Specification models and their analysis
Lecture 2: Computation Tree Logic

Kai Lampka

December 15, 2010

Institut für Technische Informatik und Kommunikationsnetze

Preliminary Remark

- In the following we will develop a mechanical view of propositional logic allowing to mechanical interpret formulae of propositional logic.
- This might appear long-winded. But, remember what we need is a formalized apparatus for mechanically (= done by a machine) interfere validity of statements about a computer system.
- This elegant style will furthermore allow us to define how to interpret CTL-formulae over Kripke structures.

Part I

Preliminaries: Propositional Logic

- Let $\mathcal{AP}$ be a set of atomic propositions, e. g. the sun shines, it is raining, etc. ..
- The atomic propositions are **either** wrong **or** false, unlike the variables employed in logic of circuits (Schaltungslogik) which can be 0 or 1.
- Let $\mathfrak{true}$ be a constant, 0-ary operator denoted as true.
- Let $\wedge$ be a binary operator denoted as conjunction.
- Let $\vee$ be a binary operator denoted as disjunction.
- Let $\neg$ be the unary operator denoted negation.
- Syntactically correct formulae of propositional logic are defined by the following grammar:

$$A ::= \mathfrak{true} \mid \alpha \in \mathcal{AP} \mid \neg A \mid A \wedge A \mid (A) \mid A \vee A$$

The grey colored parts are not essential, we use them here for convenience.

Propositional Logic

- Let $\mu : \mathcal{AP} \cup \{\mathbf{true}\} \rightarrow \{1, 0\}$ be an assignment for the atomic proposition, s.t. $\forall \alpha \in \mathcal{AP}$: $\mu(\alpha) = 1$ or $\mu(\alpha) = 0$ holds. The terminal symbol **true** is always mapped to 1.
- Let $\mathcal{F}$ be the set of all formulae which can be constructed according to the above production rule and which solely uses the terminal symbols of $\mathcal{AP}$.
- Let $\tilde{\mu} : \mathcal{F} \rightarrow \{1, 0\}$ be now an evaluation of each formulae contained in $\mathcal{F}$, where we say that $C \in \mathcal{F}$ is true if $\tilde{\mu}(C) = 1$ otherwise we say C is false. We use here 0 and 1 instead of *true* and *false* for clearly separating the syntactical level from the level of interpretation.

Given the evaluation of atomic proposition and complex formulae over $\mathcal{AP}$ we can now define how $\tilde{\mu}$ operates, which fixes the way how formulae of the above kind have to be interpreted. For convenience we do this by defining a binary relation $\models \subset \mu \times \mathcal{F}$ denoted as **satisfaction** relation.

- Let $\models \subseteq \mu \times \mathcal{F}$ be the binary relation where for any formula C and a matching assignment μ holds that

$$(\mu, C) \in \models \text{ if and only if } \tilde{\mu}(C) = 1$$

- The elements of $\models$ are those formulae which posses a fulfilling assignment μ to the atomic propositions of the formula C.

Simplification:

- Each time $\tilde{\mu}$ is applied to an atomic proposition $\alpha \in \mathcal{AP}$ it returns the value μ defined for α, i. e., 1 or 0.
- Thus assignment μ and evaluation $\tilde{\mu}$ agree on the elements of $\mathcal{AP}$. This allows us to drop the distinction between atomic proposition and formulae and solely employ evaluation μ.

For interpreting formulae of propositional logic we need now to make the following distinction w. r. t. satisfaction relation $\models$.

1. **Interpretation of the terminal symbolic $\mathfrak{true}$**: $(\mu, \mathfrak{true}) \in \models$ per definition. The constant true-function evaluates to $\mathfrak{true}$ for every assignment ("true is true, no matter what").
2. **Interpretation of atomic propositions**:
   $\alpha \in \mathcal{AP}$: $(\mu, \alpha) \in \models$ if and only if $\mu : \alpha \mapsto 1$ holds.
3. **Interpretation of negation**:
   $(\mu, \neg A) \in \models$ if and only if $(\mu, A) \notin \models$ holds.
4. **Interpretation of conjunction**:
   $(\mu, A \wedge B) \in \models$ if and only if $(\mu, A) \in \models$ and $(\mu, B) \in \models$ holds.
5. **Interpretation of disjunction**:
   $(\mu, A \vee B) \in \models$ if and only if one of the following holds:
   (a) $(\mu, A) \in \models$ and $(\mu, B) \notin \models$; (b) $(\mu, A) \notin \models$, and $(\mu, B) \in \models$;
   (c) $(\mu, A) \in \models$ and $(\mu, B) \in \models$.

- The above rules can be recursively applied until one reaches base case (1) or (2), hence the satisfaction relation defines how formulae of propositional logic can be interpreted, we say it defines a semantic of propositional logic.
- In case $(\mu, C) \in \models$ one also says that μ is a model for C. In case there is no model satisfying $\neg C$ formula C is called a **tautology** and in case C has at least one valid model it is called **satisfiable**.
- In the following we write $\mu \models C$ instead of $(\mu, C) \in \models$ and we write $\mu \not\models C$ instead of $(\mu, C) \notin \models$.

- In the following we use logic implication. This operator is defined as follows:

$$\mu \models (A \Rightarrow B) \text{ if and only if } \mu \models \neg A \vee B \text{ holds.}$$

Operator $\vee$ is defined as before. An implication which is based on a false formulae (or statement), here A is always true; "from rubbish you can interfere everything".

- This allows us now to define an operator for the formulation **if and only if**, where one commonly uses the symbol $\Leftrightarrow$, respectively *iff*:

$$\mu \models A \Leftrightarrow B \text{ if } \mu \models (A \Rightarrow B) \wedge (B \Rightarrow A) \text{ holds.}$$

How to evaluate complex formulae of propositional logic

A parse tree is an (ordered, rooted) tree that represents the syntactic structure of a string according to some formal grammar. In a parse tree, the interior nodes are labelled by non-terminals of the grammar, while the leaf nodes are label-ed by terminals of the grammar.

1. Take formula C and convert it s.t. it only contains the basic operators $\neg$, $\wedge$ or $\vee$.
2. Generate a parse tree for the formula s.t. the leaves of the parse tree carry the atomic propositions or the constant *true* and each inner node carries an operator of $\{\neg, \wedge, \vee\}$;
3. evaluate the atomic propositions at the leave nodes by defining $\mu : \mathcal{AP} \to \{1, 0\}$; $\mu(\mathit{true}) = 1$ is determined.
4. process the parse tree bottom-up, and label each inner node with the respective 0 or 1 value.
5. When arriving at the root node the interpretation of C w. r. t. model μ has been achieved.

Decidability

- A true/false-question is decidable *iff* there is an algorithm which after **finitely** many steps returns with either true or false.
- If a true/false-question is semi-decidable the computation **may** not be finished after **finitely** many steps (= the algorithm runs forever).
- In fact, propositional logic is decidable, as all formulae over finitely many variables can be evaluated to 1 or 0. Does this really hold for formulae of infinite length?
- However, not all theories are decidable, i. e., not for all theories it is possible to mechanically carry out proofs.

Decidability

Most celebrated result in theoretical computer science are Goedel's incompleteness theorems (from wikipedia):

1. The first incompleteness theorem states that no consistent system of axioms whose theorems can be listed by an "effective procedure" (essentially, a computer program) is capable of proving all facts about the natural numbers. For any such system, there will always be statements about the natural numbers that are true, but that are unprovable within the system.

2. The second incompleteness theorem shows that if such a system is also capable of proving certain basic facts about the natural numbers, then one particular arithmetic truth the system cannot prove is the consistency of the system itself.

- Semi-decidability of first-order logics for formulating statements about natural numbers (= propositional logic enriched with predicates, functions, and exists and all quantifiers). This result implies that there are no fully automatic proof producing machines for theories of a certain complexity. In fact Goedel showed that the theory of arithmetic formulae is already undecidable.

  Intuition Consider the following formula:

$$\begin{aligned} F := \quad & \forall x \in \mathbb{D} : x < x + 1 \wedge \forall y \in \mathbb{D} : \neg(y < y) \\ & \wedge(\forall u \forall v \forall w) \in \mathbb{D} : ((u < v) \wedge (v < w)) \Rightarrow u < w \end{aligned}$$

  This formula is true only for infinite domains $\mathbb{D}$ such as $\mathbb{N}$, thus it is impossible to (mechanically) construct a model in finitely many steps s.t. the above formula is correct.

- Besides propositional logic, there are other decidable theories, e. g. Pressburger arithmetics (= 1'st-order theory on $\mathbb{N}$ with $+$ and $=$ and induction as proof scheme). But, Pressburger arithmetics already extended with multiplication (= Peano arithmetics), is undecidable, which is the system Goedel originally dealt with.

Part II

Introduction

- In (formal) logic one studies how to combine propositional formulae consisting of atomic propositions, manipulate the formulae, and ultimately draw correct conclusions, i. e., decide if a (complex) formula (= combination of statements) is correct or not.
- This requires a decidable theory and a set of "mechanical" methods for showing that a complex statement about a system model is correct. Actually we show that a system model, here a Kripke structure, is a valid model of a complex statement, here a CTL-formulae.

We extend the notion of Labelled Transition Systems as follows:

Definition 2.1: Kripke structure

A Kripke structure $\mathcal{K}$ is a six-tuple $\mathcal{K} := (\mathbb{S}, \mathbb{S}_0, \mathcal{A}ct, \mathbb{E}, \mathcal{AP}, \mathcal{L})$, where

1. $\mathbb{S} := \{\vec{s}_1, \ldots, \vec{s}_n\}$ is an ordered (indexed) set of states with
2. $\mathbb{S}_0$ is the set of initial states.
3. $\mathcal{A}ct$ is the discrete set of transition labels,
4. $\mathbb{E} \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$ is an ordered (indexed) set of labelled state-to-state transitions.
5. $\mathcal{AP}$ is a set of atomic propositions, e. g. $\{green, blue, yellow, black\}$ and
6. $\mathcal{L} : \mathbb{S} \mapsto 2^{\mathcal{AP}}$ as state labelling function.

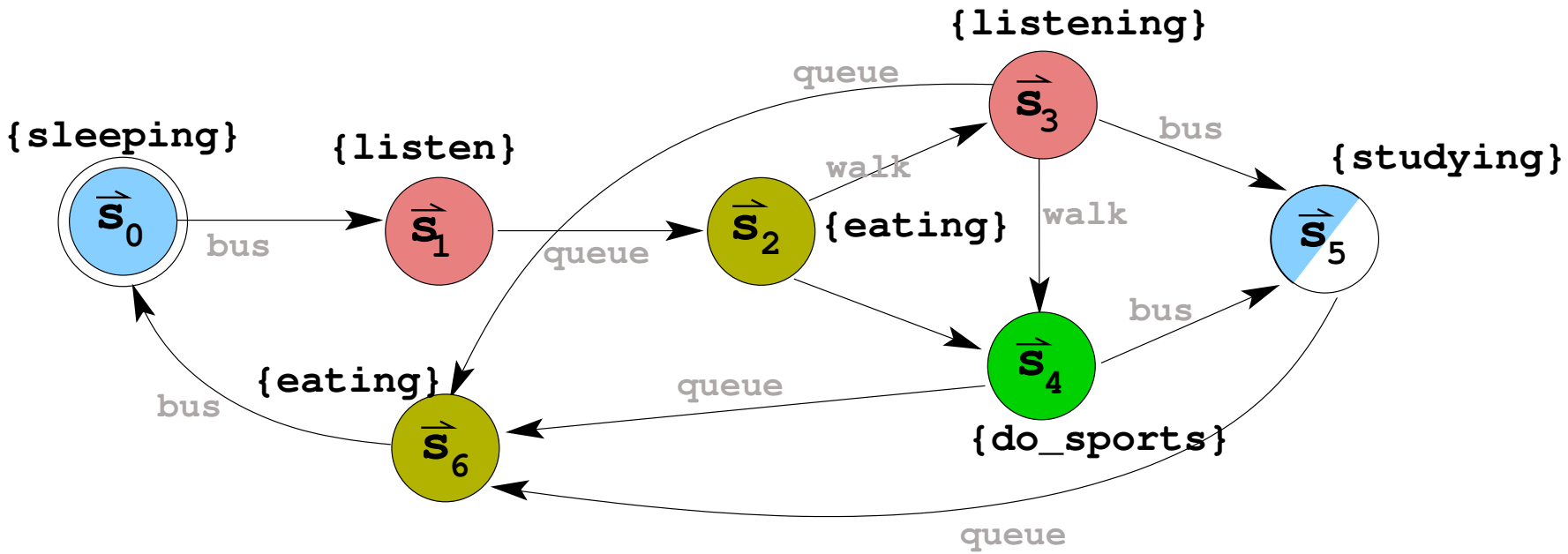
Introduction to CTL Model checking

- Analogously to propositional logic one wants to reveal if a formal statement about a system's behavior is correct or not.
- Whereas in propositional logics this is easy, –one simply needs to evaluate a formulae w. r. t. an assignment μ–, the reasoning about Kripke structures is much more demanding.
- However, at first we need to clarify how a Kripke structure defines a system behavior.

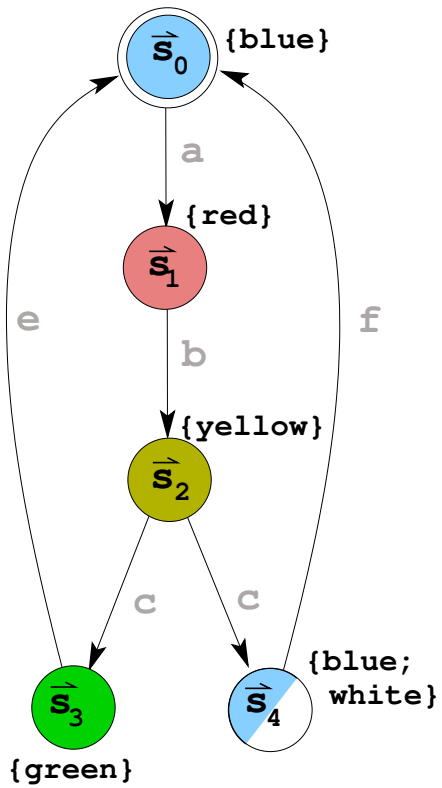
Introduction to CTL Model checking

- Analogously to propositional logic one wants to reveal if a formal statement about a system's behavior is correct or not.
- Whereas in propositional logics this is easy, –one simply needs to evaluate a formulae w. r. t. an assignment μ–, the reasoning about Kripke structures is much more demanding.
- However, at first we need to clarify how a Kripke structure defines a system behavior.

Introduction to CTL Model checking



Exists a schedule (= execution), s.t. a student will never study?



- System is given as Kripke structure, hence future behavior is defined by sequences of states denoted as **path**.
- for any pair of states within a path we require that there is a connecting edge within the Kripke structure:

$$\pi_{\vec{s}_0} := \vec{s}_0, \vec{s}_1, \vec{s}_2, \vec{s}_4, \vec{s}_0, \vec{s}_1 \text{ (finite path fragment)}$$

- in fact we are interested in the sequence of atomic propositions attached to each state ($\mathcal{L}(\vec{s}_i)$), but for simplicity we stick to the state identifiers $\vec{s}_i$

As we see from this:

- temporal logics which are the logics on transition systems are *time abstract*, i. e., they allow to reason about *ordering* of states. They *do not* allow to reason about state residence times!
- The modelling and reasoning about real-time systems is denoted timed verification.
- Hence one reasons over system behaviors which are defined by paths in the Kripke structure.
- This allows one to make statements over a single path (= linear time view), or over sets of paths (= branching time view).
- CTL follows the branching time view, hence it allows to make statements about set of paths, like $\exists$ a path s.t. , $\forall$ paths it holds: ...

Introduction to CTL Model Checking: Branching time view

- To reason about the properties of a system (model) in a branching time view one must expand all paths (= all possible behaviors), starting at the initial state.
- For simplicity we are considering in the following only Kripke structures
  - with a single initial state ($\vec{s}_0$), s.t. we only need to worry about paths starting in state $\vec{s}_0$.
  - which are non-terminal (i. e., there are no deadlocks),

$\longrightarrow$ Question 2.1: What do we get if we unroll all paths of a Kripke structure, transition by transition starting at the initial state

The computation tree (CT) of a Kripke structure $\mathcal{K} := (\mathbb{S}, \mathbb{S}_0, \mathcal{A}ct, \mathbb{E}, \mathcal{AP}, \mathcal{L})$ can be constructed as follows:

- each node of the CT carries a state label contained in $\mathbb{S}$;
- the root of the CT is labelled with the state label $\vec{s}_0$;
- each child of a CT-node c is labelled with a state-label $\vec{s}$
- each children node is a successor of its parent node w. r. t. the associated state descriptors and the transition relation of $\mathcal{K}$.
- For the set of children nodes of a CT-node c we have:

$$\texttt{child}(c) := \bigcup_{\forall l \in \mathcal{A}ct : (\vec{s}, l, \vec{t}) \in \mathbb{E}} \vec{t}$$

- Since each node of the CT carries a state label $\vec{s}$, it can be annotated with the set of atomic propositions which are actually fulfilled by the resp. state $\vec{s}$, i. e., with $\mathcal{L}(\vec{s})$.

Part III

Model Checking with CTL

CTL Model Checking: Defining CTL

CTL has the following ingredients:

1. atomic propositions, where a state $\vec{s}$ satisfies a atomic proposition $a \in \mathcal{AP}$ if it carries the respective label ($\mathcal{L}(\vec{s}) = a$)

2. standard logic operators $\wedge, \neg$ and their derivatives, e. g. $\Rightarrow$, which allow to construct more complex state formulae;

   $\longrightarrow$ Example 3.1: $a \Rightarrow \neg(c \vee b)$

3. quantifiers $\exists$ and $\forall$ applied to path formulae, i. e., sequences of state properties to be fulfilled w. r. t. some starting state $\vec{s}_0$.

   $\longrightarrow$ Example 3.2: $\exists\Psi, \forall\Psi$

4. temporal operators $\bigcirc$ (= next) and U (= until) which we apply to state formulae and which gives us path formulae;

   $\longrightarrow$ Example 3.3: $\Psi := \bigcirc b$ $\Psi' := a \,\mathsf{U}\, b$

Definition 3.1: Computation Tree Logic

- CTL formula consists of sub-formulae which are either **path** formulae (Ψ) or **state** formulae (ϕ). With $a \in \mathcal{AP}$ as set of atomic propositions we give the following definitions:
  - A CTL **state formula** ϕ is defined as

$$\phi := true \mid a \in \mathcal{AP} \mid \phi \wedge \phi \mid \neg\phi \mid \exists\Psi \mid \forall\Psi$$

    with Ψ as CTL path formula.
  - A CTL **path formula** Ψ is defined as

$$\Psi := \bigcirc \phi \mid \phi \,\mathsf{U}\, \phi'$$

    where the ϕ's are CTL **state formulae**.

Consider the following CTL formulae with $\{coin, wash\} =: \mathcal{AP}$

- $\exists \bigcirc coin$
- $\forall(true \,\mathsf{U}\, wash)$
- $\exists(coin \wedge \forall \bigcirc wash)$
- $\exists \bigcirc (coin \wedge \forall \bigcirc wash)$

1. Which of the above formulae are syntactically correct?
2. How does a non-trivial fulfilling CT could look like?

As for propositional logics we define a **satisfaction relation** $\models$ for CTL-formulae:

Definition 3.2: Semantics of CTL

1. For a Kripke structure $\mathcal{K}$ and a state $\vec{s}$ we define the following:
   - $\vec{s} \models a \Leftrightarrow a \in \mathcal{L}(\vec{s})$
   - $\vec{s} \models \neg\phi \Leftrightarrow \vec{s} \not\models \phi$
   - $\vec{s} \models \phi \wedge \phi' \Leftrightarrow \vec{s} \models \phi \wedge \vec{s} \models \phi'$
   - $\vec{s} \models \exists\Psi \Leftrightarrow \pi_{\vec{s}} \models \Psi$ for **some** path $\pi_{\vec{s}}$ in $\mathcal{K}$
   - $\vec{s} \models \forall\Psi \Leftrightarrow \pi_{\vec{s}} \models \Psi$ for **all** paths $\pi_{\vec{s}}$ in $\mathcal{K}$
2. For a path $\pi_{\vec{s}}$ in $\mathcal{K}$ we define:
   - $\pi_{\vec{s}} \models \bigcirc\phi \Leftrightarrow \pi_{\vec{s}}[1] \models \phi$
   - $\pi_{\vec{s}} \models \phi \,\mathsf{U}\, \phi'$
     $\Leftrightarrow \exists j \geq 0 : \pi_{\vec{s}}[j] \models \phi' \wedge \forall(k : 0 \leq k < j) : \pi_{\vec{s}}[k] \models \phi,$
     where $\pi_{\vec{s}}[x]$ refers to the x'th state of path $\pi_{\vec{s}}$.

- However complex CTL-formulae might also contain non-standard operators, e. g. $a \Rightarrow \neg(c \vee b)$.
- For reducing the number of cases to be covered ($true, a \in \mathcal{AP}, \wedge, \neg, \forall\bigcirc, \exists\bigcirc, \forall\,\mathsf{U}, \exists\,\mathsf{U}$), as well as for simplifying their treatment each CTL-formula is converted into a **normal form**
- In the following we will make use of the so called existential normal form (ENF) which solely employs the operators $\neg, \wedge, \exists\bigcirc, \exists\,\mathsf{U}$ and $\exists\Box$ where $\Box$ is the always operator.

Definition 3.3: The always operator ($\Box$)

- potentially always: $\exists\Box\phi := \neg\forall(true\,\mathsf{U}\,\neg\phi)$
  there is (at least one) path π s.t. ϕ holds in each state of π.
- invariant: $\forall\Box\phi := \neg\exists(true\,\mathsf{U}\,\neg\phi)$
  for all paths Π and hence all states ϕ holds

⟶ Example 3.4: Example for the $\Box$-operator

Definition 3.4: Existential normal form

A CTL-formula is in existential normal form (ENF) if it is of the following type:

$$\phi := \mathit{true} \mid a \in \mathcal{AP} \mid \phi \wedge \phi \mid \neg\phi \mid \exists \bigcirc \phi \mid \exists(\phi \,\mathsf{U}\, \phi) \mid \exists\Box\phi$$

For converting a CTL formula in ENF one needs to replace the universal by the existential quantifier. This is possible by exploiting the following dualities:

- $\forall \bigcirc \phi = \neg\exists \bigcirc \neg\phi$
- $\forall(\phi' \,\mathsf{U}\, \phi'') = \neg\exists\,[\neg\phi'' \,\mathsf{U}\, (\neg\phi' \wedge \neg\phi'')] \wedge \neg\exists\Box\neg\phi''$

Thus for deciding if a system $\mathcal{L}$ complies with a property a resp. model checking algorithm must only cover the above 7 ENF-base cases.

- For actually model checking a LTS $\mathcal{L}$ we need to extend the above defined satisfaction relation to transition systems (we also do not want to expand the CT explicitly).
- Let Ω be a CTL-formula and let $\mathcal{L}$ be a finite non-terminal LTS

$$\mathcal{L} \models \Omega \Leftrightarrow \vec{s}_0 \models \Omega$$

- This gives the outline of the CTL model checking procedure:
  1. Construct $\mathit{Satisfy}(\Omega)$ which is the set of states for which a given CTL-formula Ω holds and which we therefore define as follows:

$$\mathit{Satisfy}(\Omega) := \{\vec{s} \in \mathbb{S} \mid \vec{s} \models \Omega\}$$

  2. Check if the initial state of $\mathcal{L}$ is contained in this set, since

$$\mathcal{L} \models \Omega \Leftrightarrow \vec{s}_0 \in \mathit{Satisfy}(\Omega)$$

- How to compute the set *Satisfy* is of major concern now.

The model checking procedure: Overview

Preliminary: take CTL-formula and convert it into ENF and provide state labellings for LTS w. r. t. the atomic propositions of the CTL formula.

1. generate a parse tree for the CTL formula s.t. the leaves of the parse tree carry atomic propositions or the constant *true*
2. construct $Satisfy(\Omega)$ by processing the parse tree bottom-up, i. e., one computes the satisfaction sets of the leave nodes then for their parent nodes and so on and on ...
3. check if the initial state is contained in the satisfaction set $Satisfy(\Omega)$

Definition 3.5: Parse Tree

Given a CTL-formula Ω we construct a parse tree s.t.

- a leaf of the parse tree carries an atomic proposition or the constant *true* as occurring in a sub-formulae of the CTL-formula to be parsed
- the inner nodes carry combined operators as employed for connecting different state formulae, i. e., $op \in \{\neg, \wedge, \forall\bigcirc, \exists\bigcirc, \forall\,\mathsf{U}, \exists\,\mathsf{U}\}$.

$\longrightarrow$ Example 3.5: Parse tree for $\exists\bigcirc a \wedge \exists(b\,\mathsf{U}\,[\neg\forall(\mathit{true}\,\mathsf{U}\,\neg c)]$

(I) What do we need to do for the **leaves** of the parse tree,
i. e., . how do we compute $\mathit{Satisfy}(\phi)$ for $\phi := \mathit{true} \,|\, a \in \mathcal{AP}$?

1. $\phi = \mathit{true}$
   this set contains all states, since all states are satisfying the constant *true* formula, i. e., we have
   $$\mathit{Satisfy}(\phi) := \mathit{Satisfy}(\mathit{true}) := \mathbb{S}$$
2. $\phi \in \mathcal{AP}$
   we collect all states labelled with ϕ, i. e.,
   $$\mathit{Satisfy}(\phi) := \{\vec{s} \in \mathbb{S} \mid \mathcal{L}(\vec{s}) = \phi\}$$

(II) What do we need to do for the **inner** nodes of the parse tree?

1. Simple case covering the computation of $\mathit{Satisfy}(\phi)$ for

$$\phi := \neg\rho \,|\rho' \wedge \rho'' \,|\exists \bigcirc \rho$$

- $\underline{\phi = \neg\rho}$: $\mathit{Satisfy}(\phi)$ is the complement of $\mathit{Satisfy}(\rho)$ w. r. t. $\mathbb{S}$

$$\mathit{Satisfy}(\phi) := \mathbb{S} \setminus \mathit{Satisfy}(\rho)$$

- $\underline{\phi = \rho' \wedge \rho''}$: $\mathit{Satisfy}(\phi)$ is the intersection of the satisfaction sets of ρ' and ρ'':

$$\mathit{Satisfy}(\phi) := \mathit{Satisfy}(\rho') \cap \mathit{Satisfy}(\rho'')$$

- $\underline{\phi = \exists \bigcirc \rho}$: $\mathit{Satisfy}(\phi)$ are all those states which predecessors satisfy ρ, i. e.,

$$\mathit{Satisfy}(\phi) := \{\vec{s} \in \mathbb{S} \mid \mathcal{P}ost(\vec{s}) \cap \mathit{Satisfy}(\rho) \neq \emptyset\}$$

$\longrightarrow$ Example 3.6: $\mathit{Satisfy}(\exists \bigcirc \rho)$

(II) Handling of **inner** nodes of the parse tree (continued).

2. Complex case requires fixed point computation for obtaining $\mathit{Satisfy}(\phi)$ in case

$$\phi := \rho' \,\mathsf{U}\, \rho'' \mid \exists\Box\rho$$

- $\phi = \exists(\rho' \,\mathsf{U}\, \rho'')$:

$$\begin{aligned} \mathit{Satisfy}_0(\phi) &:= \mathit{Satisfy}(\rho'') \\ \mathit{Satisfy}_{i+1}(\phi) &:= \mathit{Satisfy}_i(\phi) \cup \\ &\quad \{\vec{s} \in \mathit{Satisfy}(\rho') \mid \mathcal{P}ost(\vec{s}) \cap \mathit{Satisfy}_i(\phi) \neq \emptyset\} \end{aligned}$$

- $\phi = \exists\Box\rho$:

$$\begin{aligned} \mathit{Satisfy}_0(\phi) &:= \mathit{Satisfy}(\rho) \\ \mathit{Satisfy}_{i+1}(\phi) &:= \{\vec{s} \in \mathit{Satisfy}_i(\rho) \mid \mathcal{P}ost(\vec{s}) \cap \mathit{Satisfy}_i(\phi) \neq \emptyset\} \end{aligned}$$

$\longrightarrow$ Example 3.7: Model Checking of "weather" LTS

1. Witnesses and counter examples:
   - path demonstrating $\mathcal{L} \models \phi$ is denoted **witnesses**
   - path demonstrating $\mathcal{L} \not\models \phi$ is denoted **counter example**.
2. A last operator (eventually):

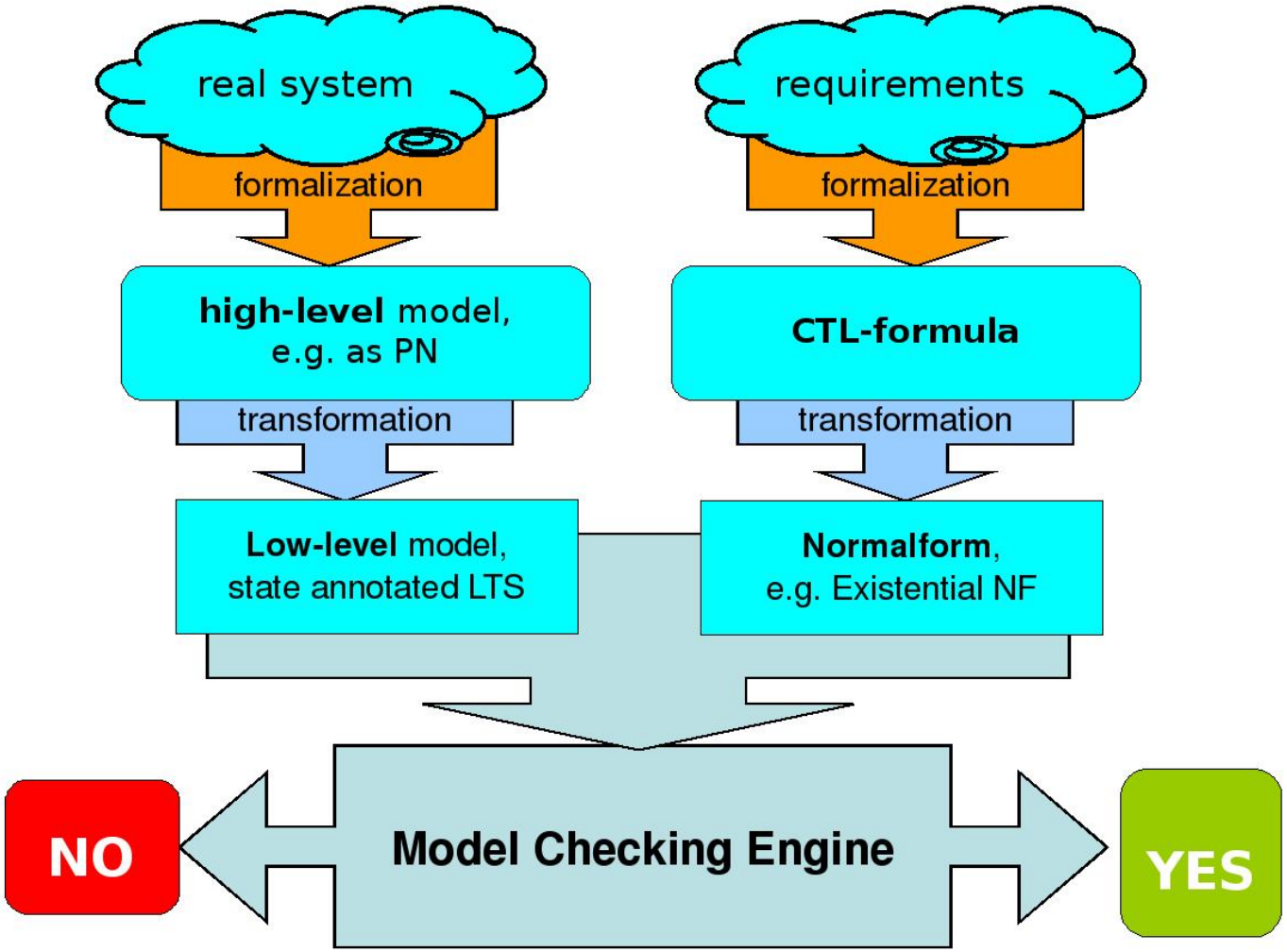
Definition 3.6: The eventually operator ($\Diamond$)

- potentially: $\exists\Diamond\phi := \exists(\mathit{true}\,\mathsf{U}\,\phi)$

  at least one path π goes at least through one state where ϕ holds.
- inevitable: $\forall\Diamond\phi := \forall(\mathit{true}\,\mathsf{U}\,\phi)$

  all paths go at least through one state there ϕ holds.

Summary


real system
formalization
requirements
formalization
high-level model, e.g. as PN
CTL-formula
transformation
transformation
Low-level model, state annotated LTS
Normalform, e.g. Existential NF
NO
Model Checking Engine
YES