\ & \sigma \in \textit{dom}(\tau_{\Phi_1}) \ \wedge \\ & \sigma \in \textit{pre}(\textit{dom}(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{10}$$

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. 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



(a) Iteration 0 (b) Iteration 1

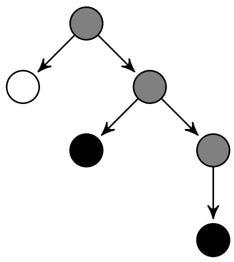
(c) Iteration 2 (d) Iteration 3

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

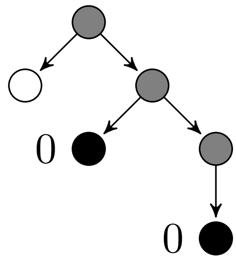
while backtracking, we construct a ranking function sucht that the value assigned to each state is an upper bound on the number of steps until a state satisfying Φ_2 is reached.

The $\widetilde{pre}$ relation guarantees that during the backtracking only those states are considered which exclusively transition to states satisfying $\forall(\Phi_1 U \Phi_2)$. 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).

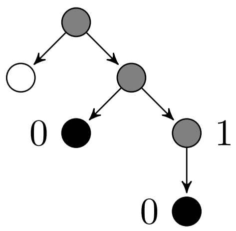
Figures 3 and 4 give an example on how the iterative computation for 'until' properties works for universal and existential quantifiers. Note how figure 4 has one additional iteration because of the existential quantifier. The initial state is added to the ranking function in the last iteration because 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.



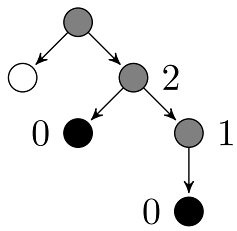
(a) Iteration 0



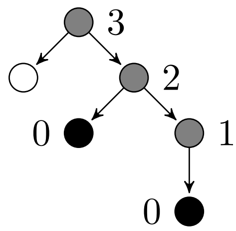
(b) Iteration 1



(c) Iteration 2



(d) Iteration 3



(e) Iteration 4

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

Global

The CTL semantics for universal and existential ‘global’ properties are defined as greates 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 (see definition 5.5). This definition is based on the *recurrence semantics* presented in [1].

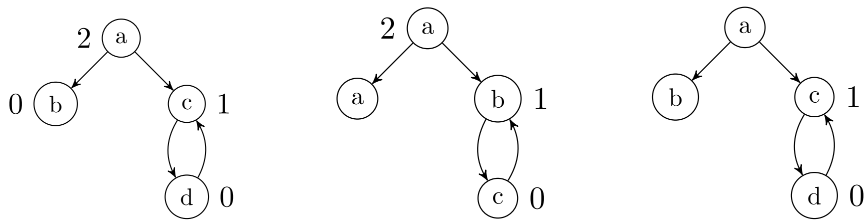
Definition 5.5 *Equations for CTL global operator*

$$\tau_{\forall\Box\Phi} \stackrel{\mathsf{def}}{=} gfp^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\forall\Box\Phi} \tag{11}$$

$$\phi_{\forall\Box\Phi} f \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} f(x) & \textit{if } \sigma \in \widetilde{pre}(dom(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{12}$$

$$\tau_{\exists\Box\Phi} \stackrel{\mathsf{def}}{=} gfp^{\sqsubseteq}_{\tau_\Phi}\ \phi_{\exists\Box\Phi} \tag{13}$$

$$\phi_{\exists\Box\Phi} f \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} f(x) & \textit{if } \sigma \in pre(dom(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{14}$$



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

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

(a) Iteration 0 (b) Iteration 1

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

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 *some* path in case of the existential quantifier ($\exists\Box\Phi$). 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 ranking function τ_Φ of the inner property Φ. In each iteration, every state that is still part of the domain of the ranking function is inspected. The inspected state is kept in the domain of the ranking function if *all* its successor states (or *some* for the existential case) are also part of the domain of the ranking function, otherwise it is removed. That way, only those states are kept in the ranking function which are part of an infinite path consisting exclusively of states satisfying Φ.

Figures 5 and 6 show this for $\tau_{\exists\Box\Phi}$ and $\tau_{\forall\Box\Phi}$. Both iterations start with some initial ranking 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 ranking 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 ranking function. Note that only infinite paths are considered to hold globally.

Next

Definition 5.6 defines the CTL 'next' operator. The 'next' operator is path dependent but does not require fixed-point iterations. A state satisfies $\forall\bigcirc\Phi$ if all its successors satisfy the property Φ, correspondingly $\exists\bigcirc\Phi$ is satisfied if at least one successor satisfies the property Φ. This corresonds to the definition of the $\widetilde{pre}$ and pre relations. Zero is assigned to each state that satisfies the property to construct a valid ranking function.

Definition 5.6 *Equations for CTL next operator*

$$\tau_{\forall\bigcirc\Phi} \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} 0 & \textit{if } \sigma \in \widetilde{pre}(dom(\tau_\Phi)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{15}$$

$$\tau_{\exists\bigcirc\Phi} \stackrel{\mathsf{def}}{=} \lambda\sigma. \begin{cases} 0 & \textit{if } \sigma \in pre(dom(\tau_\Phi)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{16}$$

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 $\mathcal{X}$.*

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

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

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 modeled as control-flow-graphs (CFG) (see definition 6.2). 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 $(u, s, v) \in edge$ 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 $(\text{cfg}, l_{entry}, l_{exit}) \in prog$ consists of a control-flow-graph and two special nodes that defined the entry and exit point of the program.

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

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

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

We introduce the following auxiliary functions on nodes of a control-flow-graph to refere 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. Decision trees encode piecewise-defined ranking functions which are used as an abstraction the of general ranking functions (see section 4). First we give a description

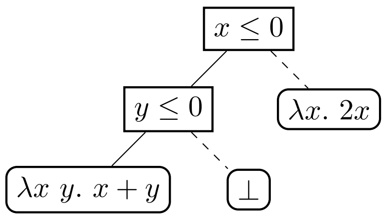


Figure 7: Example for decision tree

of the decision tree abstrac 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 constrains and the leafs are linear functions. Decision trees partition a state space, given by a set of variables $\mathcal{X}$, into linear 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 ranking 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 $x \le 0$ and $y \le 0$. The left most leaf is the function $\lambda x\ y.\ x + y$. It is defined for all states satisfying $x \le 0 \land y \le 0$ 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 $\neg(x \le 0)$ (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. Combining all constraints and functions of the decision tree in figure 7 yields the following partial function:

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

Now we formalize the domain using mathematical notations.

Constraints

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

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ c_1X_1 + \cdots + c_nX_n + c_{n+1} \geq 0 \;\middle|\; \begin{array}{l} \mathcal{X} = \{X_1, \ldots, X_n\} \\ c_1, \ldots, c_n, c_{n+1} \in \mathbb{Z} \\ gcd(|c_1|, \ldots, |c_n|, |c_{n+1}|) = 1 \end{array} \right\}$$

Elements of $\mathcal{C}$ can be instances of the *interval abstract domain*, the *octagon abstract domain* or the *polyhedra abstract domain* (TODO cite).

Functions

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

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

Functions that return a constant value $n \in \mathbb{N}$ are written 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 ranking function that the decision tree represents is defined for that partition.

Decision Trees

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

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

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

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

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

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

Orders & Abstractions

Decision trees are used as an abstraction of ranking functions. It is important to be able to reason about the soundness of this abstraction. For that purpose we will now introduce a number of partial orders which are then used to express soundness.
For the elements of $\mathcal{F}$ we defined the *computational order* $\sqsubseteq_{\mathcal{F}}$ and *approximation order* $\preceq_{\mathcal{F}}$. Both orders are defined as follows for defined leafs.

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

Undefined leafs are ordered according to the Hasse diagrams in figure 8.

Now we lift these orders to decision trees. The *computational order* $\sqsubseteq_{\mathcal{T}}$ and *approximation order* $\preceq_{\mathcal{T}}$ are defined by leaf-wise comparison of two decision trees with the corresponding orders $\sqsubseteq_{\mathcal{F}}$ and $\preceq_{\mathcal{F}}$. A more detailed description can be found in [6]. Intuitively, the *computational order* $\sqsubseteq_{\mathcal{T}}$ is an approximation of the *computational order* $\sqsubseteq$ defined in section 5 (definition 5.3).

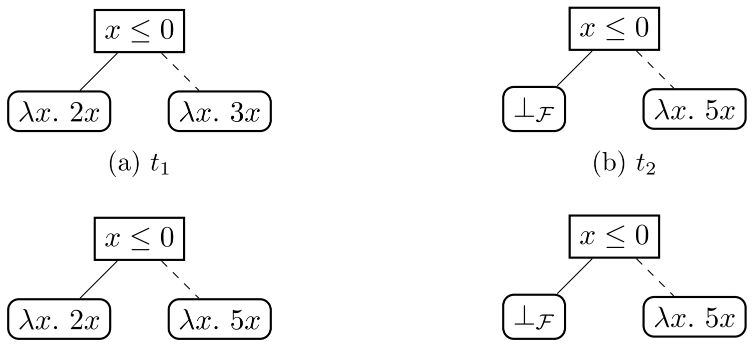
The *approximation order* $\preceq_{\mathcal{T}}$ is used to define sound abstractions.

Definition 7.1 *Let $\alpha \in (\Sigma \to \mathbb{N}) \to \mathcal{T}$ denote the abstraction function from ranking functions to decision trees. A decision tree $t \in \mathcal{T}$ is a sound abstraction of a ranking function $f \in \Sigma \to \mathbb{N}$ if $\alpha(f) \preceq_{\mathcal{T}} t$.*

This definition allows us to use $\top_{\mathcal{F}}$ leafs in decision trees to represent uncertainty about some values of a ranking function. Due to definition 7.1 we can use such a decision tree to soundly abstract a more precise ranking function.

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



(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

both trees consist of the same partitions. They only differ in the valus of the leafs. Then the two trees can be joined leaf-wise. There are two join variations. The *computational join* $\sqcup_{\mathcal{T}} \colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \rightarrow \mathcal{T}_{NIL}$ and the *approximation join* $\curlyvee_{\mathcal{T}} \colon (\mathcal{T}_{NIL} \times \mathcal{T}_{NIL}) \rightarrow \mathcal{T}_{NIL}$. The first one joins leafs (by forming the smallest upper bound) with $\sqsubseteq_{\mathcal{F}}$ the latter with $\preceq_{\mathcal{F}}$. 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. For detailed description of the two join types see [6].

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* $\texttt{MEET}_{\sqsubseteq} \colon (\mathcal{T}_{\texttt{NIL}} \times \mathcal{T}_{\texttt{NIL}}) \rightarrow \mathcal{T}_{\texttt{NIL}}$ and the *approximation meet* $\texttt{MEET}_{\preceq} \colon (\mathcal{T}_{\texttt{NIL}} \times \mathcal{T}_{\texttt{NIL}}) \rightarrow \mathcal{T}_{\texttt{NIL}}$. Both combine defined leafs using the smallest upper bound according to $\preceq_{\mathcal{F}}$. If at least one of the two leafs is `NIL`, then the result is $\perp_{\mathcal{F}}$ in case of the *computational meet* and `NIL` in case of the *approximation meet*.

7.4 Filter

The filter operator $\texttt{FILTER}[\![bexp]\!] \colon \mathcal{T}_{\texttt{NIL}} \rightarrow \mathcal{T}_{\texttt{NIL}}$ prunes all partitions of a decision tree that do not satisfy a given boolean expression. Leafs are pruned by replacing them with `NIL`. The regular version of `FILTER` prunes the partitions using overapproximation. That means that the resulting tree can still contain states that do not satisfy the boolean expression. It is however guaranteed that all states that satisfy the boolean expression remain in the tree. In addition to the overapproximating version there is also the underapproximating version $\texttt{FILTER\_UNDER}[\![bexp]\!]: \mathcal{T}_{\texttt{NIL}} \to \mathcal{T}_{\texttt{NIL}}$. Here the resulting decision tree is guaranteed to not contain any partitions that do not satisfy the boolean expressions. States that do satisfy it might be removed however if the underlying numerical domain is not expressive enough.

7.5 Reset

The $\texttt{RESET}[\![bexp]\!]: \mathcal{T}_{\texttt{NIL}} \to \mathcal{T}_{\texttt{NIL}}$ operator can be used to reset the value of the ranking function to zero for all partitons that satisfy the boolean expression. This operator has the same implementation as $\texttt{FILTER\_UNDER}[\![bexp]\!]$ but replaces leafs that do not satisfy the property with the constant function 0. Furthermore it uses undreapproximation to guarantee that only those partitions are reset that actually satisfy the boolean expression.

7.6 Backward Assign

The operator $\texttt{B\_ASSIGN}[\![X: \ = \texttt{aexp}]\!]: \mathcal{T}_{\texttt{NIL}} \to \mathcal{T}_{\texttt{NIL}}$ handles the backward assignment of the arithmetic expression `aexp` to variable X. The linear constraints of the decission tree nodes and the functions at the leafs are adjusted accordingly. `B_ASSIGN` uses overapproximation on the underlying numerical domains. As with the `FILTER` operator, there also exists an underapproximating version $\texttt{B\_ASSIGN\_UNDER}[\![X: \ = \texttt{aexp}]\!]: \mathcal{T}_{\texttt{NIL}} \to \mathcal{T}_{\texttt{NIL}}$.

8 Abstract Semantics for CTL

In this section we present a sound and computable approximation of the *CTL semantics* τ_Φ defined in section 5. Computing the ranking function τ_Φ is in general not computable as one can easily encode the halting problem. Therefore, we approximation τ_Φ by using the decision tree abstract domain (section 7) to approximate ranking functions in terms of piecewise defined ranking functions.

Definition 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 $\preceq$.*

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 partitioned by decision trees $t \in \mathcal{T}$ and grouped by program labels $l \in \mathcal{L}$. 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 i.e. all leafs of the decision tree are defined.

The following sections present how to compute $\tau^\sharp_\Phi$ 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 $\tau^\sharp_a$ 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 RESET$[\![a]\!]$ operator on the totally undefined decision tree $\bot_\mathcal{T}$.

Definition 8.2 *Abstract CTL semantics for atomic propositions*

$$\tau^\sharp_a \stackrel{\text{def}}{=} \lambda l.\ \mathit{RESET}[\![a]\!]\bot_\mathcal{T} \tag{17}$$

Fixing the RESET operator

The RESET$[\![a]\!]$ operator was originally introduced in [6] in the context of abstract guarantee semantics and would overapproximate 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 common numerical domains. An overapproximating implementation of 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 an underapproximating implementation of RESET as presented in section 7.

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

Definition 8.3 *Abstract CTL semantics for logic operators*

$$\tau^{\sharp}_{\Phi_1 \wedge \Phi_2} \stackrel{\text{def}}{=} \lambda l.\ (\tau^{\sharp}_{\neg\Phi_1} l)\ \sqcup_{\mathcal{T}}\ (\tau^{\sharp}_{\neg\Phi_2} l) \tag{18}$$

$$\tau^{\sharp}_{\Phi_1 \vee \Phi_2} \stackrel{\text{def}}{=} \lambda l.\ (\tau^{\sharp}_{\neg\Phi_1} l)\ \sqcap_{\mathcal{T}}\ (\tau^{\sharp}_{\neg\Phi_2} l) \tag{19}$$

$$\tau^{\sharp}_{\neg\Phi} \stackrel{\text{def}}{=} \lambda l.\ \mathit{COMPLEMENT}\ (\tau^{\sharp}_{\neg\Phi} l) \tag{20}$$

The abstract CTL semantics for $\wedge$ and $\vee$ (equations 18 and 19) 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 smallest upper bound of the assigned value is formed to approximate the concrete CTL semantics.

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). As with the logical $\vee$, the smallest upper bound is formed if a partition is defined in both trees. By using the computational version of the operator, we ensure that no `NIL` leafs are introduced when forming the intersection.

For the logical $\neg$ operator we introduce the `COMPLEMENT`: $\mathcal{T}_{\texttt{NIL}} \to \mathcal{T}_{\texttt{NIL}}$ operator. This operator replaces all defined leafs with a $\bot$-leaf and all $\bot$-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.

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 overapproximation on the underlying numerical domains, the second one underapproximation.

Algorithm 1 Tree Complement

function COMPLEMENT(t) ▷ $t \in \mathcal{T}_{NIL}$
  if $(isNode(t) \wedge t.f = \top) \vee isNil(t)$ **then**
    return t ▷ ignore $\top$ and NIL
  else if $isLeaf(t) \wedge t.f = \bot$ **then**
    return $LEAF : 0$ ▷ undefined becomes defined
  else if $isLeaf(t)$ **then**
    return $LEAF : \bot$ ▷ defined becomes undefined
  else
    $l \leftarrow$ COMPLEMENT($t.l$)
    $r \leftarrow$ COMPLEMENT($t.r$)
    return $NODE\{t.c\} : l; r$
  end if
end function

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 computed a decision tree for the target node of some edge. This decision tree represents the value of the ranking 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 ranking function at the source node of this edge.

Definition 8.4 *Abstract semantics for basic statements*

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

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

The `skip` statement is handled by the STEP operator. This operator increases the value of all defined partitions in the decision tree by one (see XY TODO). 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 B-ASSIGN and FILTER operators that were introduced in section 7. The definitons 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

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

starting from the totally undefind decision tree $\bot_{\mathcal{T}}$ (see definition 8.5). We will first discuss the universal version and then explain what changes for the existential case.

Definition 8.5 *Abstract semantics for 'until' operator.*

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

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

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 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 applied 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 $\forall$.

We use the overapproximating version of the $[\![\cdot]\!]_o$ function. This might temporarely lead to unsound decision trees due to overapproximation. 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 those partitions which satisfy $\forall(\Phi_1 U \Phi_2)$.

The abstract tansformer 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 underapproximating $[\![stmt]\!]_u$ function when processing satements to guarantee soundness.

Global

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

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

starting from the abstract CTL semantics $\tau^\sharp_\Phi$ of the inner CTL property Φ (see definition 8.6).

Definition 8.6 *Abstract semantics for 'global' operator.*

$$\tau^\sharp_{\forall\Box\Phi} \stackrel{\text{def}}{=} gfp^{\sqsubseteq_\mathcal{T}}_{\tau^\sharp_\Phi}\, \phi^\sharp_{\forall\Box\Phi}$$

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

$$\phi^\sharp_{\forall\Box\Phi}(m)l \stackrel{\text{def}}{=} \texttt{MASK}[\![t_\curlyvee(l)]\!](m(l))$$

$$\tau^\sharp_{\exists\Box\Phi} \stackrel{\text{def}}{=} gfp^{\sqsubseteq_\mathcal{T}}_{\tau^\sharp_\Phi}\, \phi^\sharp_{\exists\Box\Phi}$$

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

$$\phi^\sharp_{\exists\Box\Phi}(m)l \stackrel{\text{def}}{=} \texttt{MASK}[\![t_\sqcup(l)]\!](m(l))$$

The abstract transformer for the 'global' operator uses the same approach as the 'until' operator to join outgoing edges w.r.t $\forall$ and $\exists$. In the final step however, the current decision tree $m(l)$ is masked with the updated decision tree $t_\curlyvee(l)$ and $t_\sqcup(l)$. Masking means the all defined partitions in $m(l)$ that are not also defined in $t_\curlyvee(l)$ (or $t_\sqcup(l)$) are discarded. Note that the decision tree $\tau^\sharp_\Phi(l_{exit})$ is the totally undefind decision tree $\bot_\mathcal{T}$. That way all states that do not satisfy Φ indefinitely among all (or some) paths are iteratively removed from the decision tree until a fixed-point is reached. The `MASK` operator is defined in algorithm 4.

Next

The abstract CTL semantics for the 'next' operator are given in definition 8.7.

Definition 8.7 *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 applied to the `ZERO` operator which sets all defined partitions to zero (see algorithm 5).

Algorithm 2 Tree Until Filter

function FILTER_UNTIL(t, t_{valid})
  if $isNil(t) \vee isNil(t_{\text{valid}})$ **then**
    ▷ ignore NIL nodes
    return t
  else if $isLeaf(t) \wedge isLeaf(t_{\text{valid}}) \wedge isDefined(t)$ **then**
    ▷ t is defined in t_{valid}
    return t
  else if $isLeaf(t) \wedge isLeaf(t_{\text{valid}}) \wedge \neg isDefined(t)$ **then**
    ▷ t is not defined in t_{valid}, make undefined
    return $LEAF : \bot$
  else
    $l \leftarrow$ FILTER_UNTIL($t.l, t_{\text{valid}}.l$)
    $r \leftarrow$ FILTER_UNTIL($t.r, t_{\text{valid}}.r$)
    return $NODE\{t.c\} : l; r$
  end if
end function

Algorithm 3 Tree Until

function RESET_UNTIL(t, t_{reset})
    if $isNil(t) \lor isNil(t_{\text{reset}})$ **then**
        ▷ ignore NIL nodes
        return t
    else if $isLeaf(t) \land isLeaf(t_{\text{reset}}) \land isDefined(t)$ **then**
        ▷ t is defined in t_{valid}, reset leaf
        return $LEAF : 0$
    else if $isLeaf(t) \land isLeaf(t_{\text{valid}}) \land \neg isDefined(t)$ **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

Algorithm 4 Tree Mask

function MASK$[\![t_{\text{MASK}}]\!](t)$
  function MASK_AUX(t, t_{mask})
    if $isNil(t) \vee isNil(t_{\text{reset}})$ **then**
      $\triangleright$ 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**
        $\triangleright$ t is defined and t_{mask} is undefined, discard leaf
        return $LEAF : \perp$
      else
        return t
      end if
    else
      $l \leftarrow$ MASK$(t.l, t_{\text{mask}}.l)$
      $r \leftarrow$ MASK$(t.r, t_{\text{mask}}.r)$
      return $NODE\{t.c\} : l; r$
    end if
  end function
  $(t_1, t_2) \leftarrow$ TREE_UNIFICATION(t, t_{MASK})
  return MASK_AUX(t_1, t_2)
end function

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

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*$^{\text{th}}$ *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.