Chapter 3

Model Checking Branching Temporal Logic

Pnueli (1977) has introduced linear temporal logic to the computer science community for the specification and verification of reactive systems. In the previous chapter we have treated one important exponent of linear temporal logic, called PLTL. This temporal logic is called linear, since the (qualitative notion of) time is linear: at each moment of time there is only one possible successor state and thus only one possible future. Technically speaking, this follows from the fact that the interpretation of linear temporal logic-formulas using the satisfaction relation $\models$ is defined in terms of a model in which a state s has precisely one successor state $R(s)$. Thus for each state s the model thus generates a unique infinite sequence of states $s, R(s), R(R(s)), \ldots$. A sequence of states represents a computation. Since the semantics of linear temporal logic is based on such "sequence-generating" models, the temporal operators X, U, F and G in fact describe the ordering of events along a *single* time path, i.e. a single computation of a system.

In the early eighties another type of temporal logic for specification and verification purposes which is not based on a linear notion of time, but on a branching notion of time was introduced (Clarke and Emerson, 1980). This logic is formally based on models where at each moment there may be several different possible futures.[1] Due to this branching notion of time, this class of temporal logic is

[1]This does not imply that linear temporal logic cannot be interpreted over a branching model. It is possible to consider this logic over sequences of state, i.e. possible traversals through the branching model. Notice that in Chapter2 we took a different approach by directly defining linear temporal logic over sequences.

called *branching temporal logic*. Technically speaking, branching boils down to the fact that a state can have different possible successor states. Thus $R(s)$ is a (non-empty) set of states, rather than a single state as for PLTL. The underlying notion of the semantics of a branching temporal logic is thus a *tree* of states rather than a sequence. Each path in the tree is intended to represent a single possible computation. The tree itself thus represents all possible computations. More precisely, the tree rooted at state s represents all possible infinite computations that start in s.

The temporal operators in branching temporal logic allow the expression of properties of (some or all) computations of a system. For instance, the property $\mathsf{EF}\,\phi$ denotes that there exists a computation along which $\mathsf{F}\,\phi$ holds. That is, it states that there is at least one possible computation in which a state is eventually reached that fulfills ϕ. This does, however, not exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which ϕ is never fulfilled. The property $\mathsf{AF}\,\phi$, for instance, differs from this existential property over computations in that it requires that all computations satisfy the property $\mathsf{F}\,\phi$.

The existence of two types of temporal logic — linear and branching temporal logic — has resulted in the development of two model checking "schools", one based on linear and one based on branching temporal logic. Although much can be said about the differences and the appropriateness of linear versus branching temporal logic, there are, in our opinion, two main issues that justify the treatment of model checking linear and branching temporal logic in these lecture notes:

- The *expressiveness* of many linear and branching temporal logics is incomparable. This means that some properties that are expressible in a linear temporal logic cannot be expressed in certain branching temporal logic, and vice versa.
- The traditional *techniques* used for efficient model checking of linear tem-

poral logic are quite different from those used for efficient model checking of branching temporal logic. (Although some promising unifying developments are currently taking place.) This results, for instance, in significantly different complexity results.

Various types of branching temporal logic have been proposed in the literature. They basically differ in expressiveness, i.e. the type of formulas that one can state in the logic. To mention a few important ones in increasing expressive power:

- Hennessy-Milner logic (1985)
- Unified System of Branching-Time Logic (Ben-Ari, Manna and Pnueli, 1983)
- Computation Tree Logic (Clarke and Emerson, 1980)
- Extended Computation Tree Logic (Clarke and Emerson, 1981) and
- Modal μ-Calculus (Kozen, 1983).

Modal μ-calculus is thus the most expressive among these languages, and Hennessy-Milner logic is the least expressive. The fact that the modal μ-calculus is the most expressive means that for any formula ϕ expressed in one of the other types of logic mentioned, an equivalent formula ψ in the modal μ-calculus can be given.

In this chapter we consider model checking of Computation Tree Logic (CTL). This is not only the temporal logic that was used originally by Clarke and Emerson (1981) and (in a slightly different form) by Quielle and Sifakis (1982) for model checking, but — more importantly — it can be considered as a branching counterpart of PLTL, the linear temporal logic that we considered in the first chapter, for which efficient model checking is possible.

3.1 Syntax of CTL

The syntax of computation tree logic is defined as follows. The most elementary expressions in CTL are atomic propositions, as in the definition of PLTL. The set

of atomic propositions is denoted by AP with typical elements p, q, and r. We define the syntax of CTL in Backus-Naur Form:

Definition 18. (**Syntax of computation tree logic**)
For $p \in AP$ the set of CTL-formulas is defined by:

$$\phi \quad ::= \quad p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{EX}\,\phi \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi].$$

Four temporal operators are used:

- EX (pronounced "for some path next")
- E (pronounced "for some path")
- A (pronounced "for all paths") and
- U (pronounced "until").

X and U are the linear temporal operators that express a property over a single path, whereas E expresses a property over soma path, and A expresses a property over all paths. The existential and universal path operators E and A can be used in combination with either X or U. Note that the operator AX is not elementary and is defined below.

The boolean operators true, false, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the usual way (see the previous chapter). Given that $\mathsf{F}\,\phi = \text{true}\,\mathsf{U}\,\phi$ we define the following abbreviations:

$$\begin{aligned} \mathsf{EF}\,\phi &\equiv \mathsf{E}[\text{true}\,\mathsf{U}\,\phi] \\ \mathsf{AF}\,\phi &\equiv \mathsf{A}[\text{true}\,\mathsf{U}\,\phi]. \end{aligned}$$

$\mathsf{EF}\,\phi$ is pronounced "ϕ holds potentially" and $\mathsf{AF}\,\phi$ is pronounced "ϕ is inevitable". Since $\mathsf{G}\,\phi = \neg\mathsf{F}\,\neg\phi$ and $\mathsf{A}\,\phi = \neg\mathsf{E}\,\neg\phi$ we have in addition: [2]

$$\begin{aligned}\mathsf{EG}\,\phi &\equiv \neg\mathsf{AF}\,\neg\phi\\ \mathsf{AG}\,\phi &\equiv \neg\mathsf{EF}\,\neg\phi\\ \mathsf{AX}\,\phi &\equiv \neg\mathsf{EX}\,\neg\phi.\end{aligned}$$

For instance, we have

$$\begin{aligned}&\neg\mathsf{A}\,(\mathsf{F}\,\neg\phi)\\ \Leftrightarrow\ &\{\ \mathsf{A}\,\psi \equiv \neg\mathsf{E}\,\neg\psi\ \}\\ &\neg\,\neg\mathsf{E}\,\neg\,(\mathsf{F}\,\neg\phi)\\ \Leftrightarrow\ &\{\ \mathsf{G}\,\psi \equiv \neg\mathsf{F}\,\neg\phi;\ \text{calculus}\ \}\\ &\mathsf{EG}\,\phi.\end{aligned}$$

$\mathsf{EG}\,\phi$ is pronounced "potentially always ϕ", $\mathsf{AG}\,\phi$ is pronounced "invariantly ϕ" and $\mathsf{AX}\,\phi$ is pronounced "for all paths next ϕ". The operators E and A bind equally strongly and have the highest precedence among the unary operators. The binding power of the other operators is identical to that of linear temporal logic PLTL. Thus, for example, $(\mathsf{AG}\,p) \Rightarrow (\mathsf{EG}\,q)$ is simply denoted by $\mathsf{AG}\,p \Rightarrow \mathsf{EG}\,q$, and should not be confused with $\mathsf{AG}\,(p \Rightarrow \mathsf{EG}\,q)$.

Example 18. *Let* $AP = \{\, x = 1, x < 2, x \geqslant 3 \,\}$ *be the set of atomic propositions.*

- *Examples of* CTL*-formulas are:* $\mathsf{EX}\,(x = 1)$, $\mathsf{AX}\,(x = 1)$, $x < 2 \vee x = 1$, $\mathsf{E}\,[(x < 2)\,\mathsf{U}\,(x \geqslant 3)]$ *and* $\mathsf{AF}\,(x < 2)$.
- *The expression* $\mathsf{E}\,[x = 1 \wedge \mathsf{AX}\,(x \geqslant 3)]$ *is, for example, not a* CTL*-formula, since* $x = 1 \wedge \mathsf{AX}\,(x \geqslant 3)$ *is neither a next- nor an until-formula. The expression* $\mathsf{EF}\,[\mathsf{G}\,(x = 1)]$ *is also not a* CTL*-formula.* $\mathsf{EG}\,[x = 1 \wedge \mathsf{AX}\,(x \geqslant 3)]$ *is, however, a well-formed* CTL*-formula. Likewise* $\mathsf{EF}\,[\mathsf{EG}\,(x = 1)]$ *and* $\mathsf{EF}\,[\mathsf{AG}\,(x = 1)]$ *are well-formed* CTL*-formulas.*

[2] The equation $\mathsf{A}\,\phi = \neg\mathsf{E}\,\neg\phi$ only holds for ϕ where ϕ is a CTL-formula for which the outermost existential or universal path quantifier is "stripped off". It allows, for instance, rewriting $\mathsf{E}\,[\neg\mathsf{F}\,\neg\phi]$ into $\neg\mathsf{AF}\,\neg\phi$.

(End of example.)

The syntax of CTL requires that the linear temporal operators X, F, G, and U are immediately preceded by a path quantifier E or A. If this restriction is dropped, then the more expressive branching temporal logic CTL* (Clarke and Emerson, 1981) is obtained. The logic CTL* permits an arbitrary PLTL-formula to be preceded by either E or A. It contains, for instance, $\mathsf{E}\,[p \wedge \mathsf{X}\,q]$ and $\mathsf{A}\,[\mathsf{F}\,p \wedge \mathsf{G}\,q]$, formulas which are not syntactical terms of CTL. CTL* can therefore be considered as *the* branching counterpart of PLTL since each PLTL sub-formula can be used in a CTL*-formula. The precise relationship between PLTL, CTL and CTL* will be described in Section 3.3. We do not consider model checking CTL* in these lecture notes, since model checking of CTL* is of intractable complexity: the model checking problem for this logic is PSPACE-complete in the size of the system specification (Clarke, Emerson and Sistla, 1986). We therefore consider CTL for which more efficient model checking algorithms do exist. Although CTL does not possess the full expressive power of CTL*, various (industrial) case studies have shown that it is frequently sufficiently powerful to express most required properties.

3.2 Semantics of CTL

As we have seen in the previous chapter, the interpretation of the linear temporal logic PLTL is defined in terms of a model $\mathcal{M} = (S, R, \mathit{Label})$ where S is a set of states, *Label* an assignment of atomic propositions to states, and R a total function that assigns a unique successor state to any given state. Since the successor of state s, $R(s)$, is unique, the model $\mathcal{M}$ generates for each state s a sequence of states $s, R(s), R(R(s)), \ldots$. These sequences represent computation paths starting at s, and since PLTL-formulas refer to a single path, the interpretation of PLTL is defined in terms of such sequences.

Branching temporal logic does, however, not refer to a single computation path, but to some (or all) possible computation paths. A single sequence is therefore insufficient to model this. In order to adequately represent the moments at which branching is possible, the notion of sequence is replaced by the notion

of a *tree*. Accordingly a CTL-model is a "tree-generating" model. Formally, this is expressed as follows:

Definition 19. (**CTL-Model**)
A CTL-model is a triple $\mathcal{M} = (S, R, Label)$ where

- S is a non-empty set of states,
- $R \subseteq S \times S$ is a total relation on S, which relates to $s \in S$ its possible successor states,
- $Label : S \longrightarrow 2^{AP}$, assigns to each state $s \in S$ the atomic propositions $Label(s)$ that are valid in s.

$\mathcal{M}$ is also known as a *Kripke structure* [3], since Kripke used similar structures to provide a semantics for modal logic, a kind of logic that is closely related to temporal logic (Kripke, 1963).

Notice that the only difference to a PLTL-model is that R is now a total relation rather than a total function. A relation $R \subseteq S \times S$ is total if and only if it relates to each state $s \in S$ at least one successor state: $\forall s \in S.(\exists s' \in S.(s, s') \in R)$.

Example 19. *Let $AP = \{ x = 0, x = 1, x \neq 0 \}$ be a set of atomic propositions, $S = \{ s_0, \ldots, s_3 \}$ be a set of states with labelling $Label(s_0) = \{ x \neq 0 \}$, $Label(s_1) = Label(s_2) = \{ x = 0 \}$, and $Label(s_3) = \{ x = 1, x \neq 0 \}$, and transition relation R be given by*

$$R = \{ (s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_2, s_3), (s_3, s_2) \}.$$

$\mathcal{M} = (S, R, Label)$ is a CTL-model and is depicted in Figure 3.1(a). Here states are depicted by circles, and the relation R is denoted by arrows, i.e. there is an arrow from s to s' if and only if $(s, s') \in R$. The labelling $Label(s)$ is indicated beside the state s. *(End of example.)*

[3] Although usually a Kripke structure is required to have an identified set of initial state $S_0 \subseteq S$ with $S_0 \neq \varnothing$.

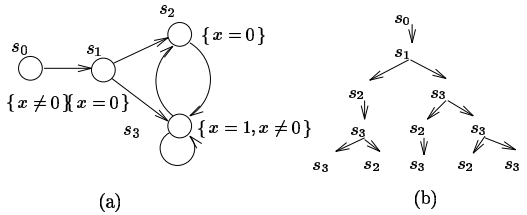


Figure 3.1: An example of a CTL-model and a (prefix of) of one of its infinite computation trees

Before presenting the semantics we introduce some auxiliary concepts. Let $\mathcal{M} = (S, R, Label)$ be a CTL-model.

Definition 20. (**Path**)
A *path* is an infinite sequence of states $s_0\, s_1\, s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$.

Let $\sigma \in S^\omega$ denote a path (of states). For $i \geqslant 0$ we use $\sigma[i]$ to denote the $(i{+}1)$-th element of σ, i.e. if $\sigma = t_0\, t_1\, t_2 \ldots$ then $\sigma[i] = t_i$, where t_i is a state.

Definition 21. (**Set of paths starting in a state**)
The set of paths starting in state s of the model $\mathcal{M}$ is defined by

$$P_{\mathcal{M}}(s) = \{ \sigma \in S^\omega \mid \sigma[0] = s \}.$$

For any CTL-model $\mathcal{M} = (S, R, Label)$ and state $s \in S$ there is an infinite computation tree with root labelled s such that (s', s'') is an arc in the tree if and only if $(s', s'') \in R$. A state s for which $p \in Label(s)$ is sometimes called a p-state. σ is called a p-path if it consists solely of p-states.

Example 20. *Consider the CTL-model of Figure 3.1(a). A finite prefix of the infinite computation tree rooted at state s_0 is depicted in Figure 3.1(b). Examples of paths are $s_0\, s_1\, s_2\, s_3^\omega$, $s_0\, s_1\, (s_2\, s_3)^\omega$ and $s_0\, s_1\, (s_3\, s_2)^*\, s_3^\omega$. $P_{\mathcal{M}}(s_3)$, for example, equals the set $\{ (s_3\, s_2)^* s_3^\omega, (s_3^+\, s_2)^\omega \}$.* *(End of example.)*

The semantics of CTL is defined by a satisfaction relation (denoted by $\models$) between a model $\mathcal{M}$, one of its states s, and a formula ϕ. As before, we write $\mathcal{M}, s \models \phi$ rather than $((\mathcal{M}, s), \phi) \in \models$. We have $(\mathcal{M}, s) \models \phi$ if and only if ϕ is valid in state s of model $\mathcal{M}$. As in the previous chapter, we omit $\mathcal{M}$ if the model is clear from the context.

Definition 22. (**Semantics of** CTL)
Let $p \in AP$ be an atomic proposition, $\mathcal{M} = (S, R, \mathit{Label})$ be a CTL-model, $s \in S$, and ϕ, ψ be CTL-formulas. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s \models p & \text{iff } p \in \mathit{Label}(s) \\
s \models \neg \phi & \text{iff } \neg (s \models \phi) \\
s \models \phi \vee \psi & \text{iff } (s \models \phi) \vee (s \models \psi) \\
s \models \mathsf{EX}\, \phi & \text{iff } \exists \sigma \in P_{\mathcal{M}}(s).\, \sigma[1] \models \phi \\
s \models \mathsf{E}\,[\phi\, \mathsf{U}\, \psi] & \text{iff } \exists \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge\ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi)) \\
s \models \mathsf{A}\,[\phi\, \mathsf{U}\, \psi] & \text{iff } \forall \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge\ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi)).
\end{array}
$$

The interpretations for atomic propositions, negation and conjunction are as usual. $\mathsf{EX}\, \phi$ is valid in state s if and only if there exists some path σ starting in s such that in the next state of this path, state $\sigma[1]$, the property ϕ holds. $\mathsf{A}\,[\phi\, \mathsf{U}\, \psi]$ is valid in state s if and only if every path starting in s has an initial finite prefix (possibly only containing s) such that ψ holds in the last state of this prefix and ϕ holds at all other states along the prefix. $\mathsf{E}\,[\phi\, \mathsf{U}\, \psi]$ is valid in s if and only if there exists a path starting in s that satisfies the property $\phi\, \mathsf{U}\, \psi$.

The interpretation of the temporal operators $\mathsf{AX}\, \phi$, $\mathsf{EF}\, \phi$, $\mathsf{EG}\, \phi$, $\mathsf{AF}\, \phi$ and $\mathsf{AG}\, \phi$ can be derived using the above definition. To illustrate such a derivation we derive the formal semantics of $\mathsf{EG}\, \phi$.

$$
\begin{array}{ll}
 & s \models \mathsf{EG}\, \phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{EG} \ \} \\
 & s \models \neg\, \mathsf{AF}\, \neg \phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{AF} \ \} \\
 & s \models \neg\, \mathsf{A}\,[\mathsf{true}\, \mathsf{U}\, \neg \phi] \\
\Leftrightarrow & \{ \text{ semantics of } \neg \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \neg\, (s \models \mathsf{A}\,[\mathsf{true}\, \mathsf{U}\, \neg \phi]) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{A}\,[\phi\, \mathsf{U}\, \psi] \ \} \\
 & \neg\, [\forall \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\sigma[j] \models \neg \phi \ \wedge\ (\forall 0 \leqslant k < j.\sigma[k] \models \mathsf{true}))] \\
\Leftrightarrow & \{ \ s \models \mathsf{true} \text{ for all states } s \ \} \\
 & \neg\, [\forall \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\sigma[j] \models \neg \phi)] \\
\Leftrightarrow & \{ \text{ semantics of } \neg \text{; predicate calculus } \} \\
 & \exists \sigma \in P_{\mathcal{M}}(s).\neg\, (\exists j \geqslant 0.\, \neg (\sigma[j] \models \phi)) \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
 & \exists \sigma \in P_{\mathcal{M}}(s).(\forall j \geqslant 0.\, \sigma[j] \models \phi)\,.
\end{array}
$$

Thus $\mathsf{EG}\, \phi$ is valid in state s if and only if there exists some path starting at s such that for each state on this path the property ϕ holds. In a similar way one can derive that $\mathsf{AG}\, \phi$ is valid in state s if and only if for all states on any path starting at s the property ϕ holds. $\mathsf{EF}\, \phi$ is valid in state s if and only if ϕ holds eventually along some path that starts in s. $\mathsf{AF}\, \phi$ is valid iff this property holds for all paths that start in s. The derivation of the formal interpretation of these temporal operators is left to the interested reader.

Example 21. Let the CTL*-model $\mathcal{M}$ be given as depicted in Figure 3.2(a). In the figures below the validity of several formulas is indicated for all states of $\mathcal{M}$. A state is colored black if the formula is valid in that state, and otherwise colored white.*

- *The formula $\mathsf{EX}\, p$ is valid for all states, since all states have some direct successor state that satisfies p.*
- *$\mathsf{AX}\, p$ is not valid for state s_0, since a possible path starting at s_0 goes directly to state s_2 for which p does not hold. Since the other states have only direct successors for which p holds, $\mathsf{AX}\, p$ is valid for all other states.*
- *For all states except state s_2, it is possible to have a computation (such as $s_0\, s_1\, s_3^{\omega}$) for which p is globally valid. Therefore $\mathsf{EG}\, p$ is valid in these states. Since $p \notin \mathit{Label}(s_2)$ there is no path starting at s_2 for which p is globally valid.*

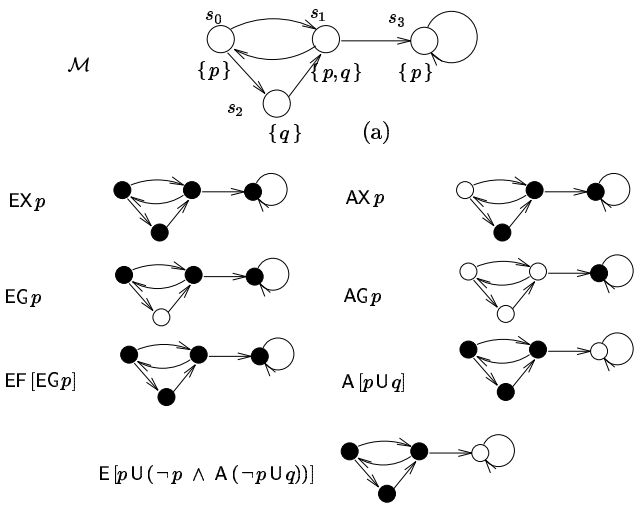


Figure 3.2: Interpretation of several CTL-formulas in an example of a model

- AGp *is only valid for* s_3 *since its only path,* s_3^ω*, always visits a state in which* p *holds. For all other states it is possible to have a path which contains* s_2*, which does not satisfy* p*. So, for these states* AGp *is not valid.*

- EF [EGp] *is valid for all states, since from each state a state (either* s_0*,* s_1 *or* s_3*) can be reached eventually from which some computation starts along which* p *is globally valid.*

- A[p U q] *is not valid in* s_3 *since its only computation (*s_3^ω*) never reaches a state for which* q *holds. For all other states this is, however true and in addition, the proposition* p *is valid before* q *is valid.*

- *Finally,* E[p U ($\neg p$ $\wedge$ A ($\neg p$ U q))] *is not valid in* s_3*, since from* s_3 *a* q*-state can never be reached. For the states* s_0 *and* s_1 *the property is valid, since state* s_2 *can be reached from these states via a* p*-path,* $\neg p$ *is valid in* s_2*, and from* s_2 *all possible paths satisfy* $\neg p$ U q*, since* s_2 *is a* q*-state. For instance, for state* s_0 *the path* $(s_0\, s_2\, s_1)^\omega$ *satisfies* p U ($\neg p$ $\wedge$ A ($\neg p$ U q)) *since* $p \in Label(s_0)$*,* $p \notin Label(s_2)$ *and* $q \in Label(s_1)$*. For state* s_2 *the property is valid since* p *is invalid in* s_2 *and for all paths starting at* s_2 *the next state is a* q*-state. Thus, the property* $\neg p$ $\wedge$ A ($\neg p$ U q) *is reached after a path of length 0.*

(End of example.)

3.3 Expressiveness of CTL, CTL* and PLTL

In order to understand better the relationship between PLTL, CTL and CTL* we present an alternative characterization of the syntax of CTL and CTL* in terms of PLTL. We do this by explicitly distinguishing between state formulas, i.e. expressions over states, and path formulas, i.e. properties that are supposed to hold along paths. Table 3.1 summarizes the syntax of the three types of logic considered.[4] (Although we did not consider the formal semantics of CTL* in these lecture notes we assume that this interpretation is intuitively clear from the interpretation of CTL.)

Clearly, by inspecting the syntax it can be seen that CTL is a subset of CTL*: all CTL-formulas belong to CTL*, but the reverse does not hold (syntactically). Since CTL and CTL* are interpreted over branching models, whereas PLTL is interpreted over sequences, a comparison between these three logics is not straightforward. A comparison is facilitated by changing the definition of PLTL slightly, such that its semantics is also defined in terms of a branching model. Although there are different ways of interpreting PLTL over branching structures — for instance, should ϕ in case of Xϕ hold for all or for some possible successor states of the current state? — the simplest and most common approach is to consider that a PLTL-formula ϕ holds for *all* paths. Since a PLTL-formula is usually implicitly universally quantified over all possible computations, this choice is well justified (Emerson and Halpern, 1986). This results in the following embedding of PLTL in terms of CTL*:

[4] Here we have taken an alternative way to define the syntax of CTL that facilitates the comparison. It is left to the reader to show that this alternative definition corresponds to Definition 18. Hereby, the reader should bear in mind that $\neg$ E $\neg \psi$ equals A ψ for path-formula ψ.

PLTL		$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \phi\,\mathsf{U}\,\phi$
CTL	state formulas	$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{E}\psi$
	path formulas	$\psi ::= \neg\psi \mid \mathsf{X}\phi \mid \phi\,\mathsf{U}\,\phi$
CTL*	state formulas	$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{E}\psi$
	path formulas	$\psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi\,\mathsf{U}\,\psi$

Table 3.1: Summary of syntax of PLTL, CTL and CTL*

Definition 23. (**Formulation of PLTL in terms of CTL***)
The state formulas of PLTL are defined by $\phi ::= \mathsf{A}\psi$ where ψ is a path formula. The path formulas are defined according to

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi\,\mathsf{U}\,\psi.$$

As a result, CTL, CTL* and PLTL are now interpreted in terms of the same model, and this common semantical basis allows a comparison. Let us first clarify how the expressiveness of two temporal logics is compared. Clearly, one logic is more expressive than another if it allows the expression of terms that cannot be expressed in the other. This syntactical criterion applies to CTL versus CTL*: the former is syntactically a subset of the latter. This criterion is in general too simple. More precisely, it does not exclude the fact that for some formula ϕ in one temporal logic $\mathcal{L}$, say, there does not exist an equivalent formula ψ — that syntactically differs from ϕ — in the temporal logic $\mathcal{L}'$. Here, by equivalent we mean:

Definition 24. (**Equivalence of formulas**)

Formulas ϕ and ψ are *equivalent* if and only if for all models $\mathcal{M}$ and states s:

$$\mathcal{M}, s \models \phi \text{ if and only if } \mathcal{M}, s \models \psi.$$

We are now in a position to define formally what it means for two temporal logics to be equally expressive.

Definition 25. (**Comparison of expressiveness**)
Temporal logic $\mathcal{L}$ and $\mathcal{L}'$ are *equally expressive* if for all models $\mathcal{M}$ and states s

$$\begin{aligned} & \forall \phi \in \mathcal{L}.\ (\exists \psi \in \mathcal{L}'.(\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}, s \models \psi)) \\ \wedge\ & \forall \psi \in \mathcal{L}'.\ (\exists \phi \in \mathcal{L}.(\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}, s \models \psi))\,. \end{aligned}$$

If only the first conjunct is valid, but not the second, then $\mathcal{L}$ is (strictly) *less expressive* than $\mathcal{L}'$.

Figure 3.3 depicts the relationship between the three logics considered in this

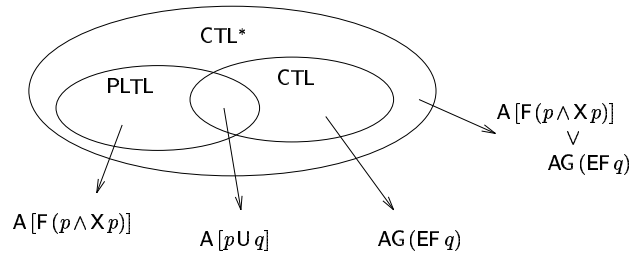


Figure 3.3: Relationship between PLTL, CTL and CTL*

section. It follows that CTL* is more expressive than both PLTL and CTL, whereas PLTL and CTL are incomparable. An example of a formula which distinguishes the part of the figure it belongs to is given for CTL, CTL* and PLTL.

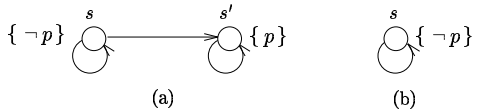
In PLTL, but not in CTL. For example, $\mathsf{A}[\mathsf{F}(p \wedge \mathsf{X}p)]$ is a PLTL-formula for which there does not exist an equivalent formula in CTL. The proof of this is by (Emerson & Halpern, 1986) and falls outside the scope of these lecture notes.

Another example of a PLTL-formula for which an equivalent formulation in CTL does not exist is:

$$\mathsf{A}[\mathsf{G}\,\mathsf{F}\,p \Rightarrow \mathsf{F}\,q]$$

which states that if p holds infinitely often, then q will be valid eventually. This is an interesting property which occurs frequently in proving the correctness of systems. For instance, a typical property for a communication protocol over an unreliable communication medium (such as a radio or infra-red connection) is that "if a message is being sent infinitely often, it will eventually arrive at the recipient".

In CTL, but not in PLTL. The formula $\mathsf{AG}\,\mathsf{EF}\,p$ is a CTL-formula for which there does not exist an equivalent formulation in PLTL. The property is of use in practice, since it expresses the fact that it is possible to reach a state for which p holds irrespective of the current state. If p characterizes a state where a certain error is repaired, the formula expresses that it is always possible to recover from a certain error. The proof sketch that for $\mathsf{AG}\,\mathsf{EF}\,p$ there does not exist an equivalent formulation in PLTL is as follows (Huth and Ryan, 1999). Let ϕ be a PLTL-formula such that $\mathsf{A}\,\phi$ is equivalent to $\mathsf{AG}\,\mathsf{EF}\,p$. Since $\mathcal{M}, s \models \mathsf{AG}\,\mathsf{EF}\,p$ in the left-hand figure below (a), it follows that $\mathcal{M}, s \models \mathsf{A}\,\phi$. Let $\mathcal{M}'$ be the sub-model of $\mathcal{M}$ shown in the right-hand diagram (b). The paths starting from s in $\mathcal{M}'$ are a subset of those starting from s in $\mathcal{M}$, so we have $\mathcal{M}', s \models \mathsf{A}\,\phi$. However, it is not the case that $\mathcal{M}', s \models \mathsf{AG}\,\mathsf{EF}\,p$, since p is never valid along the only path s^ω.

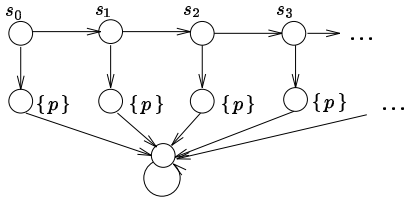


Relationship between CTL* and PLTL. The following holds for the relationship between PLTL and CTL* (Clarke and Draghicescu, 1988):

> *For any* CTL*-*formula* ϕ*, an equivalent* PLTL*-formula (if such exists) must be of the form* $\mathsf{A}\,f(\phi)$ *where* $f(\phi)$ *equals* ϕ *where all path-quantifiers are eliminated.*

For instance, for $\phi = \mathsf{EF}\,\mathsf{EG}\,p \Rightarrow \mathsf{AF}\,q$ we obtain $f(\phi) = \mathsf{F}\,\mathsf{G}\,p \Rightarrow \mathsf{F}\,q$. Thus ϕ is a distinguishing formula for CTL* if for ϕ its equivalent $\mathsf{A}\,f(\phi)$ is not a PLTL-formula.

As a final example of the difference in expressiveness of PLTL, CTL and CTL* consider the PLTL-formula $\mathsf{G}\,\mathsf{F}\,p$, infinitely often p. It is not difficult to see that prefixing this formula with an existential or a universal path quantifier leads to a CTL*-formula: $\mathsf{AG}\,\mathsf{F}\,p$ and $\mathsf{EG}\,\mathsf{F}\,p$ are CTL*-formulas. $\mathsf{AG}\,\mathsf{F}\,p$ is equivalent to $\mathsf{AG}\,\mathsf{AF}\,p$ — for any model $\mathcal{M}$ the validity of these two formulas is identical — and thus for $\mathsf{AG}\,\mathsf{F}\,p$ an equivalent CTL-formula does exist, since $\mathsf{AG}\,\mathsf{AF}\,p$ is a CTL-formula. For $\mathsf{EG}\,\mathsf{F}\,p$, however, no equivalent CTL-formula does exist. This can be seen by considering the following example model.



We have $s \models \mathsf{EG}\,\mathsf{EF}\,p$, since for the path $s_0\,s_1\,s_2\ldots$ for each s_j a p-state is eventually reachable. However, $s \not\models \mathsf{EG}\,\mathsf{F}\,p$, since there is no path starting in s such that p is infinitely often valid.

3.4 Specifying properties in CTL

In order to illustrate the way in which properties can be expressed in CTL we treat a simple two-process mutual exclusion program. Each process (P_1 and P_2) can be

in one of the following three states: the critical section (C), the attempting section (T), and the non-critical section (N). A process starts in the non-critical section and indicates that it wants to enter its critical section by entering its attempting section. It remains in the attempting section until it obtains access to its critical section, and from its critical section it moves to the non-critical section. The state of process P_i is denoted $P_i.s$, for i=1,2. Some required properties and their formal specification in CTL are as follows.

1. "It is not possible for both processes to be in their critical section at the same time".

   $$\mathsf{AG}\,[\,\neg\,(P_1.s = C \;\wedge\; P_2.s = C)]$$

2. "A process that wants to enter its critical section is eventually able to do so".

   $$\mathsf{AG}\,[P_1.s = T \;\Rightarrow\; \mathsf{AF}\,(P_1.s = C)]$$

3. "Processes must strictly alternate in having access to their critical section".

   $$\mathsf{AG}[P_1.s = C \;\Rightarrow\; \mathsf{A}\,(P_1.s = C\,\mathsf{U}\,(P_1.s \neq C \;\wedge\; \mathsf{A}(P_1.s \neq C\,\mathsf{U}\,P_2.s = C)))]$$

3.5 Automaton-based approach for CTL?

The model checking problem for CTL is to check for a given CTL-model $\mathcal{M}$, state $s \in S$, and CTL-formula ϕ whether $\mathcal{M}, s \models \phi$, that is, to establish whether the property ϕ is valid in state s of model $\mathcal{M}$. Here, it is assumed that $\mathcal{M}$ is *finite*. This means that the set S of states is finite (but non-empty).

For linear temporal logic we have seen that the model checking algorithm is strongly based on Büchi automata, a kind of automata that accepts infinite words. The key result that makes this approach possible is the fact that for each PLTL-formula ϕ, a Büchi automaton can be constructed that accepts precisely those (infinite) sequences of atomic propositions that make ϕ valid. This allows the reduction of model checking PLTL to known automata-theoretic problems.

When dealing with CTL the question is whether a similar method can be used. The answer is positive, since an automaton-based approach is possible also for CTL. The difference between PLTL and CTL is that the former expresses properties related to infinite sequences of states, whereas the latter focuses on infinite *trees* of states. This suggests transforming a CTL-formula into a Büchi-like automaton that accepts infinite trees (of states) rather than infinite sequences. And, indeed, each CTL-formula can be transformed into a *Büchi-tree* automaton. The time complexity of this transformation is *exponential* in the length of the CTL-formula under consideration, as in the PLTL case.

Recent work of Bernholtz, Vardi and Wolper (1994) has led to a significant improvement of the automaton-based approach for branching temporal logic. They propose to use a variant of non-deterministic tree automata, called (weak) alternating tree automata. The time complexity of their approach is linear in the length of the CTL-formula and the size of the system specification. An interesting aspect of this approach is that the space complexity of their algorithm is NLOGSPACE in the size of the model $\mathcal{M}$. This means that model checking can be done in space which is polynomial in the size of the system specification (rather than exponential, as is usual). To our knowledge, no tools have been constructed yet based on alternating tree automata.

Although these current developments are interesting and quite promising, there does exist an efficient and well-established method for model checking CTL based on another paradigm that is conceptually simpler. As originally shown by Clarke and Emerson (1981) model checking for CTL can be performed in a time complexity that is linear in the size of the formula (and the system specification). In these lecture notes we discuss this traditional scheme that has its roots in *fixed point theory*. This enables us to use a tool, SMV, which is based on this approach.

3.6 Model checking CTL

Suppose we want to determine whether the CTL-formula ϕ is valid in the finite CTL-model $\mathcal{M} = (S, R, \mathit{Label})$. The basic concept of the model checking algo-

rithm is to "label" each state $s \in S$ with the sub-formulas of ϕ that are valid in s. The set of sub-formulas of ϕ is denoted by $Sub(\phi)$ and is inductively defined as follows.

Definition 26. (Sub-formulas of a CTL-formula)
Let $p \in AP$, and ϕ, ψ be CTL-formulas. Then

$$\begin{aligned}
Sub(p) &= \{ p \} \\
Sub(\neg \phi) &= Sub(\phi) \cup \{ \neg \phi \} \\
Sub(\phi \vee \psi) &= Sub(\phi) \cup Sub(\psi) \cup \{ \phi \vee \psi \} \\
Sub(\mathsf{EX}\, \phi) &= Sub(\phi) \cup \{ \mathsf{EX}\, \phi \} \\
Sub(\mathsf{E}[\phi \,\mathsf{U}\, \psi]) &= Sub(\phi) \cup Sub(\psi) \cup \{ \mathsf{E}[\phi \,\mathsf{U}\, \psi] \} \\
Sub(\mathsf{A}[\phi \,\mathsf{U}\, \psi]) &= Sub(\phi) \cup Sub(\psi) \cup \{ \mathsf{A}[\phi \,\mathsf{U}\, \psi] \}.
\end{aligned}$$

(Note that $\phi \,\mathsf{U}\, \psi$ is not a sub-formula of $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$.) The labelling procedure mentioned above is performed iteratively, starting by labelling the states with the sub-formulas of length 1 of ϕ, i.e. the atomic propositions (and true and false) that occur in ϕ. (Actually this step is easy, since the function *Label* already provides this labelling.) In the $(i+1)$-th iteration of the labelling algorithm sub-formulas of length $i+1$ are considered and the states are labelled accordingly. The labels already assigned to states are used for that purpose, being sub-formulas of ϕ of length at most i ($i \geqslant 1$). For instance, if $\phi = \phi_1 \vee \phi_2$ then state s is labelled with ϕ, a formula of length $i+1$, if s is already labelled with ϕ_1 or with ϕ_2 in some previous iteration. The labelling algorithm ends by considering the sub-formula of length $\mid \phi \mid$, ϕ itself.

The model checking problem for CTL, deciding whether $\mathcal{M}, s \models \phi$, for a given model $\mathcal{M}$ and CTL-formula ϕ, can now be solved for any state s in S by considering its labelling:

$\mathcal{M}, s \models \phi$ if and only if s is "labelled" with ϕ.

function *Sat* (ϕ : *Formula*) : **set of** *State*;
(* precondition: true *)
begin
  if $\phi =$ true $\longrightarrow$ **return** S
  [] $\phi =$ false $\longrightarrow$ **return** $\varnothing$
  [] $\phi \in AP \longrightarrow$ **return** $\{ s \mid \phi \in Label(s) \}$
  [] $\phi = \neg \phi_1 \longrightarrow$ **return** $S - Sat(\phi_1)$
  [] $\phi = \phi_1 \vee \phi_2 \longrightarrow$ **return** $(Sat(\phi_1) \cup Sat(\phi_2))$
  [] $\phi = \mathsf{EX}\, \phi_1 \longrightarrow$ **return** $\{ s \in S \mid (s, s') \in R \wedge s' \in Sat(\phi_1) \}$
  [] $\phi = \mathsf{E}[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat_{EU}(\phi_1, \phi_2)$
  [] $\phi = \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat_{AU}(\phi_1, \phi_2)$
  fi
(* postcondition: $Sat(\phi) = \{ s \mid \mathcal{M}, s \models \phi \}$ *)
end

Table 3.2: Outline of main algorithm for model checking CTL

Actually, the model checking algorithm can be presented in a very compact and elegant way by determining for a given ϕ and $\mathcal{M}$:

$$Sat(\phi) = \{ s \in S \mid \mathcal{M}, s \models \phi \}$$

in an iterative way (as indicated above). By computing $Sat(\phi)$ according to the algorithm depicted in Table 3.2 the problem of checking $\mathcal{M}, s \models \phi$ reduces to checking $s \in Sat(\phi)$. (In the program text it is assumed that the model $\mathcal{M} = (S, R, Label)$ is a global variable.)

Notice that by computing $Sat(\phi)$ a more general problem than just checking whether $\mathcal{M}, s \models \phi$ is solved. In fact, it checks for *any* state s in $\mathcal{M}$ whether $\mathcal{M}, s \models \phi$, and not just for a given one. In addition, since $Sat(\phi)$ is computed in an iterative way by considering the sub-formulas of ϕ, the sets $Sat(\psi)$ for any sub-formula ψ of ϕ are computed, and thus $\mathcal{M}, s \models \psi$ can be easily checked as well.

The computation of $Sat(\phi)$ is done by considering the syntactical structure of ϕ. For $\phi =$ true the program just returns S, the entire state space of $\mathcal{M}$, thus

indicating that any state fulfills true. Accordingly, $Sat(\text{false}) = \varnothing$, since there is no state in which false is valid. For atomic propositions, the labelling $Label(s)$ provides all the information: $Sat(p)$ is simply the set of states that is labelled by $Label$ with p. For the negation $\neg\,\phi$ we compute $Sat(\phi)$ and take its complement with respect to S. Disjunction amounts to a union of sets. For $\mathsf{EX}\,\phi$ the set $Sat(\phi)$ is recursively computed and all states s are considered that can reach some state in $Sat(\phi)$ by traversing a single transition. Finally, for $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ the specific functions Sat_{EU} and Sat_{AU} are invoked that perform the computation of $Sat(\mathsf{E}\,[\phi\,\mathsf{U}\,\psi])$ and $Sat(\mathsf{A}\,[\phi\,\mathsf{U}\,\psi])$. These algorithms are slightly more involved, and their correctness is based on the computation of so-called *fixed points*. In order to understand these program fragments better, we first give a summary of the most important results and concepts of fixed point theory (based on partial orders[5]).

3.7 Fixed point theory based on posets

In this section we briefly recall some results and definitions from basic domain theory as far as they are needed to understand the fundamentals of model checking CTL.

Definition 27. (**Partial order**)
A binary relation $\sqsubseteq$ on set A is a *partial order* iff, for all $a, a', a'' \in A$:

1. $a \sqsubseteq a$ (reflexivity)
2. $(a \sqsubseteq a' \;\wedge\; a' \sqsubseteq a) \;\Rightarrow\; a = a'$ (anti-symmetry)
3. $(a \sqsubseteq a' \;\wedge\; a' \sqsubseteq a'') \;\Rightarrow\; a \sqsubseteq a''$ (transitivity).

The pair $\langle A, \sqsubseteq\rangle$ is a partially ordered set, or shortly, *poset*. If $a \not\sqsubseteq a'$ and $a' \not\sqsubseteq a$ then a and a' are said to be incomparable. For instance, for S a set of states,

[5] Another well-known variant of fixed-point theory is based on metric spaces — domains that are equipped with an appropriate notion of distance between any pair of its elements — where the existence of fixed points is guaranteed by Banach's contraction theorem for certain types of functions. This theory falls outside the scope of these lecture notes.

it follows that $\langle 2^S, \subseteq\rangle$, where 2^S denotes the power-set of S and $\subseteq$ the usual subset-relation, is a poset.

Definition 28. (**Least upper bound**)
Let $\langle A, \sqsubseteq\rangle$ be a poset and $A' \subseteq A$.

1. $a \in A$ is an *upper bound* of A' if and only if $\forall\, a' \in A' : a' \sqsubseteq a$.
2. $a \in A$ is a *least* upper bound (lub) of A', written $\sqcup A'$, if and only if
   (a) a is an upper bound of A' and
   (b) $\forall\, a'' \in A.\, a''$ is an upper bound of $A' \;\Rightarrow\; a \sqsubseteq a''$.

The concepts of the lower bound of $A' \subseteq A$, and the notion of greatest lower bound, denoted $\sqcap A'$, can be defined similarly. Let $\langle A, \sqsubseteq\rangle$ be a poset.

Definition 29. (**Complete lattice**)
$\langle A, \sqsubseteq\rangle$ is a *complete lattice* if for each $A' \subseteq A$, $\sqcup A'$ and $\sqcap A'$ do exist.

A complete lattice has a unique least element $\sqcap A = \bot$ and a unique greatest element $\sqcup A = \top$.

Example 22. *Let $S = \{0, 1, 2\}$ and consider $\langle 2^S, \subseteq\rangle$. It is not difficult to check that for any two subsets of 2^S a least upper bound and greatest upper bound do exist. For instance, for $\{0, 1\}$ and $\{0, 2\}$ the lub is $\{0, 1, 2\}$ and the glb $\{0\}$. That is, the poset $\langle 2^S, \subseteq\rangle$ is a complete lattice where intersection and union correspond to $\sqcap$ and $\sqcup$. The least and greatest element of this example lattice are $\varnothing$ and S.* *(End of example.)*

Definition 30. (**Monotonic function**)
Function $F : A \longrightarrow A$ is *monotonic* if for each $a_1, a_2 \in A$ we have $a_1 \sqsubseteq a_2 \;\Rightarrow\; F(a_1) \sqsubseteq F(a_2)$.

Thus F is monotonic if it preserves the ordering $\sqsubseteq$. For instance, the function $F = S \cup \{0\}$ is monotonic on $\langle 2^S, \subseteq\rangle$.

Definition 31. (Fixed point)
For function $F : A \longrightarrow A$, $a \in A$ is called a *fixed point* of F if $F(a) = a$.

a is the *least* fixed point of F if for all $a' \in A$ such that $F(a') = a'$ we have $a \sqsubseteq a'$. The greatest fixed point of F is defined similarly. The following important result for complete lattices is from Knaster and Tarski (1955).

Theorem 32.
Every monotonic function over a complete lattice has a complete lattice of fixed points (and hence a unique greatest and unique least fixed point).

Notice that the lattice of fixed points is in general different from the lattice on which the monotonic function is defined.

The least fix-point of monotonic function F on the complete lattice $\langle A, \sqsubseteq \rangle$ can be computed by $\sqcup_i F^i(\bot)$, i.e. the least upper bound of the series $\bot, F(\bot), F(F(\bot)), \ldots$. This series is totally ordered under $\sqsubseteq$, that is, $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$ for all i. This roughly follows from the fact that $\bot \sqsubseteq F(\bot)$, since $\bot$ is the least element in the lattice, and the fact that $F(\bot) \sqsubseteq F(F(\bot))$, since F is monotonic. (In fact this second property is the key step in a proof by induction that $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$.) The greatest fixed point can be computed by $\sqcap_i F^i(\top)$, that is the greatest lower bound of the series $\top, F(\top), F(F(\top)), \ldots$, a sequence that is totally ordered by $F^{i+1}(\top) \sqsubseteq F^i(\top)$ for all i.

In the following subsection we are interested in the construction of monotonic functions on lattices of CTL-formulas. Such functions are of particular interest to us, since each monotonic function on a complete lattice has a unique least and greatest fixed point (cf. Theorem 32). These fixed points can be easily computed and such computations form the key to the correctness of functions Sat_{EU} and Sat_{AU}.

3.8 Fixed point characterization of CTL-formulas

The labelling procedures for $\mathsf{E}[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}[\phi\,\mathsf{U}\,\psi]$ are based on a fixed point characterization of CTL-formulas. Here, the technique is to characterize $\mathsf{E}[\phi\,\mathsf{U}\,\psi]$ as the least (or greatest) fixed point of a certain function (on CTL-formulas), and to apply an iterative algorithm — suggested by Knaster and Tarski's result — to compute such fixed points in order to carry out the labelling of the states. To do this basically two main issues need to be resolved.

1. First, a *complete lattice on* CTL*-formulas* needs to be defined such that the existence (and uniqueness) of least and greatest fixed points is guaranteed. The basis of this lattice is a partial order relation on CTL-formulas.
2. Secondly, *monotonic functions on* CTL*-formulas* have to be determined such that $\mathsf{E}[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}[\phi\,\mathsf{U}\,\psi]$ can be characterized as least (or greatest) fixed points of these functions. For this purpose it turns out that an axiomatization of CTL is useful.

In the sequel we start by defining a complete lattice of formulas, after which we introduce some axioms that are helpful in finding the monotonic functions mentioned in step 2. Finally, we show that $\mathsf{E}[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}[\phi\,\mathsf{U}\,\psi]$ are particular fixed points of these functions.

A complete lattice of CTL-formulas

The partial order $\sqsubseteq$ on CTL-formulas is defined by associating with each formula ϕ the set of states in $\mathcal{M}$ for which ϕ holds. Thus ϕ is identified with the set

$$[\![\phi]\!] = \{\, s \in S \mid \mathcal{M}, s \models \phi \,\}.$$

(Strictly speaking $[\![\ ]\!]$ is a function of $\mathcal{M}$ as well, i.e. $[\![\ ]\!]_{\mathcal{M}}$ would be a more correct notation, but since in all cases $\mathcal{M}$ is known from the context we omit this

subscript.) The basic idea now is to define the order $\sqsubseteq$ by:

$$\phi \sqsubseteq \psi \text{ if and only if } [\![\phi]\!] \subseteq [\![\psi]\!].$$

Stated in words, $\sqsubseteq$ corresponds to $\subseteq$, the well-known subset relation on sets. Notice that $[\![\phi]\!] \subseteq [\![\psi]\!]$ is equivalent to $\phi \Rightarrow \psi$. Clearly, $\langle 2^S, \subseteq \rangle$ is a poset, and given that for any two subsets $S_1, S_2 \in 2^S$, $S_1 \cap S_2$ and $S_1 \cup S_2$ are defined, it follows that it is a complete lattice. Here, $\cap$ is the lower bound construction and $\cup$ the upper bound construction. The least element $\bot$ in the lattice $\langle 2^S, \subseteq \rangle$ is $\varnothing$ and the greatest element $\top$ equals S, the set of all states.

Since $\sqsubseteq$ directly corresponds to $\subseteq$, it follows that the poset $\langle \mathsf{CTL}, \sqsubseteq \rangle$ is a complete lattice. The lower bound construction in this lattice is conjunction:

$$[\![\phi]\!] \cap [\![\psi]\!] = [\![\phi \wedge \psi]\!]$$

and the upper bound corresponds to disjunction:

$$[\![\phi]\!] \cup [\![\psi]\!] = [\![\phi \vee \psi]\!].$$

Since the set of CTL-formulas is closed under conjunction and disjunction, it follows that for any ϕ and ψ their upper bound and lower bound do exist. The least element $\bot$ of the lattice $\langle \mathsf{CTL}, \sqsubseteq \rangle$ is false, since $[\![\,\mathsf{false}\,]\!] = \varnothing$, which is the bottom element for 2^S. Similarly, true is the greatest element in $\langle \mathsf{CTL}, \sqsubseteq \rangle$ since $[\![\,\mathsf{true}\,]\!] = S$.

Some CTL-axioms

In the previous chapter on linear temporal logic we have seen that axioms can be helpful in order to prove the equivalence between formulas: rather than proving the equivalence using the semantic definition, it is often sufficient to use the axioms that are defined on the syntax of the formulas. This facilitates proving

the equivalence of the formulas. An important axiom for model checking PLTL (cf. Chapter 2) is the *expansion rule* for until:

$$\phi \,\mathsf{U}\, \psi \equiv \psi \vee (\phi \wedge \mathsf{X}[\phi \,\mathsf{U}\, \psi]).$$

By instantiating this rule with the definitions of F and G one obtains

$$\begin{aligned} \mathsf{G}\,\phi &\equiv \phi \wedge \mathsf{X}\,\mathsf{G}\,\phi \text{ and} \\ \mathsf{F}\,\phi &\equiv \phi \vee \mathsf{X}\,\mathsf{F}\,\phi. \end{aligned}$$

For CTL similar axioms do exist. Given that each linear temporal operator U, F, and G can be prefixed with either an existential or a universal quantification, we obtain the axioms as listed in Table 3.3. For all these axioms the basic idea is to express the validity of a formula by a statement about the current state (without the need to use temporal operators) and a statement about the direct successors of this state (using either EX or AX depending on whether an existential or a universally quantified formula is treated). For instance, $\mathsf{EG}\,\phi$ is valid in state s if ϕ is valid in s (a statement about the current state) and ϕ holds for all states along some path starting at s (statement about the successor states). The first

$\mathsf{EG}\,\phi$	$\equiv$	$\phi \wedge \mathsf{EX}[\mathsf{EG}\,\phi]$
$\mathsf{AG}\,\phi$	$\equiv$	$\phi \wedge \mathsf{AX}[\mathsf{AG}\,\phi]$
$\mathsf{EF}\,\phi$	$\equiv$	$\phi \vee \mathsf{EX}[\mathsf{EF}\,\phi]$
$\mathsf{AF}\,\phi$	$\equiv$	$\phi \vee \mathsf{AX}[\mathsf{AF}\,\phi]$
$\mathsf{E}[\phi \,\mathsf{U}\, \psi]$	$\equiv$	$\psi \vee (\phi \wedge \mathsf{EX}[\mathsf{E}(\phi \,\mathsf{U}\, \psi)])$
$\mathsf{A}[\phi \,\mathsf{U}\, \psi]$	$\equiv$	$\psi \vee (\phi \wedge \mathsf{AX}[\mathsf{A}(\phi \,\mathsf{U}\, \psi)])$

Table 3.3: Expansion axioms for CTL

four axioms can be derived from the latter two central ones. For instance, for $\mathsf{AF}\,\phi$ we derive

$\mathsf{AF}\,\phi$

$\Leftrightarrow$ { definition of AF }

$\mathsf{A}[\mathsf{true}\,\mathsf{U}\,\phi]$

$\Leftrightarrow$ { axiom for $\mathsf{A}[\phi\,\mathsf{U}\,\psi]$ }

$\phi \vee (\mathsf{true} \wedge \mathsf{AX}[\mathsf{A}(\mathsf{true}\,\mathsf{U}\,\phi)])$

$\Leftrightarrow$ { predicate calculus; definition of AF }

$\phi \vee \mathsf{AX}[\mathsf{AF}\,\phi]$.

Using this result, we derive for $\mathsf{EG}\,\phi$:

$\mathsf{EG}\,\phi$

$\Leftrightarrow$ { definition of EG }

$\neg\,\mathsf{AF}\,\neg\,\phi$

$\Leftrightarrow$ { result of above derivation }

$\neg(\neg\,\phi \vee \mathsf{AX}[\mathsf{AF}\,\neg\,\phi])$

$\Leftrightarrow$ { predicate calculus }

$\phi \wedge \neg\mathsf{AX}[\mathsf{AF}\,\neg\,\phi]$

$\Leftrightarrow$ { definition of AX }

$\phi \wedge \mathsf{EX}(\neg[\mathsf{AF}\,\neg\,\phi])$

$\Leftrightarrow$ { definition of EG }

$\phi \wedge \mathsf{EX}[\mathsf{EG}\,\phi]$.

Similar derivations can be performed in order to find the axioms for EF and AG. These are left to the reader. As stated above, the elementary axioms are the last two expansion axioms. They can be proved using the semantics of CTL, and these proofs are very similar to the proof of the expansion law for U for PLTL as discussed in the previous chapter. We therefore omit these proofs here and leave them (again) to the reader.

CTL-formulas as fixed points

The expansion axiom

$$\mathsf{E}[\phi\,\mathsf{U}\,\psi] \equiv \psi \vee (\phi \wedge \mathsf{EX}[\mathsf{E}(\phi\,\mathsf{U}\,\psi)])$$

suggests considering the expression $\mathsf{E}[\phi\,\mathsf{U}\,\psi]$ as a fixed point of the function G : CTL $\longrightarrow$ CTL defined by

$$G(z) = \psi \vee (\phi \wedge \mathsf{EX}\,z)$$

since clearly, one obtains $G(\mathsf{E}[\phi\,\mathsf{U}\,\psi]) = \mathsf{E}[\phi\,\mathsf{U}\,\psi]$ from the above expansion rule. The functions for the other temporal operators can be determined in a similar way. In order to explain the model checking algorithms in a more elegant and compact way it is convenient to consider the set-theoretical counterpart of G (Huth and Ryan, 1999). More precisely, $[\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!]$ is a fixed point of the function $F : 2^S \longrightarrow 2^S$, where F is defined by

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\}).$$

where $R(s)$ denotes the set of direct successors of s, that is, $R(s) = \{\, s' \in S \mid (s, s') \in R \,\}$. Similar formulations can be obtained for the other temporal operators. To summarize we obtain the following fixed point characterizations:

Theorem 33.

1. $[\![\,\mathsf{EG}\,\phi\,]\!]$ *is the greatest fix-point of* $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\}$
2. $[\![\,\mathsf{AG}\,\phi\,]\!]$ *is the greatest fix-point of* $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \in S \mid \forall\, s' \in R(s) \cap Z \,\}$
3. $[\![\,\mathsf{EF}\,\phi\,]\!]$ *is the least fix-point of* $F(Z) = [\![\,\phi\,]\!] \cup \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\}$
4. $[\![\,\mathsf{AF}\,\phi\,]\!]$ *is the least fix-point of* $F(Z) = [\![\,\phi\,]\!] \cup \{\, s \in S \mid \forall\, s' \in R(s) \cap Z \,\}$

5. $[\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!]$ *is the least fix-point of*

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\})$$

6. $[\![\,\mathsf{A}[\phi\,\mathsf{U}\,\psi]\,]\!]$ *is the least fix-point of*

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cup \{\, s \in S \mid \forall\, s' \in R(s) \cap Z \,\}).$$

It is not difficult to check using the axioms of Table 3.3 that for each case the CTL-formula is indeed a fixed point of the function indicated. To determine whether it is the least or greatest fixed point is slightly more involved.

Proving the fixed point characterizations

We illustrate the proof of Theorem 33 by checking the case for $[\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!]$; the proofs for the other cases are conducted in a similar way. The proof consists of two parts: first we prove that the function $F(Z)$ is monotonic on $\langle 2^S, \subseteq\rangle$, and then we prove that $[\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!]$ is the least fixed point of F.

Proving monotonicity. Let $Z_1, Z_2 \in 2^S$ such that $Z_1 \subseteq Z_2$. It must be proven that $F(Z_1) \subseteq F(Z_2)$. Then we derive:

$$\begin{array}{ll} & F(Z_1) \\ = & \{\text{ definition of } F \,\} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z_1 \,\}) \\ \subseteq & \{\, Z_1 \subseteq Z_2 \text{; set calculus } \} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z_2 \,\}) \\ = & \{\text{ definition of } F \,\} \\ & F(Z_2). \end{array}$$

This means that for any arbitrary ψ and ϕ we have that $F(Z_1) \subseteq F(Z_2)$, and thus F is monotonic on 2^S. Given this result and the fact that $\langle 2^S, \subseteq\rangle$ is a complete lattice, it follows by Knaster-Tarski's theorem that F has a complete lattice of fixed points, including a unique least and greatest fixed point.

Proving the least fixed point. Recall that the least element of $\langle 2^S, \subseteq\rangle$ is $\varnothing$. If the set of states S has $n{+}1$ states, it is not difficult to prove that the least fixed point of F equals $F^{n+1}(\varnothing)$ for monotonic F on S. We now prove that

$$[\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!] = F^{n+1}(\varnothing) \text{ by induction on } n.$$

By definition we have $F^0(\varnothing) = \varnothing$. For $F(\varnothing)$ we derive:

$$\begin{array}{ll} & F(\varnothing) \\ = & \{\text{ definition of } F \,\} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap \varnothing \,\}) \\ = & \{\text{ calculus } \} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \varnothing) \\ = & \{\text{ calculus } \} \\ & [\![\,\psi\,]\!]. \end{array}$$

Thus $F(\varnothing) = [\![\,\psi\,]\!]$, the set of states that can reach $[\![\,\psi\,]\!]$ in 0 steps. Now

$$\begin{array}{ll} & F^2(\varnothing) \\ = & \{\text{ definition of } F \,\} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap F(\varnothing) \,\}) \\ = & \{\, F(\varnothing) = [\![\,\psi\,]\!] \,\} \\ & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap [\![\,\psi\,]\!] \,\}). \end{array}$$

Thus $F(F(\varnothing))$ is the set of states that can reach $[\![\,\psi\,]\!]$ along a path of length at most one (while traversing $[\![\,\phi\,]\!]$). By mathematical induction it can be proven that $F^{k+1}(\varnothing)$ is the set of states that can reach $[\![\,\psi\,]\!]$ along a path through $[\![\,\phi\,]\!]$ of length at most k. But since this holds for any k we have:

$$\begin{array}{ll} & [\![\,\mathsf{E}[\phi\,\mathsf{U}\,\psi]\,]\!] \\ = & \{\text{ by the above reasoning } \} \\ & \bigcup_{k \geqslant 0} F^{k+1}(\varnothing) \\ = & \{\, F^i(\varnothing) \subseteq F^{i+1}(\varnothing) \,\} \\ & F^{k+1}(\varnothing). \end{array}$$

Computing least and greatest fixed points

What do these results mean for the labelling approach to CTL model checking? We discuss this by means of an example. Consider $\chi = \mathsf{E}[\phi \,\mathsf{U}\, \psi]$. An iterative approach is taken to the computation of $Sat_{EU}(\phi, \psi)$. Since $[\![\mathsf{E}[\phi \,\mathsf{U}\, \psi]]\!]$ is the least fixed point of

$$F(Z) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \{ s \in S \mid \exists s' \in R(s) \cap Z \})$$

the problem of computing $Sat_{EU}(\phi, \psi)$ boils down to computing the fixed point of F, which is (according to Knaster-Tarski's theorem) equal to $\bigcup_i F^i(\varnothing)$. $\bigcup_i F^i(\varnothing)$ is computed by means of iteration: $\varnothing$, $F(\varnothing)$, $F(F(\varnothing)), \ldots$. Since we deal with *finite* CTL-models, this procedure is guaranteed to terminate, i.e. there exists some k such that $F^{k+1}(\varnothing) = F^k(\varnothing)$. Then, $F^{k+1}(\varnothing) = \bigcup_i F^i(\varnothing)$.

Intuitively, this iterative procedure can be understood as follows: one starts with no state being labelled with $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$. This corresponds to the approximation for which the formula is nowhere valid, i.e. $F^0(\varnothing) = \varnothing$. Then in the first iteration we consider $F^1(\varnothing)$ which reduces to $[\![\psi]\!]$ (see proof above), and label all states for which ψ holds with $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$. In the second iteration we consider $F^2(\varnothing)$ and label — in addition to the states which habe already been labelled — the states s for which ϕ holds and which have some direct successor state for which ψ holds. We continue in this way until the fixed point $F^k(\varnothing)$ is reached for some k. At this point of the computation all states are labelled that satisfy $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$.

The resulting procedures Sat_{EU} and Sat_{AU}, for the formula $\mathsf{A}[\phi \,\mathsf{U}\, \psi]$, are listed in Table 3.4 and Table 3.5.

At the beginning of the $(i{+}1)$-th iteration in these procedures we have as an invariant $Q = F^{i+1}(\varnothing)$ and $Q' = F^i(\varnothing)$ for the function F. The iterations end when $Q = Q'$, that is, when $F^{i+1}(\varnothing) = F^i(\varnothing)$. The difference between Sat_{EU} and Sat_{AU} is that for the latter a new state is labelled if ϕ holds in that state and if all successor states are already labelled, i.e. are member of Q'. This corresponds to AX.


```
function Sat_EU (φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
        Q, Q' := Sat(ψ), ∅;
        do Q ≠ Q' ⟶
            Q' := Q;
            Q := Q ∪ ({ s | ∃ s' ∈ Q. (s, s') ∈ R } ∩ Sat(φ))
        od;
        return Q
(* postcondition: Sat_EU(φ, ψ) = { s ∈ S | M, s ⊨ E[φ U ψ] } *)
end
```

Table 3.4: Labelling procedure for $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$

```
function Sat_AU (φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
        Q, Q' := Sat(ψ), ∅;
        do Q ≠ Q' ⟶
            Q' := Q;
            Q := Q ∪ ({ s | ∀ s' ∈ Q. (s, s') ∈ R } ∩ Sat(φ))
        od;
        return Q
(* postcondition: Sat_AU(φ, ψ) = { s ∈ S | M, s ⊨ A[φ U ψ] } *)
end
```

Table 3.5: Labelling procedure for $\mathsf{A}[\phi \,\mathsf{U}\, \psi]$

The iterative procedures for all formulas that have a least fixed point characterisation are performed in a similar way. For $\mathsf{EG}\,\phi$ and $\mathsf{AG}\,\phi$, which are greatest fixed points rather than least fixed points, the procedure is slightly different. Greatest fixed points are equal to $\bigcap_i F^i(S)$, which is computed by the series $S, F(S), F(F(S)), \ldots$. Intuitively, this means that one starts by labelling all states with the formula under consideration, say $\mathsf{EG}\,\phi$. This corresponds to $F^0(S) = S$. In each subsequent iteration states are "unlabelled" until a fixed point is reached.

Example 23. Let the CTL*-model $\mathcal{M}$ be shown in the first row of Figure 3.4 and suppose we are interested in checking whether $\phi = \mathsf{E}[p\,\mathsf{U}\,q]$ is valid in state s_0 of $\mathcal{M}$. For this purpose we compute $Sat_{EU}(p,q)$, i.e. $[\![\mathsf{E}[p\,\mathsf{U}\,q]]\!]$. It follows from the theory developed in this chapter that this reduces to computing the series $\varnothing$, $F(\varnothing)$, $F(F(\varnothing))$, ... until a fixed point is reached, where according to Theorem 33*

$$F^{i+1}(Z) = [\![q]\!] \cup ([\![p]\!] \cap \{\, s \in S \mid \exists s' \in R(s) \cap F^i(Z) \,\}).$$

We start the computation by:

$$F(\varnothing) = [\![q]\!] \cup ([\![p]\!] \cap \{\, s \in S \mid \exists s' \in R(s) \cap \varnothing \,\}) = [\![q]\!] = \{\, s_2 \,\}.$$

This is the situation just before starting the first iteration in $Sat_{EU}(p,q)$. This situation is illustrated in the first row, right column, where states which are in Q are colored black, and the others white.

For the second iteration we obtain

$$\begin{aligned}
& F(F(\varnothing)) \\
= \;& \{ \textit{ definition of } F \} \\
& [\![q]\!] \cup ([\![p]\!] \cap \{\, s \in S \mid \exists s' \in R(s) \cap F(\varnothing) \,\}) \\
= \;& \{\, F(\varnothing) = [\![q]\!] = \{\, s_2 \,\}; [\![p]\!] = \{\, s_0, s_1 \,\} \,\} \\
& \{\, s_2 \,\} \cup (\{\, s_0, s_1 \,\} \cap \{\, s \in S \mid \exists s' \in R(s) \cap \{\, s_2 \,\}) \,\}) \\
= \;& \{ \textit{ direct predecessors of } s_2 \textit{ are } \{\, s_1, s_3 \,\} \,\} \\
& \{\, s_2 \,\} \cup (\{\, s_0, s_1 \,\} \cap \{\, s_1, s_3 \,\}) \\
= \;& \{ \textit{ calculus} \,\}
\end{aligned}$$

$\{\, s_1, s_2 \,\}$.

The states that are now colored are those states from which a q-state (s_2) can be reached via a p-path (s_1) of length at most one.

From this result and the fact that the direct predecessors of $\{\, s_1, s_2 \,\}$ are $\{\, s_0, s_1, s_2, s_3 \,\}$ we obtain for the next iteration:

$$F^3(\varnothing) = \{\, s_2 \,\} \cup (\{\, s_0, s_1 \,\} \cap \{\, s_0, s_1, s_2, s_3 \,\}) = \{\, s_0, s_1, s_2 \,\}$$

Intuitively, the state s_0 is now labelled in the second iteration since it can reach a q-state (s_2) via a p-path (i.e. $s_0\, s_1$) of length two.

Since there are no other states in $\mathcal{M}$ that can reach a q-state via a p-path, the computation is finished. This can be checked formally by computing $F^4(\varnothing)$. The interested reader can check that indeed $F^4(\varnothing) = F^3(\varnothing)$, that is, the fixed point computation by $Sat_{EU}(p,q)$ has terminated. (End of example.)

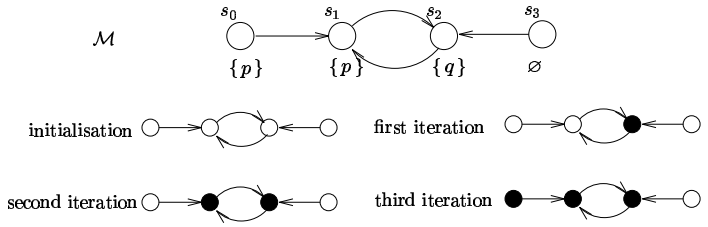


Figure 3.4: Example of iteratively computing $Sat_{EU}(p,q)$

Complexity analysis

The time complexity of model checking CTL is determined as follows. It is not difficult to see that *Sat* is computed for each sub-formula of ϕ, that is, $|\, Sub(\phi)\,|$ times. The size of $Sub(\phi)$ is proportional to the length of ϕ. The time complexity

of the labelling procedures Sat_{AU} is proportional to $|\ S_{sys}\ |^3$, because the iteration is traversed $|\ S_{sys}\ |$ times in the worst case — starting with the empty set, in each iteration a single state is added to the set Q — while for the computation of each successive Q all transitions in R have to be considered, where R in the worst case equals $S_{sys} \times S_{sys}$. This entails that the worst time complexity of model checking CTL equals $\mathcal{O}(|\ \phi\ | \times |\ S_{sys}\ |^2)$ That is, the time complexity of model checking CTL is linear in the size of the formula to be checked and cubic in the number of states of the model of the system.

This time complexity can be improved by using a different, more efficient procedure for Sat_{EG} (Clarke, Emerson and Sistla, 1986). The concept behind this variant is to define CTL in terms of EX , E$[\phi\,\mathsf{U}\,\psi]$ and EG , and to proceed for checking EG ϕ in state s as follows. First, consider only those states that satisfy ϕ and eliminate all other states and transitions. Then compute the maximal strong components in this reduced model that contain at least one transition, and check whether there is a strong component reachable from state s. If state s belongs to the reduced model and there exists such a path then — by construction of the reduced model — the property EG ϕ is satisfied; otherwise, it is not. Using this variant, the complexity of model checking CTL can be reduced to $\mathcal{O}(|\ \phi\ | \times |\ S_{sys}\ |^2)$. Thus, we conclude that

> *The worst-case time complexity of checking whether system-model sys satisfies the* CTL*-formula* ϕ *is* $\mathcal{O}(|\ S_{sys}\ |^2 \times |\ \phi\ |)$

Recall that model checking PLTL is exponential in the size of the formula. Although the difference in time complexity with respect to the length of the formula seems drastic, a few remarks on this are in order. First, formulas in PLTL are never longer than, and mostly shorter than, their equivalent formulation in CTL. This follows directly from the fact that for formulas that can be translated from CTL into PLTL, the PLTL-equivalent formula is obtained by removing all path quantifiers, and as a result is (usually) shorter (Clarke and Draghicescu, 1988). Even stronger, for each model $\mathcal{M}$ there does exist a PLTL-formula ϕ such that each CTL-formula equivalent to E ϕ (or A ϕ) — if such a formula exists in CTL — has exponential length! This is nicely illustrated by the following example, which we adopted from (Kropf, 1997).

In summary, for a requirement that can be specified in both CTL and PLTL, the shortest possible formulation in PLTL is never longer than the CTL-formula, and can even be exponentially shorter. Thus, the advantage that CTL model checking is linear in the length of the formula, whereas PLTL model checking is exponential in the length, is diminished (or even completely eliminated) by the fact that a given property needs a (much) longer formulation in CTL than in PLTL.

Example 24. We consider the problem of finding a Hamilton path in an arbitrary connected graph in terms of PLTL *and* CTL*. Consider a graph* $\mathcal{G} = (V, E)$ *where* V *denotes the set of vertices and* $E \subseteq V \times V$*, the set of edges. Suppose* $V = \{\, v_0, \ldots, v_n \,\}$*. A Hamilton path is a path through the graph which visits each state exactly once. (It is a travelling salesman problem where the cost of traversing an edge is the same for all edges.)*

We first describe the Hamilton path problem in PLTL*. The method used is to consider the graph* $\mathcal{G}$ *as a Büchi automaton which is obtained from* $\mathcal{G}$ *in the following way. We label each vertex* v_i *in* V *by a unique atomic proposition* p_i*, i.e.* $Label(v_i) = \{\, p_i \,\}$*. In addition, we introduce a new (accepting) vertex* w*, such that* $w \notin V$*, with* $Label(w) = \{\, q \,\}$*, where* q *is an atomic proposition different from any* p_i*. The vertex* w *is a direct successor of any node* v_i*, that is, the edges* (v_i, w) *are added to the graph, and* w *is a direct successor of itself, i.e.* (w, w) *is an edge. Figure 3.5 shows this construction for a connected graph with 4 vertices.*

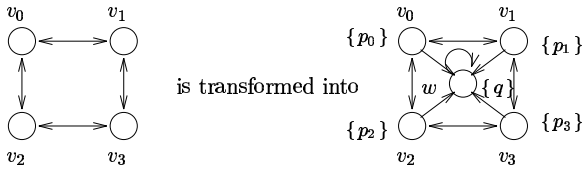


Figure 3.5: Encoding the Hamilton path problem in a model

Given this structure, the existence of a Hamilton path in a graph can be for-

mulated in PLTL *as follows*[6]*:*

$$\mathsf{E}\left[(\forall i.\,\mathsf{F}\,p_i)\ \wedge\ \mathsf{X}^{n+1}q)\right].$$

where $\mathsf{X}^1 q = \mathsf{X}q$ *and* $X^{n+1}q = \mathsf{X}(\mathsf{X}^n q)$*. This formula is valid in each state from which a path starts that fulfills each atomic proposition once. This corresponds to visiting each state* v_i *once. In order to obtain the desired infinite accepting run, such a path must have a suffix of the form* w^ω*, otherwise it would visit one or more states more than once. Notice that the length of the above formula is linear in the number of vertices in the graph.*

A formulation of the Hamilton path problem in CTL *exists and can be obtained in the following way. We start by constructing a* CTL*-formula* $g(p_0,\ldots,p_n)$ *which is valid when there exists a path from the current state that visits the states for which* $p_0 \ldots p_n$ *is valid in this order:*

$$g(p_0,\ldots,p_n) = p_0\ \wedge\ \mathsf{EX}\,(p_1\ \wedge\ \mathsf{EX}\,(\ldots\ \wedge\ \mathsf{EX}\,p_n)\ldots).$$

Because of the branching interpretation of CTL*, a formulation of the Hamilton path in* CTL *requires an explicit enumeration of all possible Hamilton paths. Let* $\mathcal{P}$ *be the set of permutations on* $\{0,\ldots n\}$*. Then we obtain:*

$$(\exists\theta\in\mathcal{P}.\,g(p_{\theta_0},\ldots,p_{\theta_n}))\ \wedge\ \mathsf{EX}^{n+1}q.$$

By the explicit enumeration of all possible permutations we obtain a formula that is exponential in the number of vertices in the graph. This does not prove that there does not exist an equivalent, but shorter, CTL*-formula which describes the Hamilton path problem, but this is impossible (Kropf, 1997). (End of example.)*

[6]Strictly speaking, this is not a well-formed PLTL-formula since the path quantifier E is not part of PLTL. However, for $\mathsf{E}\,\phi$, where ϕ does not contain any path quantifiers (as in this example) we can take the equivalent $\neg\mathsf{A}\,\neg\phi$ which can be checked by checking $\mathsf{A}\,\neg\phi$ which is a well-formed PLTL-formula according to Definition 23.

Overview of CTL model checking

We conclude this section with an overview of model checking CTL. This is shown in Figure 3.6.

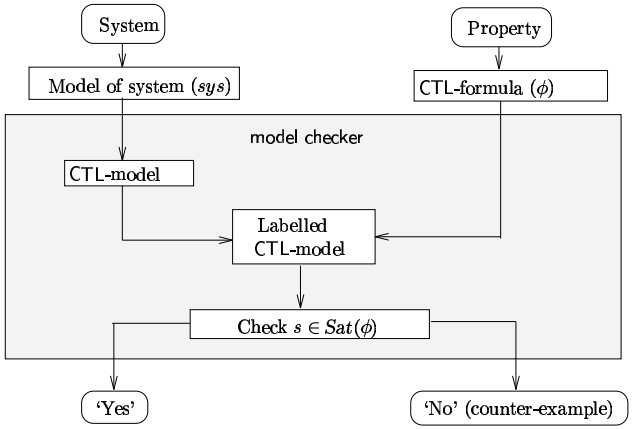


Figure 3.6: Overview of model checking CTL

3.9 Fairness

On the notion of fairness

As we have argued before, PLTL and CTL have incomparable expressiveness. An important category of properties that can be expressed in PLTL, but not in the branching temporal logic CTL, are so-called *fairness* constraints. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 25. Consider N *processes* $P_1,\ldots,P_N$ *which require a certain service. There is one server process Server which is expected to provide services to these processes. A possible strategy which Server can realize is the following. Check the*

processes starting with P_1, then P_2, and so on, and serve the first thus encountered process which requires service. On finishing serving this process, repeat this selection procedure, once again starting with checking P_1. Now suppose that P_1 is always requesting service. Then this strategy will result in Server always serving P_1. Since in this way another process has to wait infinitely long before being served, this is called an unfair *strategy. In a fair serving strategy it is required that the server eventually responds to any request by one of the processes. For instance, a round-robin scheduling strategy where each process is only served for a limited amount of time is a fair strategy: after having served one process, the next is checked and (if needed) served. (End of example.)*

In verifying concurrent systems we are quite often only interested in execution sequences in which enabled transitions (statements) are executed in some fair way. In the next section, for instance, we will treat a mutual exclusion algorithm for two processes. In order to prove the absence of individual starvation — the situation in which a process which wants to enter its critical section has to wait infinitely long — we want to exclude those execution sequences in which a single process is always being selected for execution. This type of fairness is also known as *process fairness*, since it concerns the fair scheduling of the execution of processes. If we were to consider unfair execution sequences when proving the absence of individual starvation we would usually fail, since there always exists an unfair strategy according to which some process is always neglected, and thus can never make progress.

Process fairness is a particular form of fairness. In general, fairness assumptions are needed for proving liveness properties ("something good will eventually happen") when the model to be checked considers non-determinism. In the above example the scheduling of processes is non-deterministic: the choice of the next process to be executed (if there are at least two processes which can be potentially selected) is arbitrary. Other examples where non-determinism occurs are in sequential programs, when constructs like

$$\textbf{do } \text{true} \longrightarrow S_1 [] \, \text{true} \longrightarrow S_2 \textbf{ od}$$

are allowed. Here an unfair mechanism might always choose S_1 to be executed, and as a consequence, a property that is established by executing S_2 is never

reached. Another prominent example where fairness is used to "resolve" non-determinism is in modeling concurrent processes by means of interleaving. Interleaving boils down to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed.

In general, a fair computation is characterized by the fact that certain (fairness) constraints are always fulfilled.

Types of fairness expressed in PLTL

In linear temporal logic such as PLTL, fairness can be expressed syntactically. We will briefly describe how three different forms of fairness can be formally specified in PLTL. Let ψ be the desired property (such as absence of individual starvation) and ϕ be the fairness constraint under consideration (like a process has to have its turn). Then we distinguish between

- *Unconditional fairness.* A path is unconditionally fair with respect to ψ if

  $$\mathsf{G}\,\mathsf{F}\,\psi$$

  holds. Such a property expresses, for instance, that a process enters its critical section infinitely often (regardless of any fairness constraint).

- *Weak fairness (justice).* A path is weakly fair with respect to ψ and fairness constraints ϕ if

  $$\mathsf{F}\,\mathsf{G}\,\phi \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\psi.$$

  For instance, a typical weak fairness requirement is

  $$\mathsf{F}\,\mathsf{G}\,\mathit{enabled}(a) \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\mathit{executed}(a).$$

  Weak fairness means that if an activity such as a, like a transition or an entire process, is continuously enabled ($\mathsf{F}\,\mathsf{G}\,\mathit{enabled}(a)$), then it will be executed infinitely often ($\mathsf{G}\,\mathsf{F}\,\mathit{executed}(a)$). A computation is weakly fair with

respect to activity a if it not the case that a is always enabled beyond some point without being taken beyond this point. In the literature, weak fairness is sometimes referred to as justice.

- *Strong fairness (compassion).* A path is strongly fair with respect to ψ and fairness constraints ϕ if

  $$\mathsf{GF}\,\phi \;\Rightarrow\; \mathsf{GF}\,\psi.$$

  The difference to weak fairness is that FG is replaced by GF in the premise. Strong fairness means that if an activity is infinitely often enabled (but not necessarily always, i.e. there may be periods during which ϕ is not valid), then it will be executed infinitely often. A computation is strongly fair with respect to activity a if it not the case that a is infinitely often enabled without being taken beyond a certain point.

Fairness constraints in CTL

Weak and strong fairness cannot be expressed syntactically In the branching temporal logic CTL. Additional constructs in the CTL-model are employed in order to be able to deal with fairness constraints. A CTL-model is extended such that it allows one to specify a set of fairness constraints $F_1, \ldots, F_k$. A fairness constraint is defined by (a predicate over) sets of states. For instance, in a mutual exclusion algorithm such a fairness constraint could be "process one is not in its critical section". This imposes the fairness constraint that there must be infinitely many states in a computation such that process one is not in its critical section. The basic approach is not to interpret CTL-formulas over all possible execution paths — as in the semantics of CTL which we have dealt with throughout this chapter — but rather to only consider the fair executions, i.e. those executions which satisfy all imposed fairness constraints $F_1, \ldots, F_k$. A fair CTL-model is defined as follows.

Definition 34. (**Fair model for** CTL)
A *fair* CTL-model is a quadruple $\mathcal{M} = (S, R, \mathit{Label}, \mathcal{F})$ where (S, R, Label) is a CTL-model and $\mathcal{F} \subseteq 2^S$ is a set of fairness constraints.

Definition 35. (**An $\mathcal{F}$-fair path**)
A path $\sigma = s_0\, s_1\, s_2 \ldots$ is called $\mathcal{F}$-fair if for every set of states $F_i \in \mathcal{F}$ there are infinitely many states in σ that belong to F_i.

Formally, if $\mathit{lim}(\sigma)$ denotes the set of states which are visited by σ infinitely often, then σ is $\mathcal{F}$-fair if

$$\mathit{lim}(\sigma) \cap F_i \neq \varnothing \text{ for all } i.$$

Notice that this condition is identical to the condition for accepting runs of a generalized Büchi automaton (see Chapter 2). Indeed, a fair CTL-model is an ordinary CTL-model which is extended with a generalized Büchi acceptance condition.

The semantics of CTL in terms of fair CTL-models is identical to the semantics given earlier (cf. Definition 22), except that all quantifications over paths are interpreted over $\mathcal{F}$-fair paths rather than over all paths. Let $P^f_{\mathcal{M}}(s)$ be the set of $\mathcal{F}$-fair paths in $\mathcal{M}$ which start in state s. Clearly, $P^f_{\mathcal{M}}(s) \subseteq P_{\mathcal{M}}(s)$ if $\mathcal{F} \neq \varnothing$. The fair interpretation of CTL is defined in terms of the satisfaction relation $\models_f$. $(\mathcal{M}, s) \models_f \phi$ if and only if ϕ is valid in state s of fair model $\mathcal{M}$.

Definition 36. (**Fair semantics of** CTL)
Let $p \in AP$ be an atomic proposition, $\mathcal{M} = (S, R, \mathit{Label}, \mathcal{F})$ a fair CTL-model, $s \in S$, and ϕ, ψ CTL- formulas. The satisfaction relation $\models$ is defined by:

$$\begin{array}{ll}
s \models_f p & \text{iff } p \in \mathit{Label}(s) \\
s \models_f \neg\phi & \text{iff } \neg(s \models_f \phi) \\
s \models_f \phi \vee \psi & \text{iff } (s \models_f \phi) \vee (s \models_f \psi) \\
s \models_f \mathsf{EX}\,\phi & \text{iff } \exists\, \sigma \in P^f_{\mathcal{M}}(s).\, \sigma[1] \models_f \phi \\
s \models_f \mathsf{E}[\phi\,\mathsf{U}\,\psi] & \text{iff } \exists\, \sigma \in P^f_{\mathcal{M}}(s).\, (\exists\, j \geqslant 0.\, \sigma[j] \models_f \psi \;\wedge \\
 & \qquad (\forall\, 0 \leqslant k < j.\, \sigma[k] \models_f \phi)) \\
s \models_f \mathsf{A}[\phi\,\mathsf{U}\,\psi] & \text{iff } \forall\, \sigma \in P^f_{\mathcal{M}}(s).\, (\exists\, j \geqslant 0.\, \sigma[j] \models_f \psi \;\wedge \\
 & \qquad (\forall\, 0 \leqslant k < j.\, \sigma[k] \models_f \phi)).
\end{array}$$

The clauses for the propositional logic terms are identical to the semantics

given earlier; for the temporal operators the difference lies in the quantifications which are over fair paths rather than over all paths. The expressiveness of fair CTL is strictly larger than that of CTL, and fair CTL is (like CTL) a subset of CTL*. As CTL, fair CTL is incomparable to PLTL.

A few remarks are in order to see what type of fairness (unconditional, weak or strong fairness) can be supported by using this alternative interpretation for CTL. Suppose that $\phi_1, \ldots, \phi_n$ are the desired fairness constraints and ψ is the property to be checked for CTL-model $\mathcal{M}$. Let $\mathcal{M}_f$ be equal to $\mathcal{M}$ with the exception that $\mathcal{M}_f$ is a fair CTL-model where the fairness constraints $\phi_1, \ldots, \phi_n$ are realized by appropriate acceptance sets F_j. Then

$$\mathcal{M}_f, s \models_f \psi \text{ if and only if } \mathcal{M}, s \models \mathsf{A}[(\forall i.\, \mathsf{G}\,\mathsf{F}\,\phi_i) \Rightarrow \psi]$$

(Notice that $\mathsf{A}[(\forall i.\, \mathsf{G}\,\mathsf{F}\,\phi_i) \Rightarrow \psi]$ is a CTL*-formula, and not a CTL-formula.) The intuition of this result is that the formulas $\mathsf{G}\,\mathsf{F}\,\phi_i$ exactly characterize those paths which visit F_j infinitely often.

For example, strong fairness (compassion) can now be imposed by checking

$$s \models_f \mathsf{AG}\,\mathsf{AF}\,\psi$$

where we use the result that the PLTL-formula $\mathsf{G}\,\mathsf{F}\,\psi$ is equivalent to the CTL-formula $\mathsf{AG}\,\mathsf{AF}\,\psi$.

Example 26. Consider the CTL*-model depicted in Figure 3.7 and suppose we are interested in checking* $\mathcal{M}, s_0 \models \mathsf{AG}[p \Rightarrow \mathsf{AF}\,q]$. *This property is invalid since there is a path* $s_0\, s_1\, (s_2\, s_4)^\omega$ *which never goes through a q-state. The reason for that this property is not valid is that at state* s_2 *there is a non-deterministic choice between moving either to* s_3 *or to* s_4*, and by always ignoring the possibility of going to* s_3 *we obtain a computation for which* $\mathsf{AG}[p \Rightarrow \mathsf{AF}\,q]$ *is invalid:*

$$\mathcal{M}, s_0 \not\models \mathsf{AG}[p \Rightarrow \mathsf{AF}\,q].$$

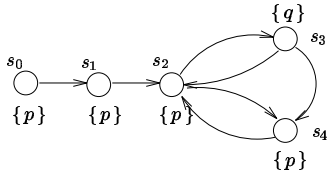


Figure 3.7: An example CTL-model

Usually, though, the intuition is that if there is infinitely often a choice of moving to s_3 *then* s_3 *should be visited sometime (or infinitely often).*

We transform the CTL*-model shown in Figure 3.7 into a fair* CTL*-model* $\mathcal{M}'$ *by defining* $\mathcal{F} = \{ F_1, F_2 \}$ *where* $F_1 = \{ s_3 \}$ *and* $F_2 = \{ s_4 \}$. *Let us now check* $\mathsf{AG}[p \Rightarrow \mathsf{AF}\,q]$ *on this fair model, that is, consider* $\mathcal{M}', s_0 \models_f \mathsf{AG}[p \Rightarrow \mathsf{AF}\,q]$. *Any* $\mathcal{F}$*-fair path starting at* s_0 *has to go infinitely often through some state in* F_1 *and some state in* F_2. *This means that states* s_3 *and* s_4 *must be visited infinitely often. Such fair paths exclude paths like* $s_0\, s_1\, (s_2\, s_4)^\omega$, *since* s_3 *is never visited along this path. Thus we deduce that indeed*

$$\mathcal{M}', s_0 \models_f \mathsf{AG}[p \Rightarrow \mathsf{AF}\,q].$$

(End of example.)

Model checking fair CTL

It is beyond the scope of these lecture notes to treat in detail the extensions to the model checking algorithms which we have presented earlier which are concerned with the treatment of fairness. The fixed point characterizations of EF , EG and so on can be extended with fairness constraints. Based on these characterizations model checking of fair CTL can be performed in a similar way as model checking of CTL. The most important changes to the algorithms are that we have to consider the fair paths from a state, rather than all paths (like we did up to so far) for the procedures Sat_{EU} and Sat_{AU} and the treatment of $\mathsf{EX}\,\phi$. For this purpose, we need an algorithm to compute fair paths. In essence, fair paths can be determined

as follows. Suppose we want to determine whether there is an $\mathcal{F}$-fair path in $\mathcal{M}$ starting at state s. Then:

1. We check whether we can reach from s some state $s' \in F_i$ for the fairness constraint $F_i \in \mathcal{F}$;
2. If such state exists for some i, we check whether we can reach s' from itself via a path which goes through a state in F_j for each $F_j \in \mathcal{F}$.

If both checks are positive then we have found a cycle which is reachable from the state s such that in the cycle each fairness constraint is satisfied. As for the calculation of emptiness for Büchi automata in the previous chapter, this strategy is the same as the computation of maximal strong components in a graph (cf. the programs *ReachAccept* and *DetectCycle* of Chapter 2). The standard algorithm to compute maximal strong components by Tarjan (1972) has a known worst case time-complexity of $\mathcal{O}(\mid S \mid + \mid R \mid)$.

A second major change to the model checking algorithms discussed before are the fixed point characterizations of $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$ and $\mathsf{A}[\phi \,\mathsf{U}\, \psi]$. These have to be changed in order to incorporate the fairness constraints $\mathcal{F}$. We do not provide the details here of this adaptation (see e.g. Clarke, Grumberg and Long, 1993) and only mention that the worst-case time-complexity of the model checking algorithms grows from $\mathcal{O}(\mid \phi \mid \times \mid S \mid^3)$ to

$$\mathcal{O}(\mid \phi \mid \times \mid S \mid^3 \times \mid \mathcal{F} \mid).$$

As for model checking plain CTL, this time complexity can be improved to being quadratic in the size of the state space, by using an alternative approach for EG.

3.10 The model checker SMV

The tool SMV (Symbolic Model Verifier) supports the verification of cooperating processes which "communicate" via shared variables. The tool has been

developed at Carnegie-Mellon University by Ken McMillan and was originally developed for the automatic verification of synchronous hardware circuits. It is publically available, see `www.cs.cmu.edu/~modelcheck`. The model checker has been very useful for verifying hardware circuits and communication protocols. It has recently also been applied to large software systems, for instance, in the airplane-industry (Chan et. al, 1998). Processes can be executed in either a synchronous or an asynchronous fashion and model checking of CTL-formulas is supported. The model checking algorithms which are used by SMV are basically the algorithms which we have covered in this chapter and support the treatment of fair CTL. It is not our intention to give a complete introduction to the tool SMV in these lecture notes. For a more extensive introduction we refer to (McMillan, 1992/1993). We rather want to demonstrate by example how systems can be specified and verified using the model checker.

An SMV specification consists of process declarations, (local and global) variable declarations, formula declarations and a specification of the formulas that are to be verified. The main module is called `main`, as in C. The global structure of an SMV specification is:

```
MODULE main
VAR variable declarations
ASSIGN global variable assignments
DEFINITION definition of property to be verified   /* optional */
SPEC CTL-specification to verify

MODULE    /* submodule 1 */

MODULE    /* submodule 2 */

.............
```

The main ingredients of system specifications in SMV are:

- *Data types.* The only basic data types provided by SMV are bounded integer subranges and symbolic enumerated types.

- *Process declarations and initializations.* A process named `P` is defined as follows:

  ```
  MODULE P (formal parameters)
  VAR local definitions
  ASSIGN initial assignments to variables
  ASSIGN next assignments to variables
  ```

  This construct defines a module called `P` which can be instantiated by either

  ```
  VAR Pasync: process P(actual parameters)
  ```

  to obtain a process instantiation called `Pasync` which executes in asynchronous mode, or by

  ```
  VAR Psync: P(actual parameters)
  ```

  to obtain a process instantiation `Psync` which executes in synchronous mode. (The difference between asynchronous and synchronous is discussed below.)

- *Variable assignments.* In SMV a process is regarded as a finite-state automaton and is defined by listing for each (local and global) variable the initial value (i.e. the values in the initial state), and the value which is to be assigned to the variable in the next state. The latter value usually depends on the current values of the variables. For instance, `next(x) := x+y+2` assigns to `x` the value `x+y+2` in the next state. For variable `x` the assignment `init(x) := 3` denotes that `x` initially has the value 3. The assignment `next(x) := 0` assigns to `x` the value 0 in the next state. Assignments can be non-deterministic, e.g. the statement `next(x) := {0, 1}` means that the next value of `x` is either 0 or 1. Assignments can be conditional. For instance, the assignment

  ```
  next(x) := case b = 0: 2;
                  b = 1: {7, 12}
             esac;
  ```


  assigns to variable `x` the value 2 if `b` equals 0 and (non-deterministically) the value 7 or 12 if `b` equals 1. If `x` is a variable in process instantiation `Q` then we write `Q.x`.

  Assignments to global variables are only allowed if these variables occur as parameters of the process instantiation.

- *Synchronous versus asynchronous mode.* In the synchronous mode all assignments in any process are carried out in a single indivisible step. Intuitively this means that there is a single global clock and at each of its ticks each module performs a step. At any given time, a process is either running or not running. An assignment to the `next` value of a variable only occurs if the process is running. If not, the value of the variable remains unchanged in the next step.

  In asynchronous mode only the variables of the "active" process are assigned a new value. That is, at each tick of the global clock a single process is non-deterministically chosen for execution and a single step of this process is performed (while the other, not selected processes, keep their state). Synchronous composition is useful for e.g. modelling synchronous hardware circuits, whereas asynchronous composition is useful for modeling communication protocols or asynchronous hardware systems.

- *Specification of* CTL*-formulas.* For the specification of CTL-formulas, SMV uses the symbols `&` for conjunction, `|` for disjunction, `->` for implication and `!` for negation. The SMV model checker verifies that all possible initial states satisfy the specification.

Peterson and Fischer's mutual exclusion algorithm

The mutual exclusion algorithm of Peterson and Fischer is used by two processes which communicate via shared variables. It is intended to prevent the processes being simultaneously in their critical section. Each process i (i=1,2) has local variables t_i and y_i which are readable by the other process but which can only be written by process i. These variables range over $\{\bot, \text{false}, \text{true}\}$. The operator $\neg$ is not defined for $\bot$ and has its usual meaning for false and true. This means that in an expression like $\neg y_2 = y_1$ it is assumed that $y_1 \neq \bot$ and $y_2 \neq \bot$.

The initial values of the variables are $y_1, y_2, t_1, t_2 := \bot, \bot, \bot, \bot$. Process 1 is as follows:

$start_1$: (* Process 1 *)
$t_1 :=$ **if** $y_2 =$ false **then** false **else** true **fi**
$y_1 := t_1$
if $y_2 \neq \bot$ **then** $t_1 := y_2$ **fi**
$y_1 := t_1$
loop while $y_1 = y_2$
critical section 1; $y_1, t_1 := \bot, \bot$
goto $start_1$

Process 2 is as follows:

$start_2$: (* Process 2 *)
$t_2 :=$ **if** $y_1 =$ true **then** false **else** true **fi**
$y_2 := t_2$
if $y_1 \neq \bot$ **then** $t_2 := \neg y_2$ **fi**
$y_2 := t_2$
loop while $(\neg y_2) = y_1$
critical section 2; $y_2, t_2 := \bot, \bot$
goto $start_2$

The basic method of the algorithm is that when both processes compete to enter their critical section, it is guaranteed that $y_1 \neq \bot$ and $y_2 \neq \bot$, and therefore the conditions for entering the critical sections can never be valid simultaneously. Notice that the two processes are not symmetric.

Modeling the mutual exclusion algorithm in SMV

The translation of the algorithm just described into an SMV specification is fairly straightforward. The two process instantiations are created by


```
VAR prc1 : process P(t1,t2,y1,y2);
    prc2 : process Q(t1,t2,y1,y2);
```

The behavioral part of the first process of the previous section is given below. For our purpose we have labelled each statement in the algorithm starting from label `l1`. These labels are used to determine the next statement after the execution of a statement; they are a kind of program counter. The SMV code for the other process is rather similar, and is omitted here. Comments start with `--` and end with a carriage return.

```
MODULE P(t1,t2,y1,y2)
VAR label : {l1,l2,l3,l4,l5,l6,l7}; -- local variable label
ASSIGN init(label) := l1;           -- initial assignment
ASSIGN                              -- next assignments
   next(label) :=
       case
         label = l1              : l2;
         label = l2              : l3;
         label = l3              : l4;
         label = l4              : l5;
         label = l5 & y1 = y2    : l5;  -- loop
         label = l5 & !(y1 = y2) : l6;
         label = l6              : l7;
         label = l7              : l1;  -- goto start
       esac;

   next(t1) :=
       case
         label = l1 & y2 = false     : false;
         label = l1 & !(y2 = false)  : true;
         label = l3 & y2 = bottom    : t1;
         label = l3 & !(y2 = bottom) : y2;
         label = l6                  : bottom;
         1                           : t1;  -- otherwise keep
                                            -- t1 unchanged
```

```
    esac;

next(y1) :=
    case
      label = l2 | label = l4 : t1;
      label = l6              : bottom;
      1                       : y1;  -- otherwise keep
                                     -- y1 unchanged
    esac;
```

Notice that the label `prc1.l6` corresponds to the critical sections of process 1 and symmetrically, label `prc2.m6` corresponds to the critical sections of process 2.

Model checking the SMV specification

There are two major properties which a mutual exclusion algorithm must possess. First there must be at most one process in its critical section at the same time. This is expressed in CTL as:

$$\mathsf{AG}\ \neg(prc1.label = l6\ \wedge\ prc2.label = m6)$$

In SMV this property is defined by

```
DEFINE MUTEX := AG !(prc1.label = l6 & prc2.label = m6)
```

where `MUTEX` is simply a macro. The result of the verification using `SPEC MUTEX` is positive:

```
-- specification MUTEX is true

resources used:
```



```
user time: 1.68333 s, system time: 0.533333 s
BDD nodes allocated: 12093
Bytes allocated: 1048576
BDD nodes representing transition relation: 568 + 1
reachable states: 157 (2^7.29462) out of 3969 (2^11.9546)
```

For the moment we ignore the messages concerning BDDs. SMV uses BDDs to represent (and store) the state space in a compact way in order to improve the capabilities of the model checker. These techniques will be treated in Chapter 5 of these lecture notes.

A second important property of a mutual exclusion algorithm is the absence of individual starvation. That means that it should not be possible that a process which wants to enter its critical section can never do so. This can be formalized in the current example in the following way

```
NST := AG ((prc1.label in {l1,l2,l3,l4,l5} -> AF prc1.label = l6) &
           (prc2.label in {m1,m2,m3,m4,m5} -> AF prc2.label = m6))
```

An alternative formulation of the intended property is

```
AG AF !(prc1.label in {l1,l2,l3,l4,l5}) &
AG AF !(prc2.label in {m1,m2,m3,m4,m5})
```

This states that from any state in the computation along any path the label of process 1 (and 2) will be different eventually from `l1` through `l5` (and `m1` through `m5`). This implies that process 1 and 2 are "regularly" in their critical section.

Checking the property `NST` using SMV yields an error as indicated by the following generated output:

```
-- specification NST is false
-- as demonstrated by the following execution sequence
-- loop starts here --
```

```
state 1.1:
NST = 0
MUTEX = 1
t1 = bottom
t2 = bottom
y1 = bottom
y2 = bottom
prc1.label = l1
prc2.label = m1

state 1.2:
[executing process prc2]

state 1.3:
[executing process prc2]
t2 = true
prc2.label = m2

state 1.4:
[executing process prc2]
y2 = true
prc2.label = m3

state 1.5:
[executing process prc2]
prc2.label = m4

state 1.6:
[executing process prc2]
prc2.label = m5

state 1.7:
[executing process prc2]
prc2.label = m6

state 1.8:
```



```
[executing process prc2]
t2 = bottom
y2 = bottom
prc2.label = m7

state 1.9:
prc2.label = m1
```

The counter-example is described as a sequence of changes to variables. If a variable is not mentioned, then its value has not been changed. The counterexample which the model checker produces indicates that there exists a loop in which `prc2` repeatedly obtains access to its critical section, whereas process `prc1` remains waiting forever. This should not surprise the reader, since the SMV system can, in each step of its execution, non-deterministically select a process (which can proceed) for execution. By always selecting one and the same process, we obtain the behavior illustrated above. One might argue that this result is not of much interest, since this unfair scheduling of processes is undesired. Stated differently, we would like to have a scheduling mechanism in which enabled processes — a process which can make a transition — are scheduled in a *fair* way. How this can be done is explained in the following subsection.

Model checking with fairness using SMV

SMV offers the possibility of specifying unconditional fairness requirements by stating (for *each* process):

```
FAIRNESS f
```

where `f` is an arbitrary CTL-formula. The interpretation of this statement is that the model checker, when checking some specification (which might differ from `f`), will ignore any path along which `f` does not occur infinitely often. Using this construct, we can establish process fairness in the following way. In SMV each process has a special variable `running` which is true if and only if the process is currently executing. When we add `FAIRNESS running` to our previous

SMV specification for the mutual exclusion program, the previously obtained execution that showed the presence of individual starvation is indeed no longer possible (since it is unfair). We now obtain indeed absence of individual starvation. Therefore we can conclude that under the fairness assumption that each process is executed infinitely often, the algorithm of Peterson and Fischer does not lead to individual starvation.

As another example of the use of the `FAIRNESS` construct, let us modify the mutual exclusion algorithm slightly by allowing a process to stay in its critical section for some longer time. Recall that the label `prc1.l6` corresponds to process one being in its critical section. We now change the code of `MODULE P` so that we obtain:

```
ASSIGN
   next(label) :=
     case
     .........
     label = l6 : {l6, l7};
     .........
     esac;
```

Process one can now stay in its critical section infinitely long, while blocking process two from making any progress. Notice that adding `FAIRNESS running` to the above specification does not help: process two will then be scheduled for execution, but since it is blocked, it cannot make any progress. In order to avoid this situation we add the fairness constraint:

```
FAIRNESS  !(prc1.label = l6)
```

This statement means that when evaluating the validity of CTL-formulas, the path operators A and E range over the fair paths with respect to `!(prc1.label = l6)`. Stated differently, the runs for which `!(prc1.label = l6)` is not valid infinitely often are not considered. This indeed avoids that process one remains in its critical section forever.

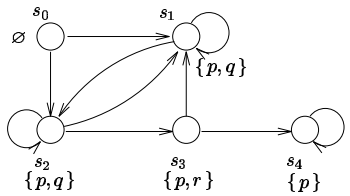
Exercises

EXERCISE 16. Prove the following equality:

$$\mathsf{A}[p\,\mathsf{U}\,q] \equiv \neg\,(\mathsf{E}[\neg q\,\mathsf{U}\,(\neg p\ \wedge\ \neg q)]\ \vee\ \mathsf{EG}\,\neg p)$$

where p and q are atomic propositions.

EXERCISE 17. Assume that we have the following model:



where states are represented by circles, for each state s the labelling $\pi(s)$ is given beside the circle representing s, and arrows denote relation R such that there is an arrow from s to s' if and only if $(s,s') \in R$. (a) Show that this model is a CTL-model and (b) check for each state in this model the validity of the following CTL-formulas in an informal way (that is, using the strategy of Example 21. Justify your answers.

1. $\mathsf{EG}\,p$
2. $\mathsf{AG}\,p$
3. $\mathsf{EF}\,[\mathsf{AG}\,p]$
4. $\mathsf{AF}\,[p\,\mathsf{U}\,\mathsf{EG}\,(p\ \Rightarrow\ q)]$
5. $\mathsf{EG}\,[((p\ \wedge\ q)\ \vee\ r)\,\mathsf{U}\,(r\,\mathsf{U}\,\mathsf{AG}\,p)]$

EXERCISE 18. We have defined in this chapter the logic CTL in terms of the basic operators EX, EU and AU. (EU and AU denote the combination of a path quantifier

with the "until" operator.) Another possibility is to consider the operators AG, AX and AU as basic and to define all other operators (EX, EU and EG) as derivatives of these operators. Give a translation from an arbitrary CTL-formula into the basic operators AG, AX and AU.

EXERCISE 19. Give an example CTL-model which shows that the PLTL-formula $\mathsf{A}[\mathsf{F}\,\mathsf{G}\,p]$ (expressed as embedded CTL*-formula) and the CTL-formula $\mathsf{AF}\,\mathsf{AG}\,p$ are distinct.

EXERCISE 20.

1. Prove that the function $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \mid \exists\, s' \in R(s) \cap Z \,\}$ is monotonic on the complete lattice $\langle 2^S, \subseteq \rangle$.
2. Given that $[\![\,\mathsf{EG}\,\phi\,]\!]$ is the greatest fixed point of the function $F(Z)$, give an algorithm Sat_{EG} which labels states in an arbitrary CTL-model $\mathcal{M} = (S, R, \pi)$ such that the postcondition of this procedure equals:

$$Sat_{EG}(\phi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{EG}\,\phi \,\}.$$

EXERCISE 21. Let $\mathcal{M} = (S, P, \pi)$, where S is a set of states, P is a set of paths (where a path is an infinite sequence of states) and π is an assignment of atomic propositions to states. If we impose the following conditions on P, then (S, P, π) can be used as a CTL-model:

I $\sigma \in P \;\Rightarrow\; \sigma_{(1)} \in P$.

II $(\rho\, s\, \sigma \in P \;\wedge\; \rho'\, s\, \sigma' \in P) \;\Rightarrow\; \rho\, s\, \sigma' \in P$.

Here σ, σ' are paths, $\sigma_{(1)}$ is σ where the first element of σ is removed, and ρ, ρ' are finite sequences of states.

1. Give an intuitive interpretation of **I** and **II**.
2. Check whether the following sets of paths satisfy **I** and **II**. Here a^ω denotes an infinite sequence of a's, and a^* denotes a finite (possibly empty) sequence of a's. Justify your answers.

(a) $\{\, a\,b\,c^\omega, d\,b\,e^\omega \,\}$

(b) $\{\, a\,b\,c^\omega, d\,b\,e^\omega, b\,c^\omega, c^\omega, b\,e^\omega, e^\omega \,\}$

(c) $\{\, a\,a^*\,b^\omega \,\}$.

EXERCISE 22. Consider the CTL-model of Exercise 17 and compute the sets $[\![\,\mathsf{EF}\,p\,]\!]$ and $[\![\,\mathsf{EF}\,[\mathsf{AG}\,p]\,]\!]$ using the fixed point characterizations.

EXERCISE 23. Consider the mutual exclusion algorithm of the Dutch mathematician Dekker. There are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 whose initial values are false, and a variable k which can take the values 1 and 2 and whose initial value is arbitrary. The i-th process (i=1,2) is described as follows, where j is the index of the other process:

```
while true do
begin b_i := true;
      while b_j do
            if k = j then begin
                              b_i := false;
                              while k = j do skip;
                              b_i := true
                              end;
      ⟨ critical section ⟩;
      k := j;
      b_i := false
end
```

Model this algorithm in SMV, and investigate whether this algorithm satisfies the following properties:

Mutual exclusion: two processes cannot be in their critical section at the same time.

Individual starvation: if a process wants to enter its critical section, it is eventually able to do so.

Hint: use the `FAIRNESS running` statement in your SMV specification to prove these properties in order to prohibit unfair executions (which might trivially violate these requirements).

EXERCISE 24. Dekker's algorithm is restricted to two processes. In the original mutual exclusion algorithm of Dijkstra in 1965, another Dutch mathematician, it is assumed that there are $n \geqslant 2$ processes, and global variables b, c : **array** $[1 \ldots n]$ of **boolean** and an integer k. Initially all elements of b and of c have the value true and the value of k is non-deterministically chosen from the range $1, 2, \ldots, n$. The i-th process may be represented as follows:

```
var j : integer;
while true do
begin b[i] := false;
      L_i : if k ≠ i then begin c[i] := true;
                                if b[k] then k := i;
                                goto L_i
                          end;
            else begin c[i] := false;
                       for j := 1 to n do
                       if (j ≠ i ∧ ¬(c[j])) then goto L_i
                 end
      ⟨ critical section ⟩;
      c[i] := true;
      b[i] := true
end
```

Questions:

(a) Model this algorithm in SMV and check the mutual exclusion and individual starvation property. In case a property is not satisfied analyze the cause for this invalidity by checking the generated counter-example.

(b) Start with n=2, increase the number of processes gradually and compare the sizes of the state spaces.

EXERCISE 25. It is well-known that Dijkstra's solution for N processes is unfair, i.e. individual starvation is possible. In order to find a fair solution for N processes, Peterson proposed in 1981 the following variant. Let $Q[1 \ldots N]$ and $T[1 \ldots N-1]$ (T for Turn), be two shared arrays which are initially 0 and 1, respectively. The variables i and j are local to the process with i containing the process number. The code of process i is as follows:

```
for j := 1 to N − 1 do
begin
      Q[i] := j;
      T[j] := i;
      wait until (∀k ≠ i. Q[k] < j) ∨ T[j] ≠ i
end;
⟨ critical Section ⟩;
Q[i] := 0
```

Check whether this is indeed a mutual exclusion program for $N = 2, 3, 4, \ldots$ (until the generated state space is too large to handle), and check whether Peterson's algorithm is indeed free from individual starvation.

3.11 Selected references

Kripke structures:

- S.A. KRIPKE. Semantical considerations on modal logic. *Acta Philosophica Fennica* **16**: 83–94, 1963.

Branching temporal logic:

- M. BEN-ARI, Z. MANNA, A. PNUELI. The temporal logic of branching time. *Acta Informatica* **20**: 207–226, 1983.
- E.A. EMERSON AND E.M. CLARKE. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming* **2**: 241–266, 1982.

- D. KOZEN. Results on the propositional μ-calculus. *Theoretical Computer Science* **27**: 333–354, 1983.

Branching temporal logic versus linear temporal logic:

- E.A. EMERSON AND J.Y. HALPERN. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM* **33**(1): 151–178, 1986.
- E.M. CLARKE AND I.A. DRAGHICESCU. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 428–437, 1988.
- T. KROPF. *Hardware Verifikation.* Habilitation thesis. University of Karlsruhe, 1997. (in German).

Model checking branching temporal logic using fixed point theory:

- E.M. CLARKE, E.A. EMERSON, A.P. SISTLA. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* **8**(2): 244–263, 1986.
- E.M. CLARKE, O. GRUMBERG, D. LONG. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency–Reflections and Perspectives*, LNCS 803, pages 124–175, 1993.
- M. HUTH AND M.D. RYAN. *Logic in Computer Science – Modelling and Reasoning about Systems.* Cambridge University Press, 1999 (to appear).

Fixed point theory:

- Z. MANNA, S. NESS, J. VUILLEMIN. Inductive methods for proving properties of programs. *Communications of the ACM* **16**(8): 491–502, 1973.

Model checking branching temporal logic using tree automata:

- O. BERNHOLTZ, M.Y. VARDI, P. WOLPER. An automata-theoretic approach to branching-time model checking (extended abstract). In *Computer Aided Verification*, LNCS 818, pages 142–156, 1994.

Fairness:

- N. FRANCEZ. *Fairness.* Springer-Verlag, 1986.

SMV:

- K.L. MCMILLAN. The SMV System. Technical Report CS-92-131, Carnegie-Mellon University, 1992.
- K.L. MCMILLAN. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

Large case study with SMV:

- W. CHAN, R.J. ANDERSON, P. BEAME, S. BURNS, F. MODUGNO, D. NOTKIN AND J.D. REESE. Model checking large software specifications. *IEEE Transactions on Software Engineering* **24**(7): 498–519, 1998.