Huth & Ryan
Logic in Computer Science 3
Cambridge 2000 Verification by model checking
Pages 148 - 181

3.1 Motivation for verification

There is a great advantage in being able to verify the correctness of computer systems (whether they are hardware, software, or a combination). This is most obvious in the case of *safety-critical systems*, but also applies to those that are *commercially critical*, such as mass-produced chips, *mission critical*, etc. Formal verification methods have quite recently become usable by industry and there is a growing demand for professionals able to apply them (witness recent job adverts by BT, Intel, National Semiconductor Corp, etc.). In this chapter, and the next one, we examine two applications of logics to the question of verifying the correctness of computer systems, or programs.

(Formal) verification techniques can be thought of as comprising three parts:

- A *framework for modelling systems*, typically a description language of some sort;
- A *specification language* for describing the properties to be verified;
- A *verification method* to establish whether the description of a system satisfies the specification.

Approaches to verification can be classified according to the following criteria:

Proof-based vs. model-based. In a proof-based approach, the system description is a set of formulas Γ (in a suitable logic) and the specification is another formula ϕ. The verification method consists of trying to find a proof that $\Gamma \vdash \phi$. This typically requires guidance and expertise from the user.

In a model-based approach, the system is represented by a finite

model $\mathscr{M}$ for an appropriate logic. The specification is again represented by a formula ϕ and the verification method consists of computing whether a model $\mathscr{M}$ satisfies ϕ ($\mathscr{M} \models \phi$). This is usually automatic, though the restriction to finite models limits the applicability.

In Chapters 1 and 2, we could see that logical proof systems are often sound and complete, meaning that $\Gamma \vdash \phi$ (provability) iff $\Gamma \models \phi$ (semantic entailment), where the latter is defined as follows: for all models $\mathscr{M}$, if $\mathscr{M} \models \Gamma$, then $\mathscr{M} \models \phi$. Thus, we see that the model-based approach is potentially simpler than the proof-based approach, for it is based on a single model $\mathscr{M}$ rather than a possibly infinite class of them.

Degree of automation. Approaches differ on how automatic the method is. Extremes are fully automatic and fully manual, with many computer-assisted techniques somewhere in the middle.

Full- vs. property-verification. The specification may describe a single property of the system, or it may describe its full behaviour. The latter is typically expensive to verify.

Intended domain of application, which may be hardware or software; sequential or concurrent; reactive or terminating; etc. A reactive system is one which reacts to its environment and is not meant to terminate (e.g. operating systems, embedded systems and computer hardware).

Pre- vs. post-development. Verification is of greater advantage if introduced early in the course of system development, because errors caught earlier on in the production cycle are less costly to rectify. (Apparently, Intel lost millions of dollars by releasing their Pentium chip with the FDIV error.)

This chapter concerns a verification method called *model checking*. In the terms of the above classification, model checking is an automatic, model-based, property-verification approach. It is intended to be used for *concurrent, reactive* systems and originated as a post-development methodology. Concurrency bugs are among the most difficult to find by *testing* (the activity of running several simulations of important scenarios), since they tend to be non-reproduceable, so it is well worth having a verification technique that can help one to find them.

By contrast, Chapter 4 describes a very different verification technique which in terms of the above classification is a proof-based, computer-assisted, property-verification approach. It is intended to be used for programs which we expect to terminate and produce a result.

Model checking is based on temporal logic. The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. Thus, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state. In model checking, the models $\mathcal{M}$ are *transition systems* and the properties ϕ are formulas in temporal logic. To verify that a system satisfies a property, we must do three things:

- Model the system using the description language of a model checker, arriving at a model $\mathcal{M}$.
- Code the property using the specification language of the model checker, resulting in a temporal logic formula ϕ.
- Run the model checker with inputs $\mathcal{M}$ and ϕ.

The model checker outputs the answer 'yes' if $\mathcal{M} \models \phi$ and 'no' otherwise; in the latter case, most model checkers also produce a trace of system behaviour which causes this failure. This automatic generation of such 'counter traces' is an important tool in the design and debugging of systems.

Since model checking is a *model-based* approach, in terms of the classification given earlier, it follows that in this chapter, unlike in the previous two, we will not be concerned with semantic entailment ($\Gamma \models \phi$), or with proof theory ($\Gamma \vdash \phi$), such as the development of a natural deduction calculus for temporal logic. We will work solely with the notion of satisfaction, i.e. the satisfaction relation between a model and a formula ($\mathcal{M} \models \phi$).

There is a whole zoo of temporal logics that people have proposed and used for various things. The abundance of such formalisms may be organised by classifying them according to their particular view of 'time'. *Linear-time* logics think of time as a chain of time instances, *branching-time* logics offer several alternative future worlds at any given point of time; the latter is most useful in modelling non-deterministic systems or computations. Another quality of 'time' is whether we think of it as being *continuous* or *discrete*. The former would be suggested if we study an analogue computer, the latter might be preferred for a synchronous network.

In this chapter, we study a logic where 'time' is branching and discrete. Such a logic has a modal aspect to it since the possible future paths of computations allow us to speak of 'possibilities' and 'necessities'; the study of such modalities is the focus of Chapter 5. The logic we now study is called *computation tree logic* (CTL), due to E. Clarke and E. A. Emerson. This logic has proven to be extremely fruitful in verifying hardware and

communication protocols; and people are beginning to apply it even to the verification of software. *Model checking* is the process of computing an answer to the question of whether $\mathcal{M}, s \models \phi$ holds, where ϕ is a formula of some logic, $\mathcal{M}$ an appropriate model, s a state of that model and $\models$ the underlying satisfaction relation. Of course, ϕ is now a formula of CTL, which we define in a little while, and the model $\mathcal{M}$ is a representation of our 'system'.

You should not confuse such models $\mathcal{M}$ with an actual physical system. Models are abstractions that omit lots of real features of a physical system, which are irrelevant to the checking of ϕ. This is similar to the abstractions that one does in calculus or mechanics. There we talk about *straight* lines, *perfect* circles, or an experiment without *friction*. These abstractions are very powerful, for they allow us to focus on the essentials of our particular concern.

Models of the kind we have in mind are very general indeed. That has the advantage that we may think of a wide spectrum of structures in computer science as having the same *type* of model. So this allows us to develop a unified approach of model checking for the verification of hardware, networks, software, etc. The fundamental constituents of such models are *states*. They could be the current values of variables in a C-like programming language, or actual states of physical devices in a network architecture, for example, the states 'busy' and 'available' of some computing resource. The second kind of constituents of these models expresses the dynamics of the underlying computational process. Such dynamical behaviour is captured in terms of *state transitions*. Thus, one may think of an assignment statement

$$x := x + 1;$$

as a state transition from state s to state s', where the latter state stores the same values for variables as s, but for x, in which it stores a value one bigger than that of s. You may think of the set of possible state transitions in a variety of ways, but it is best for our purposes to think of them in their entirety as a binary relation $\rightarrow$ on the set of states S:

$$\rightarrow \subseteq S \times S.$$

We write $s \rightarrow s'$ to express that it is possible for the system to reach state s' from state s *in one computation step*.

The expressive power of such models stems from the fact that there is generally more than one possible successor state from a given state s. For example, if we have a language with *parallel* assignment statements running on a single processor, then we expect to encounter multiple future

computation paths depending on the actual scheduler of this parallel activity. See Figure 3.2 (later) for a pictorial representation of such a model.

3.2 The syntax of computation tree logic

Computation tree logic, or CTL for short, is a temporal logic, having connectives that allow us to refer to the future. It is also a *branching-time* logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the 'actual' path that is realised.

We work with a fixed set of atomic formulas/descriptions (such as $p, q, r, \ldots$, or $p_1, p_2, \ldots$). These atoms stand for atomic descriptions of a system, like

The printer is busy.

or

There are currently no requested jobs for the printer.

or

The current content of register R1 *is the integer value* 6.

and the choice of atomic descriptions obviously depends on our particular interest in a system at hand.

Definition 3.1 We define CTL formulas inductively via a Backus Naur form (as done for the other logics of Chapters 1 and 2):

$$\phi ::= \quad \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \mathrm{A}[\phi\ \mathrm{U}\ \phi] \mid \mathrm{E}[\phi\ \mathrm{U}\ \phi] \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi.$$

where p ranges over atomic formulas/descriptions.

Thus, the symbols $\top$ and $\bot$ are CTL formulas, as are all atomic descriptions; $\neg\phi$ is a CTL formula if ϕ is one, etc. We will look at the meaning of these formulas, especially the connectives that did not occur in Chapter 1, in the next section; for now, we concentrate on their syntax. The connectives AX, EX, AG, EG, AU, EU, AF and EF are called *temporal connectives*. Notice that each of the CTL temporal connectives is a pair of symbols. The first of the pair is one of A and E. A means 'along All paths' (*inevitably*) and E means 'along at least (there Exists) one path' (*possibly*). The second one of the pair is X, F, G, or U, meaning 'neXt state', 'some Future state',

'all future states (Globally)' and Until, respectively. The pair of operators in $E[\phi_1\ U\ \phi_2]$, for example, is EU. Notice that AU and EU are binary. The symbols X, F, G and U cannot occur without being preceded by an A or an E; similarly, every A or E must have one of X, F, G and U to accompany it.

Convention 3.2 We assume similar binding priorities for the CTL connectives to what we did for propositional and predicate logic. The unary connectives (consisting of $\neg$ and the temporal connectives AG, EG, AF, EF, AX and EX) bind most tightly. Next in the order come $\wedge$ and $\vee$; and after that come $\rightarrow$, AU and EU.

Naturally, we use brackets in order to override these priorities. Let us see some examples of well-formed CTL formulas and some examples which are not well-formed, in order to understand the syntax. Suppose that p, q and r are atomic formulas. The following are well-formed CTL formulas:

- $EG\,r$;
- $AG\,(q \rightarrow EG\,r)$ — note that this is not the same as $AG\,q \rightarrow EG\,r$, for according to the binding priorities the latter formula means $(AG\,q) \rightarrow (EG\,r)$
- $A[r\ U\ q]$
- $EF\,E[r\ U\ q]$
- $A[p\ U\ EF\,r]$
- $EF\,EG\,p \rightarrow AF\,r$ (again, note that this binds as $(EF\,EG\,p) \rightarrow AF\,r$, not $EF\,(EG\,p \rightarrow AF\,r)$ or $EF\,EG\,(p \rightarrow AF\,r)$!)
- $AG\,AF\,r$
- $A[p_1\ U\ A[p_2\ U\ p_3]]$
- $E[A[p_1\ U\ p_2]\ U\ p_3]$
- $AG\,(p \rightarrow A[p\ U\ (\neg p \wedge A[\neg p\ U\ q])])$.

It is worth spending some time seeing how the syntax rules allow us to construct each of these. The following are not well-formed formulas:

- $FG\,r$ — since F and G must occur immediately after an E or an A
- $A\neg G\neg p$
- $F[r\ U\ q]$
- $EF\,(r\ U\ q)$
- $AEF\,r$
- $AF\,[(r\ U\ q) \wedge (p\ U\ r)]$.

It is especially worth understanding why the syntax rules don't allow us to construct these. For example, take $EF\,(r\ U\ q)$. EF is an operator and so is

U, so why is this not a well-formed CTL formula? The answer is that U can occur only when paired with an A or an E. The E we have is paired with the F. To make this into a well-formed CTL formula, we would have to write $\mathrm{EF}\,\mathrm{E}[r\ \mathrm{U}\ q]$ or $\mathrm{EF}\,\mathrm{A}[r\ \mathrm{U}\ q]$.

Notice that we use square brackets after the A or E, when the paired operator is a U. There is no strong reason for this; you could use ordinary round brackets instead. However, it often helps one to read the formula (because we can more easily spot where the corresponding close bracket is). Another reason for using the square brackets is that a particular model checker called SMV, which we will study later in this chapter, adopts this notation.

The reason $\mathrm{AF}\,[(r\ \mathrm{U}\ q) \wedge (p\ \mathrm{U}\ r)]$ is not a well-formed formula is that the syntax does not allow us to put a boolean connective (like $\wedge$) directly inside A[ ] or E[ ]. Occurrences of A or E must be followed by one of G, F, X or U; when they are followed by U, it must be in the form $\mathrm{A}[\phi\ \mathrm{U}\ \psi]$. Now, the ϕ and the ψ *may* contain $\wedge$, since they are arbitrary formulas; so $\mathrm{A}[(p \wedge q)\ \mathrm{U}\ (\neg r \rightarrow q)]$ *is* a well-formed formula.

Observe that AU and EU are binary connectives which mix infix and prefix notation. In pure infix, we would write $\phi_1\ \mathrm{AU}\ \phi_2$, whereas in pure prefix we would write $\mathrm{AU}(\phi_1, \phi_2)$.

As with any formal language, and as we did in the previous two chapters, it is useful to draw parse trees for well-formed formulas. The parse tree for $\mathrm{A}[\mathrm{AX}\,\neg p\ \mathrm{U}\ \mathrm{E}[\mathrm{EX}\,(p \wedge q)\ \mathrm{U}\ \neg p]]$ is shown in Figure 3.1.

Definition 3.3 A subformula of a CTL formula ϕ is any formula ψ whose parse tree is a subtree of ϕ's parse tree.

EXERCISES 3.1

We let $\{p, q, r, s, t\}$ be the set of propositional atoms for all systems of this set of exercises.

1. Write the parse trees for the following CTL formulas:
   * (a) $\mathrm{EG}\,r$
   * (b) $\mathrm{AG}\,(q \rightarrow \mathrm{EG}\,r)$
   * (c) $\mathrm{A}[p\ \mathrm{U}\ \mathrm{EF}\,r]$
   * (d) $\mathrm{EF}\,\mathrm{EG}\,p \rightarrow \mathrm{AF}\,r$ (recall Convention 3.2)
   (e) $\mathrm{A}[p\ \mathrm{U}\ \mathrm{A}[q\ \mathrm{U}\ r]]$
   (f) $\mathrm{E}[\mathrm{A}[p\ \mathrm{U}\ q]\ \mathrm{U}\ r]$
   (g) $\mathrm{AG}\,(p \rightarrow \mathrm{A}[p\ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p\ \mathrm{U}\ q])])$.
2. Explain why the following are not well-formed CTL formulas:

Fig. 3.1. The

* (a) $\mathrm{FG}\,r$
(b) $\mathrm{XX}\,r$
(c) $\mathrm{A}\neg\mathrm{G}$
(d) $\mathrm{F}[r\ \mathrm{U}$
(e) $\mathrm{EX}\,\mathrm{X}$
* (f) AEF
* (g) $\mathrm{AF}\,[($

3. State which
those which
not well-forn
(a) $\neg(\neg p$
(b) $\mathrm{X}q$
* (c) $\neg\mathrm{AX}$
(d) $p\ \mathrm{U}\ ($
* (e) $\mathrm{E}[(\mathrm{A}$
* (f) $(\mathrm{F}r) \wedge$
(g) $\neg(\mathrm{AG}$

* 4. List all subf
$q])])$.

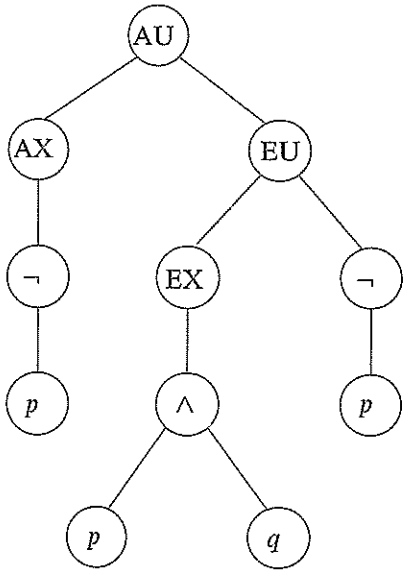

Fig. 3.1. The parse tree of a CTL formula without infix notation.

* (a) FG r
(b) XX r
(c) A¬G¬p
(d) F[r U q]
(e) EX Xr
* (f) AEF r
* (g) AF [(r U q) ∧ (p U r)].

3. State which of the strings below are well-formed CTL formulas. For those which are well-formed, draw the parse tree. For those which are not well-formed, explain why not.

(a) ¬(¬p) ∨ (r ∧ s)
(b) Xq
* (c) ¬AX q
(d) p U (AX ⊥)
* (e) E[(AX q) U (¬(¬p) ∨ (⊤ ∧ s))]
* (f) (Fr) ∧ (AG q)
(g) ¬(AG q) ∨ (EG q).

* 4. List all subformulas of the formula AG (p → A[p U (¬p ∧ A[¬p U q])]).

3.3 Semantics of computation tree logic

Definition 3.4 A model $\mathscr{M} = (S, \rightarrow, L)$ for CTL is a set of states S endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$, and a labelling function

$$L : S \rightarrow \mathscr{P}(\mathtt{Atoms}).$$

This definition looks rather mathematical; but it simply means that you have a collection of states S, a relation $\rightarrow$, saying how the system can move from state to state, and, associated with each state s, you have the set of atomic propositions $L(s)$ which are true at that particular state. We write $\mathscr{P}(\mathtt{Atoms})$ for the power set of Atoms, a collection of atomic descriptions. For example, the power set of $\{p, q\}$ is $\{\emptyset, \{p\}, \{q\}, \{p, q\}\}$. A good way of thinking about L is that it is just an assignment of truth values to all the propositional atoms as it was the case for propositional logic (we called that a *valuation*). The difference now is that we have *more than one state*, so this assignment depends on which state s the system is in: $L(s)$ contains all atoms which are true in state s.

We may conveniently express all the information about a (finite) model $\mathscr{M}$ for CTL using directed graphs whose nodes (= the states) contain all propositional atoms that are true in that state. For example, if our system has only three states s_0, s_1 and s_2; if the only possible transitions between states are $s_0 \rightarrow s_1$, $s_0 \rightarrow s_2$, $s_1 \rightarrow s_0$, $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_2$; and if $L(s_0) \stackrel{\text{def}}{=} \{p, q\}$, $L(s_1) \stackrel{\text{def}}{=} \{q, r\}$ and $L(s_2) \stackrel{\text{def}}{=} \{r\}$, then we can condense all this information into the picture in Figure 3.2. We prefer to present models by means of such pictures whenever that is feasible.

The requirement that for every $s \in S$ there is at least one $s' \in S$ such that $s \rightarrow s'$ means that no state of the system can 'deadlock'. This is no severe restriction, because we can always add an extra state s_d representing deadlock, together with new transitions $s \rightarrow s_d$ for each s which was a deadlock in the old system, as well as $s_d \rightarrow s_d$. See Figure 3.3 for such an example.

Let us define the intended meanings of the CTL connectives.

Definition 3.5 Let $\mathscr{M} = (S, \rightarrow, L)$ be a model for CTL. Given any s in S, we define whether a CTL formula ϕ holds in state s. We denote this by

$$\mathscr{M}, s \models \phi.$$

Naturally, the definition of the *satisfaction relation* $\models$ is done by structural induction on all CTL formulas:

Fig. 3.2. A concise
are states containin
state.

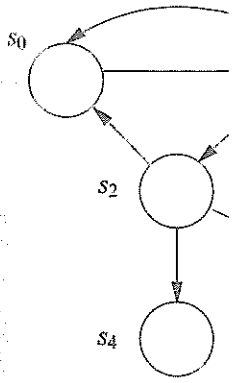

Fig. 3.3. On the left
transitions. On the
no state can deadl
'deadlock' state cor

1. $\mathscr{M}, s \models \top$ a
2. $\mathscr{M}, s \models p$ if
3. $\mathscr{M}, s \models \neg\phi$
4. $\mathscr{M}, s \models \phi_1$ /
5. $\mathscr{M}, s \models \phi_1$ \
6. $\mathscr{M}, s \models \phi_1$ -
7. $\mathscr{M}, s \models \mathrm{AX}$
   AX says: '
8. $\mathscr{M}, s \models \mathrm{EX}$

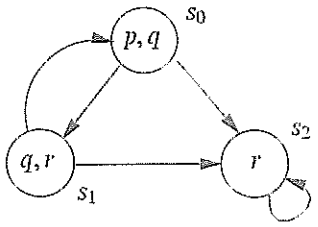

Fig. 3.2. A concise representation of a model $\mathscr{M}$ as a directed graph, whose nodes are states containing all the propositional atoms which are true in that particular state.

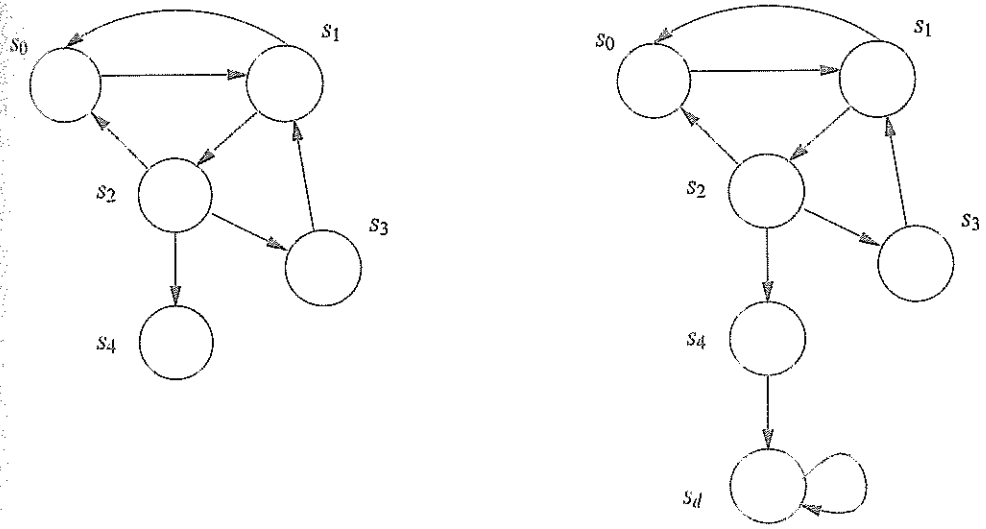

Fig. 3.3. On the left, we have a system with a state s_4 that does not have any further transitions. On the right, we expand that system with a 'deadlock' state s_d such that no state can deadlock; of course, it is then our understanding that reaching the 'deadlock' state corresponds to deadlock in the original system.

1. $\mathscr{M}, s \models \top$ and $\mathscr{M}, s \not\models \bot$ for all $s \in S$.
2. $\mathscr{M}, s \models p$ iff $p \in L(s)$.
3. $\mathscr{M}, s \models \neg\phi$ iff $\mathscr{M}, s \not\models \phi$.
4. $\mathscr{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathscr{M}, s \models \phi_1$ and $\mathscr{M}, s \models \phi_2$.
5. $\mathscr{M}, s \models \phi_1 \vee \phi_2$ iff $\mathscr{M}, s \models \phi_1$ or $\mathscr{M}, s \models \phi_2$.
6. $\mathscr{M}, s \models \phi_1 \rightarrow \phi_2$ iff $\mathscr{M}, s \not\models \phi_1$ or $\mathscr{M}, s \models \phi_2$.
7. $\mathscr{M}, s \models \mathrm{AX}\,\phi$ iff *for all* s_1 such that $s \rightarrow s_1$ we have $\mathscr{M}, s_1 \models \phi$. Thus, AX says: 'in every next state'.
8. $\mathscr{M}, s \models \mathrm{EX}\,\phi$ iff *for some* s_1 such that $s \rightarrow s_1$ we have $\mathscr{M}, s_1 \models \phi$. Thus,

EX says: 'in some next state'. E is dual to A — in exactly the same way that $\exists$ is dual to $\forall$ in predicate logic.

9. $\mathcal{M}, s \models \mathrm{AG}\,\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *for All* computation paths beginning in s the property ϕ holds *Globally*. Note that 'along the path' includes the path's initial state s.
10. $\mathcal{M}, s \models \mathrm{EG}\,\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and for all s_i along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *there Exists* a path beginning in s such that ϕ holds *Globally* along the path.
11. $\mathcal{M}, s \models \mathrm{AF}\,\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow \ldots$, where s_1 equals s, there is some s_i such that $\mathcal{M}, s_i \models \phi$. Mnemonically: *for All* computation paths beginning in s there will be some *Future* state where ϕ holds.
12. $\mathcal{M}, s \models \mathrm{EF}\,\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and for some s_i along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *there Exists* a computation path beginning in s such that ϕ holds in some *Future* state;
13. $\mathcal{M}, s \models \mathrm{A}[\phi_1\ \mathrm{U}\ \phi_2]$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, that path satisfies $\phi_1\ \mathrm{U}\ \phi_2$, i.e. there is some s_i along the path, such that $\mathcal{M}, s_i \models \phi_2$, and, for each $j < i$, we have $\mathcal{M}, s_j \models \phi_1$. Mnemonically: *All* computation paths beginning in s satisfy that ϕ_1 *Until* ϕ_2 holds on it.
14. $\mathcal{M}, s \models \mathrm{E}[\phi_1\ \mathrm{U}\ \phi_2]$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and that path satisfies $\phi_1\ \mathrm{U}\ \phi_2$ as specified in 13. Mnemonically: *there Exists* a computation path beginning in s such that ϕ_1 *Until* ϕ_2 holds on it.

For the remainder of this section, we will be concerned with the computational consequences of this definition, and with justifying and explaining its clauses.

The first six clauses are exactly what we expect from our knowledge of propositional logic gained in Chapter 1. Notice that, for example, the truth value of $\neg\phi$ in a state depends on the truth value of ϕ in the *same* state. This contrasts with the clauses for AX and EX. The truth value of $\mathrm{AX}\,\phi$ in a state s is determined not by the truth value of ϕ in s, but by ϕ in states that are related to s by the relation $\rightarrow$; if $s \rightarrow s$, then this value also depends on the truth value of ϕ in s.

The next four clauses also exhibit this phenomenon, except more so: for example, to determine the truth value of $\mathrm{AG}\,\phi$ involves looking at the truth

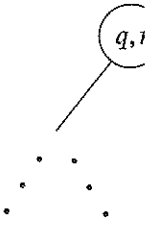

Fig. 3.4. Unwinding t
paths beginning in a

value of ϕ not only
states as well. In fa
value of ϕ in every
the current state.

Clauses 9–14 abo
useful to visualise a
unwinding the tran
whence 'computatio
state satisfies a CTL
Figure 3.2 for the de
Figure 3.4.

The diagrams in F
states satisfy the fo
course, we could ad
the satisfaction — a
illustrate a 'least' wa

Remark 3.6 Notice t

[1] If this task is done by a
visualisation of a system

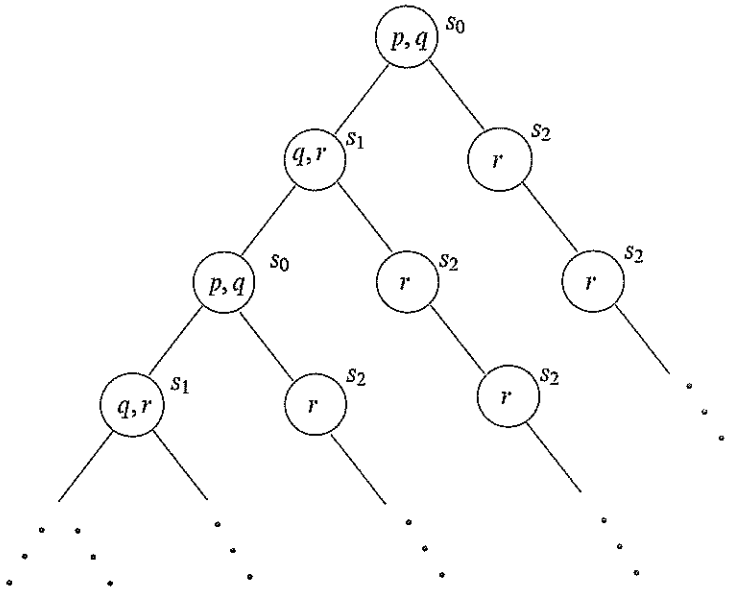

Fig. 3.4. Unwinding the system of Figure 3.2 as an infinite tree of all computation paths beginning in a particular state.

value of ϕ not only in the immediately related states, but in indirectly related states as well. In fact, in the case of $AG\,\phi$, you have to examine the truth value of ϕ in every state related by any number of forward links of $\rightarrow$ to the current state.

Clauses 9–14 above refer to computation paths in models. It is therefore useful to visualise all possible computation paths from a given state s by unwinding the transition system to obtain an infinite computation tree, whence 'computation tree logic'. This greatly facilitates deciding whether a state satisfies a CTL formula[1]. For example, if we unwind the state graph of Figure 3.2 for the designated starting state s_0, then we get the infinite tree in Figure 3.4.

The diagrams in Figures 3.5-3.8 show schematically systems whose starting states satisfy the formulas $EF\,\phi$, $EG\,\phi$, $AG\,\phi$ and $AF\,\phi$, respectively. Of course, we could add more ϕ to any of these diagrams and still preserve the satisfaction — although there is nothing to add for AG. The diagrams illustrate a 'least' way of satisfying the formulas.

Remark 3.6 Notice that, in clauses 9–14 above, *the future includes the present.*

[1] If this task is done by a computer, then that is a different matter, but *we* seem to benefit from this visualisation of a system as a tree.

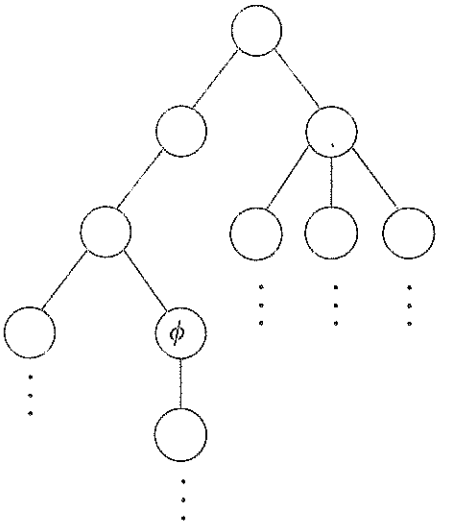

Fig. 3.5. A system whose starting state satisfies EF ϕ.

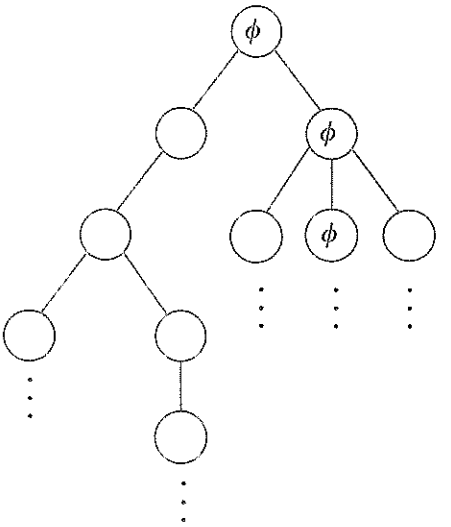

Fig. 3.6. A system whose starting state satisfies EG ϕ.

This means that, when we say 'in all future states', we are including the present state as a future state. It is a matter of convention whether we do this, or not; see the exercises below for an *exclusive* version of CTL. A consequence of adopting the convention that the future shall include the present is that the formulas $(\mathrm{AG}\, p) \rightarrow p$, $p \rightarrow \mathrm{A}[q\ \mathrm{U}\ p]$ and $p \rightarrow \mathrm{EF}\, p$ are true in every state of every model.

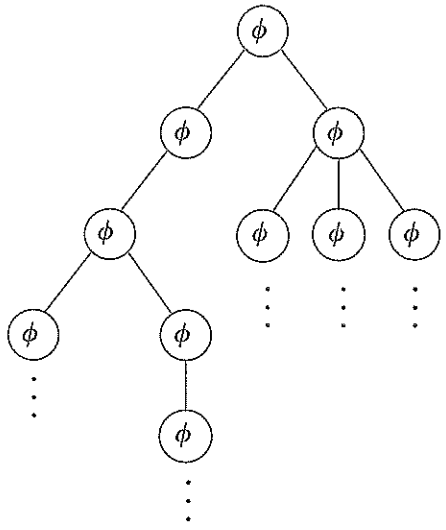

Fig. 3.7. A system whose starting state satisfies AG ϕ.

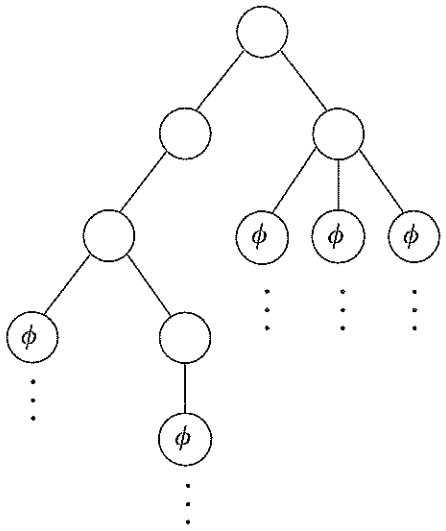

Fig. 3.8. A system whose starting state satisfies AF ϕ.

We now move our discussion to the clauses for AU and EU. The symbol U stands for 'until'. The formula ϕ_1 U ϕ_2 holds on a path if it is the case that ϕ_1 holds continuously *until* the next occurrence of ϕ_2. Moreover, ϕ_1 U ϕ_2 actually demands that ϕ_2 *does* hold in some future state, i.e. it is not enough for ϕ_1 to hold continuously forever. See Figure 3.9 for illustration: each of the states s_3 to s_9 satisfies p U q along the path shown, but s_0 to s_2 don't.

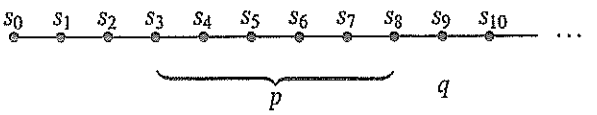

Fig. 3.9. An illustration of the meaning of Until in the semantics of CTL. Each of the states s_3 to s_9 satisfies p U q along the path shown.

Thus, we defined earlier that a particular path $s_0 \to s_1 \to s_2 \to \ldots$ satisfies ϕ_1 U ϕ_2 iff:

- there is some s_i along the path, such that $s_i \models \phi_2$, and
- for each $j < i$, we have $s_j \models \phi_1$.

The clauses for AU and EU given in Definition 3.5 reflect this intuition, for all paths and some path, respectively. Note that the semantics of ϕ_1 U ϕ_2 is not saying anything about ϕ_1 in state s_i; neither does it say anything about ϕ_2 in states s_j with $j < i$. This might be in contrast to some of the implicit meanings of 'until' in natural language usage. For example, in the sentence 'I smoked until I was 22.' it is not only expressed that the person referred to continually smoked up until he, or she, was 22 years old, but we also would interpret such a sentence as saying that this person gave up smoking from that point onwards. This is different from the semantics of Until in temporal logic.

It should be clear that we have outlined the formal foundations of a procedure that, given ϕ, $\mathcal{M}$ and s, can check whether

$$\mathcal{M}, s \models \phi$$

holds. In particular, if the given set of states is finite, then we may compute the set of *all* states satisfying ϕ. If the model $\mathcal{M}$ is clear from the context, we will simply write $s \models \phi$ instead of $\mathcal{M}, s \models \phi$. Let us now look at some example checks for the system in Figures 3.2 and 3.4.

1. $\mathcal{M}, s_0 \models p \wedge q$ holds since the atomic symbols p and q are contained in the node of s_0.
2. $\mathcal{M}, s_0 \models \neg r$ holds since the atomic symbol r is *not* contained in node s_0.
3. $\mathcal{M}, s_0 \models \top$ holds by definition.
4. $\mathcal{M}, s_0 \models \mathrm{EX}\,(q \wedge r)$ holds since we have the leftmost computation path $s_0 \to s_1 \to s_0 \to s_1 \to \ldots$ in Figure 3.4, whose second node s_1 contains q and r.
5. $\mathcal{M}, s_0 \models \neg\mathrm{AX}\,(q \wedge r)$ holds since we have the rightmost computation

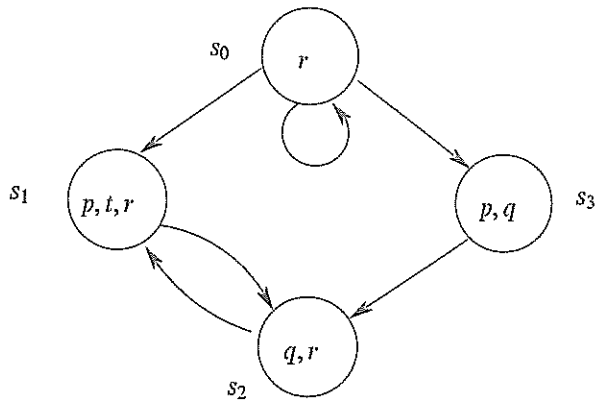

Fig. 3.10. A system with four states.

path $s_0 \to s_2 \to s_2 \to s_2 \to \ldots$ in Figure 3.4, whose second node s_2 only contains r, but *not* q.

6. $\mathcal{M}, s_0 \models \neg \mathrm{EF}(p \wedge r)$ holds since there is no computation path beginning in s_0 such that we could reach a state where $p \wedge r$ would hold. This is so, because there is simply no state whatsoever in this system, where p and r hold at the same time.
7. $\mathcal{M}, s_2 \models \mathrm{EG}\, r$ (note the s_2 instead of s_0!) holds since there is a computation path $s_2 \to s_2 \to s_2 \to \ldots$ beginning in s_2 such that r holds in all future states.
8. $\mathcal{M}, s_2 \models \mathrm{AG}\, r$ holds as well, since there is only *one* computation path beginning in s_2 and it satisfies r globally.
9. $\mathcal{M}, s_0 \models \mathrm{AF}\, r$ holds since, for all computation paths beginning in s_0, the system reaches a state (s_1 or s_2) such that r holds.
10. $\mathcal{M}, s_0 \models \mathrm{E}[(p \wedge q)\ \mathrm{U}\ r]$ holds since we have the rightmost computation path $s_0 \to s_2 \to s_2 \to s_2 \to \ldots$ in Figure 3.4, whose second node s_2 ($i = 1$) satisfies r, but all previous nodes (only $j = 0$, i.e. node s_0) satisfy $p \wedge q$.
11. $\mathcal{M}, s_0 \models \mathrm{A}[p\ \mathrm{U}\ r]$ holds since p holds at s_0 and r holds in any possible successor state of s_0, so $p\ \mathrm{U}\ r$ is true for all computation paths beginning in s_0 (so we may choose $i = 1$ independently of the path).

EXERCISES 3.2

We let $\{p, q, r, t\}$ be the set of propositional atoms for all systems of this set of exercises.

1. Consider the system $\mathcal{M}$ in Figure 3.10.

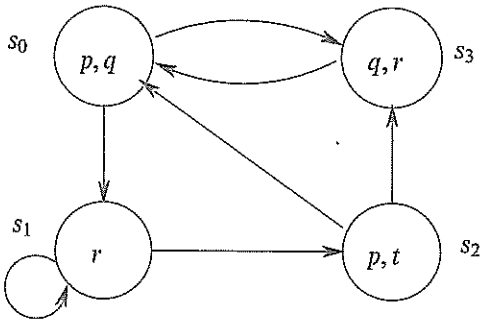

Fig. 3.11. Another system with four states.

(a) Beginning from state s_0, unwind this system into an infinite tree, and draw all computation paths up to length 4 (= the first four layers of that tree).

(b) Make the following checks $\mathcal{M}, s_0 \models \phi$, where ϕ is listed below. For that you need to explain why the check holds, or what reasons there are for its failure:

* (i) $\neg p \rightarrow r$
(ii) $\mathrm{AF}\, t$
*(iii) $\neg \mathrm{EG}\, r$
(iv) $\mathrm{E}(t\ \mathrm{U}\ q)$
(v) $\mathrm{AF}\, q$
(vi) $\mathrm{EF}\, q$
(vii) $\mathrm{EG}\, r$
(viii) $\mathrm{AG}(r \vee q)$.

(c) Make the same checks as in (b) but now for state s_2.

2. Consider the following system $\mathcal{M}$ in Figure 3.11. Check the following CTL formulas ϕ for state s_0, i.e. determine whether $\mathcal{M}, s_0 \models \phi$ holds:

(a) $\mathrm{AF}\, q$
(b) $\mathrm{AG}(\mathrm{EF}(p \vee r))$
(c) $\mathrm{EX}(\mathrm{EX}\, r)$
(d) $\mathrm{AG}(\mathrm{AF}\, q)$.

3. Do the same as in exercise 2, but for state s_2.

* 4. The meaning of the temporal operators AU, EU, AG, EG, AF and EF was defined to be such that 'the present includes the future'. For example, EF p is true for a state if p is true for that state already. Often one would like corresponding operators such that *the future excludes*

the present. Use suitable connectives of the grammar in Definition 3.1 to define such (six) modified connectives as derived operators in CTL.

3.3.1 Practical patterns of specifications

What kind of practically relevant properties can we check with formulas of CTL? We list a few of the common patterns. Suppose atomic descriptions include some words such as busy and requested. We may require some of the following properties of real systems:

- It is possible to get to a state where started holds, but ready does not hold:
  EF(started ∧ ¬ready).
- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
  AG(requested → AF acknowledged).
- A certain process is enabled infinitely often on every computation path:
  AG(AF enabled).
- Whatever happens, a certain process will eventually be permanently deadlocked:
  AF(AG deadlock).
- From any state it is possible to get to a restart state:
  AG(EF restart).
- An upwards travelling elevator at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  AG(floor=2 ∧ direction=up ∧ ButtonPressed5 →
  A[direction=up U floor=5])
  Here, our atomic descriptions are boolean expressions built from system variables, e.g. floor=2.
- The elevator can remain idle on the third floor with its doors closed:
  AG(floor=3 ∧ idle ∧ door=closed → EG(floor=3 ∧ idle ∧ door=closed)).

EXERCISES 3.3

* 1. Express in CTL: Whenever p is followed by q (after some finite amount of steps), then the system enters an 'interval' in which no r occurs until t.

2. Explain in detail why the CTL formulas for the practical specification patterns above capture the stated 'informal' properties expressed in plain English.

3. Consider the CTL formula $\mathrm{AG}(p \rightarrow \mathrm{AF}(s \wedge \mathrm{AX}(\mathrm{AF}\, t)))$. Explain what exactly it expresses in terms of the order of occurrence of events p, s and t.
4. Write down a CTL formula which says that p precedes s and t on all computation paths; you may find it easier to code the negation of that specification first.
5. Represent 'After p, q is never true.' as a CTL formula, where this constraint is meant to apply on all computation paths.
6. Find a CTL formula which expresses the following property on all computation paths:

   (a) 'Between the events q and r, p is never true.'
   (b) 'Transitions to states satisfying p occur at most twice.'.

3.3.2 *Important equivalences between CTL formulas*

Definition 3.7 Two CTL formulas ϕ and ψ are said to be *semantically equivalent* if any state in any model which satisfies one of them also satisfies the other; we denote this by $\phi \equiv \psi$.

Note that we wrote $\phi \equiv \psi$ in Chapter 1 if the propositional logic formulas ϕ and ψ had the same meaning no matter what valuation (= assignment of truth values) one considers. We may think of such valuations as $L(s)$ for a CTL model with one state s and one transition $s \rightarrow s$. Therefore, we see how the definition above extends the meaning of $\equiv$ to a larger class of models and formulas.

We have already noticed that A is a universal quantifier on paths and E is the corresponding existential quantifier. Moreover, G and F are also universal and existential quantifiers, ranging over the states along a particular path. In view of these facts, it is not surprising to find that de Morgan rules exist for A and E and also F and G:

$$\begin{aligned} \neg \mathrm{AF}\, \phi &\equiv \mathrm{EG} \neg \phi \\ \neg \mathrm{EF}\, \phi &\equiv \mathrm{AG} \neg \phi. \end{aligned} \tag{3.1}$$

On any particular path, each state has a unique successor. Therefore, X is its own dual on computation paths and we have

$$\neg \mathrm{AX}\, \phi \equiv \mathrm{EX} \neg \phi. \tag{3.2}$$

We also have the equivalences

$$\mathrm{AF}\,\phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi] \qquad \mathrm{EF}\,\phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi].$$

You can check this by looking back at the meaning of the clause for U. For $\phi_1\ \mathrm{U}\ \phi_2$ to be true in a state along a path, it is necessary that ϕ_2 become true at some future point and that ϕ_1 is true in every state until that point. In the expression above, we let ϕ_1 be $\top$, so that the requirement that ϕ_1 be true in every state until ϕ_2 is vacuously satisfied (since $\top$ is true in every state).

Naturally, any equivalence which holds in propositional logic will also hold in CTL. This is true even if the equivalence involves CTL formulas: e.g. we have that $\top$ and $(\mathrm{AX}\,p) \vee \neg\mathrm{AX}\,p$ are equivalent since the formula scheme $\phi \vee \neg\phi$ of propositional logic is equivalent to $\top$. Therefore, as in propositional logic, there is some redundancy among the connectives. For example, in Chapter 1 we saw that the set $\{\bot, \wedge, \neg\}$ forms an adequate set of connectives, since the other connectives $\vee$, $\rightarrow$, $\top$, etc., can be written in terms of those three. (Cf. exercise 3, page 71.)

This is also the case in CTL. Moreover, the connective AX can be written $\neg\mathrm{EX}\neg$ by (3.2); and AG, AF, EG and EF can be written in terms of AU and EU as follows: first, write $\mathrm{AG}\,\phi$ as $\neg\mathrm{EF}\neg\phi$ and $\mathrm{EG}\,\phi$ as $\neg\mathrm{AF}\neg\phi$ (using (3.1)) and then use $\mathrm{AF}\,\phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi]$ and $\mathrm{EF}\,\phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi]$. Therefore AU, EU and EX form an adequate set of temporal connectives.

Also EG, EU, and EX form an adequate set, for we have the equivalence

$$\mathrm{A}[\phi\ \mathrm{U}\ \psi] \equiv \neg(\mathrm{E}[\neg\psi\ \mathrm{U}\ (\neg\phi \wedge \neg\psi)] \vee \mathrm{EG}\,\neg\psi). \tag{3.3}$$

This equivalence is rather harder to demonstrate than the others; we will do it later (Section 3.8.1).

Similarly, AG, AU and AX form an adequate set. There are many other adequate sets, but we just mention one more, since we will use it later: the set consisting of AF, EU and EX. Since AF can be reduced to EG, this set reduces to EG, EU and EX which we have seen to be adequate.

Theorem 3.8 *The set of operators $\bot$, $\neg$ and $\wedge$ together with* AF, EU *and* EX *are adequate for CTL: any CTL formula can be transformed into a semantically equivalent CTL formula which uses only those logical connectives.*

Some other noteworthy equivalences in CTL are the following:

$$\begin{aligned}
\mathrm{AG}\,\phi &\equiv \phi \wedge \mathrm{AX}\,\mathrm{AG}\,\phi \\
\mathrm{EG}\,\phi &\equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi \\
\mathrm{AF}\,\phi &\equiv \phi \vee \mathrm{AX}\,\mathrm{AF}\,\phi \\
\mathrm{EF}\,\phi &\equiv \phi \vee \mathrm{EX}\,\mathrm{EF}\,\phi \\
\mathrm{A}[\phi\ \mathrm{U}\ \psi] &\equiv \psi \vee (\phi \wedge \mathrm{AX}\,\mathrm{A}[\phi\ \mathrm{U}\ \psi]) \\
\mathrm{E}[\phi\ \mathrm{U}\ \psi] &\equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi\ \mathrm{U}\ \psi]).
\end{aligned}$$

For example, the intuition for the third one is the following: in order to have $\mathrm{AF}\,\phi$ in a particular state, ϕ must be true at some point along each path from that state. To achieve this, we either have ϕ true now, in the current state; or we postpone it, in which case we must have $\mathrm{AF}\,\phi$ in each of the next states. Notice how this equivalence appears to define AF in terms of AX and AF itself — an apparently circular definition. In fact, these equivalences can be used to define the six connectives on the left in terms of AX and EX, in a *non-circular* way. This is called the fixed-point characterisation of CTL; it is the mathematical foundation for the model-checking algorithm developed in Section 3.5; and we return to it later (Section 3.9).

EXERCISES 3.4

1. Which of the following pairs of CTL formulas are equivalent? For those which are not, exhibit a model of one of the pair which is not a model of the other.
   (a) $\mathrm{EF}\,\phi$ and $\mathrm{EG}\,\phi$
   * (b) $\mathrm{EF}\,\phi \vee \mathrm{EF}\,\psi$ and $\mathrm{EF}\,(\phi \vee \psi)$
   * (c) $\mathrm{AF}\,\phi \vee \mathrm{AF}\,\psi$ and $\mathrm{AF}\,(\phi \vee \psi)$
   (d) $\mathrm{AF}\,\neg\phi$ and $\neg\mathrm{EG}\,\phi$
   * (e) $\mathrm{EF}\,\neg\phi$ and $\neg\mathrm{AF}\,\phi$
   (f) $\mathrm{A}[\phi_1\ \mathrm{U}\ \mathrm{A}[\phi_2\ \mathrm{U}\ \phi_3]]$ and $\mathrm{A}[\mathrm{A}[\phi_1\ \mathrm{U}\ \phi_2]\ \mathrm{U}\ \phi_3]$ (Hint: it might make it simpler if you think first about models that have just one path.)
   (g) $\top$ and $\mathrm{AG}\,\phi \rightarrow \mathrm{EG}\,\phi$
   * (h) $\top$ and $\mathrm{EG}\,\phi \rightarrow \mathrm{AG}\,\phi$.
2. Find operators to replace the ? marks, to make the following equivalences.
   (a) $\mathrm{EF}\,\neg\phi \equiv \neg??\phi$
   * (b) $\mathrm{AG}\,(\phi \wedge \psi) \equiv \mathrm{AG}\,\phi\ ?\ \mathrm{AG}\,\psi$.
3. Use the definition of $\models$ between states and CTL formulas to explain why $s \models \mathrm{AG}\,\mathrm{AF}\,\phi$ means that ϕ is true infinitely often along every path starting at s.

4. Prove the Equivalences (3.1) and (3.2).
* 5. Write pseudo-code for a recursive function TRANSLATE which takes as input an arbitrary CTL formula ϕ and returns as output an equivalent CTL formula ψ whose only operators are among the set $\{\bot, \neg, \wedge, \mathrm{AF}, \mathrm{EU}, \mathrm{EX}\}$.

3.4 Example: mutual exclusion

Let us now look at a larger example of verification using CTL, having to do with *mutual exclusion*. When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time. Several processes simultaneously editing the same file would not be desirable.

We therefore identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time. The critical section should include all the access to the shared resource (though it should be as small as possible so that no unnecessary exclusion takes place). The problem we are faced with is to find a *protocol* for determining which process is allowed to enter its critical section at which time. Once we have found one which we think works, we verify our solution by checking that it has some expected properties, such as the following ones:

Safety: The protocol allows only one process to be in its critical section at any time.

This safety property is not enough, since a protocol which permanently excluded every process from its critical section would be safe, but not very useful. Therefore, we should also require:

Liveness: Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

Non-blocking: A process can always request to enter its critical section.

Some rather crude protocols might work on the basis that they cycle through the processes, making each one in turn enter its critical section. Since it might be naturally the case that some of them request accesses to the shared resource more than others, we should make sure our protocol has the property:

No strict sequencing: Processes need not enter their critical section in strict sequence.

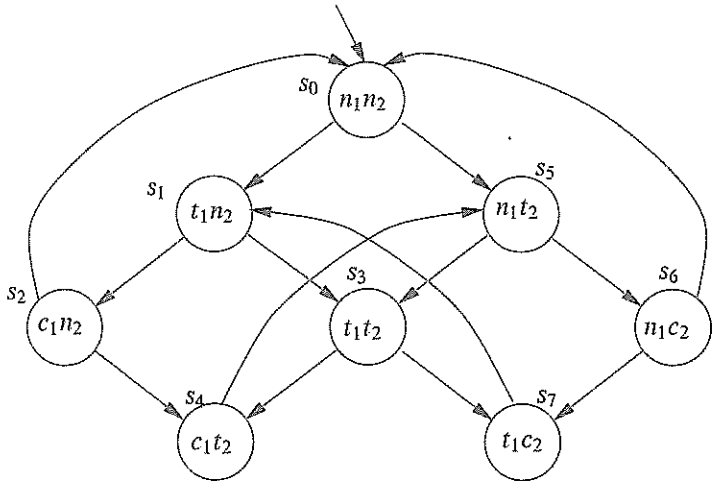

Fig. 3.12. A first-attempt model for mutual exclusion.

3.4.1 *The first modelling attempt*

We will model two processes, each of which is in its non-critical state (n), or trying to enter its critical state (t), or in its critical state (c). Each individual process undergoes transitions in the cycle $n \to t \to c \to n \to \ldots$, but the two processes interleave with each other. Consider the protocol given by the transition system $\mathscr{M}$ in Figure 3.12[1]. The two processes start off in their non-critical sections (global state s_0). State s_0 is the only *initial* state, indicated by the incoming edge with no source. Either of them may now move to its trying state, but only one of them can ever make a transition at a time (asynchronous *interleaving*). So there is a transition arrow from s_0 to s_1 and s_5. From s_1 (i.e. process 1 trying, process 2 non-critical) again two things can happen: either process 1 moves again (we go to s_2), or process 2 moves (we go to s_3). Notice that not every process can move in every state. For example, process 1 cannot move in state s_7, since it cannot go into its critical section until process 2 comes out of its critical section.

We can verify the four properties by first describing them as CTL formulas:

Safety: $\phi_1 \stackrel{\text{def}}{=} \mathrm{AG}\,\neg(c_1 \wedge c_2)$. Clearly, $\mathrm{AG}\,\neg(c_1 \wedge c_2)$ is satisfied in the initial state (indeed, in every state).

Liveness: $\phi_2 \stackrel{\text{def}}{=} \mathrm{AG}(t_1 \to \mathrm{AF}\,c_1)$. This is *not* satisfied by the initial state, for we can find a state accessible from the initial state, namely s_1, in

[1] We write $p_1p_2\ldots p_m$ in a node s to denote that $p_1, p_2, \ldots, p_m$ are the only propositional atoms true at s.

which t_1 is true but $\mathrm{AF}\, c_1$ is false, because there is a computation path $s_1 \to s_3 \to s_7 \to s_1 \to \ldots$ on which c_1 is always false.

Non-blocking: $\phi_3 \stackrel{\text{def}}{=} \mathrm{AG}(n_1 \to \mathrm{EX}\, t_1)$, which is satisfied, since every n_1 state (i.e. s_0, s_5 and s_6) has an (immediate) t_1 successor.

No strict sequencing: $\phi_4 \stackrel{\text{def}}{=} \mathrm{EF}(c_1 \wedge \mathrm{E}[c_1\ \mathrm{U}\ (\neg c_1 \wedge \mathrm{E}[\neg c_2\ \mathrm{U}\ c_1])])$. This is satisfied; e.g. by the mirror path to the computation path described for liveness, $s_5 \to s_3 \to s_4 \to s_5 \to \ldots$.

EXERCISES 3.5

* 1. Observe that the last three specifications have been written from the point of view of process 1. Can you modify those specifications such that process 2 also meets these constraints? Can you come up with a good reason explaining why we did not add the constraints of process 2?

The reason liveness failed in our first attempt at modelling mutual exclusion is that non-determinism means it *might* continually favour one process over another. The problem is that the state s_3 does not distinguish between which of the processes *first* went into its trying state. We can solve this by splitting s_3 into two states.

3.4.2 The second modelling attempt

The two states s_3 and s_9 in Figure 3.13 both correspond to the state s_3 in our first modelling attempt. They both record that both processes are in their trying states, but in s_3 it is implicitly recorded that it is process 1's turn, whereas in s_9 it is process 2's turn. Note that states s_3 and s_9 both have the labelling $t_1 t_2$. The definition of CTL models does not preclude this. We can think of there being some other, hidden, variables which are not part of the initial labelling, which distinguish s_3 and s_9.

Remark 3.9 The four properties of safety, liveness, non-blocking and no strict sequencing are satisfied by the model in Figure 3.13.

In this second modelling attempt, our transition system is still slightly over-simplified, because we are assuming that it will move to a different state on every tick of the clock (there are no transitions to the same state). We may wish to model that a process can stay in its critical state for several ticks, but if we include an arrow from s_2, or s_6, to itself, we will again violate

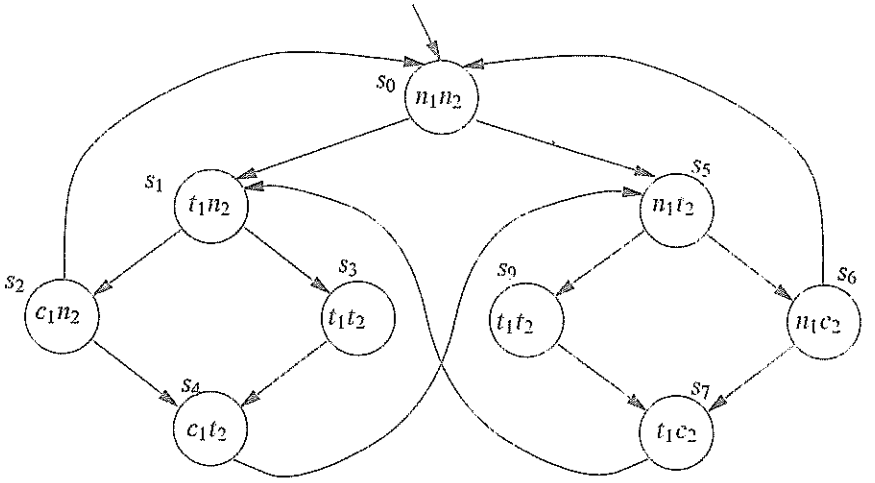

Fig. 3.13. A second-attempt model for mutual exclusion. There are now two states representing t_1t_2, namely s_3 and s_9.

liveness. This problem will be solved later in this chapter when we consider 'fairness constraints' (Section 3.7).

EXERCISES 3.6

1. Verify Remark 3.9.

3.5 A model-checking algorithm

In general, interesting transition systems will have millions of states and the formula we are interested in checking may be quite long. It is therefore well worth trying to find an efficient algorithm.

Our usage of the notion of a 'model' $\mathcal{M}$ has included directed graphs and also their unwindings into infinite trees, given a designated initial state. On doing the exercises, you very probably realised that checks on the infinite tree are easier for you to do, since all possible paths are plainly visible. However, if we think of implementing a model checker on a computer, we certainly cannot unwind transition systems into an infinite tree. We need to do checks on *finite* data structures. This is the reason why we now have to develop new insights into the semantics of CTL. Such a deeper understanding will provide the basis for an efficient algorithm which, given $\mathcal{M}$, $s \in S$ and ϕ, computes whether $\mathcal{M}, s \models \phi$ holds. In the latter case, such an algorithm can be augmented to produce an actual path (= run) of the system demonstrating

that $\mathscr{M}$ cannot satisfy ϕ. That way, we may *debug* a system by trying to fix what causes that run to refute property ϕ.

There are various ways in which one could consider

$$\mathscr{M}, s_0 \stackrel{?}{\models} \phi$$

as a computational problem. For example, one could have the model $\mathscr{M}$, the formula ϕ and a state s_0 as input; one would then expect a reply of the form 'yes' ($\mathscr{M}, s_0 \models \phi$ holds), or 'no' ($\mathscr{M}, s_0 \models \phi$ does not hold). Alternatively, the inputs could be just $\mathscr{M}$ and ϕ, where the output would be *all* states s of the model $\mathscr{M}$ which satisfy ϕ.

If we solve the second of these two problems, we automatically have a solution to the first one, since we can simply check whether s_0 is an element of the output set. Conversely, to solve the second problem, given an algorithm for the first, we would simply repeatedly call this algorithm with each state in turn, to decide whether it goes in the output set. We will describe an algorithm to solve the second problem.

3.5.1 *The labelling algorithm*

We present an algorithm which, given a model and a CTL formula, outputs the set of states of the model that satisfy the formula. The algorithm does not need to be able to handle every CTL connective explicitly, since we have already seen that the connectives $\bot$, $\neg$ and $\wedge$ form an adequate set as far as the propositional connectives are concerned; and AF, EU and EX form an adequate set of temporal connectives. Given an arbitrary CTL formula ϕ, we would simply pre-process ϕ, calling the model-checking algorithm with the output of TRANSLATE (ϕ) as input, where TRANSLATE is the function coded in Exercises 3.4(5), page 169. Here is the algorithm:

INPUT: a CTL model $\mathscr{M} = (S, \rightarrow, L)$ and a CTL formula ϕ.
OUTPUT: the set of states of $\mathscr{M}$ which satisfy ϕ.

First, change ϕ to the output of TRANSLATE (ϕ), i.e. we write ϕ in terms of the connectives AF, EU, EX, $\wedge$, $\neg$ and $\bot$ using the equivalences given earlier in the chapter. Next, label the states of $\mathscr{M}$ with the subformulas of ϕ that are satisfied there, starting with the smallest subformulas and working outwards towards ϕ.

Suppose ψ is a subformula of ϕ and states satisfying all the *immediate* subformulas of ψ have already been labelled. (An immediate subformula of ϕ is any maximal-length subformula other than ϕ itself.) We determine by a case analysis which states to label with ψ. If ψ is

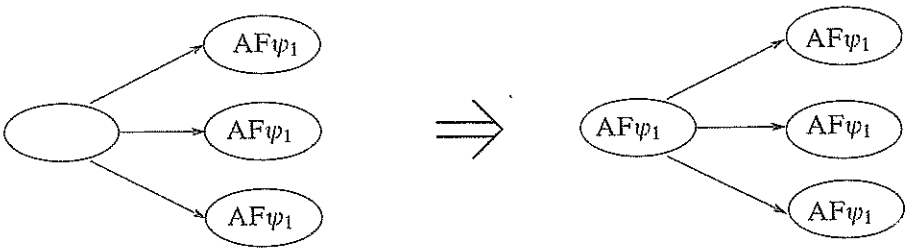

Fig. 3.14. The iteration step of the procedure for labelling states with subformulas of the form $\mathrm{AF}\,\psi_1$.

- $\bot$: then no states are labelled with $\bot$.
- p: then label s with p if $p \in L(s)$.
- $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled both with ψ_1 and with ψ_2.
- $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1.
- $\mathrm{AF}\,\psi_1$:
  - If any state s is labelled with ψ_1, label it with $\mathrm{AF}\,\psi_1$.
  - Repeat: label any state with $\mathrm{AF}\,\psi_1$ if all successor states are labelled with $\mathrm{AF}\,\psi_1$, until there is no change. This step is illustrated in Figure 3.14.
- $\mathrm{E}[\psi_1\ \mathrm{U}\ \psi_2]$:
  - If any state s is labelled with ψ_2, label it with $\mathrm{E}[\psi_1\ \mathrm{U}\ \psi_2]$.
  - Repeat: label any state with $\mathrm{E}[\psi_1\ \mathrm{U}\ \psi_2]$ if it is labelled with ψ_1 and at least one of its successors is labelled with $\mathrm{E}[\psi_1\ \mathrm{U}\ \psi_2]$, until there is no change. This step is illustrated in Figure 3.15.
- $\mathrm{EX}\,\psi_1$: label any state with $\mathrm{EX}\,\psi_1$ if one of its successors is labelled with ψ_1.

Having performed the labelling for all the subformulas of ϕ (including ϕ itself), we output the states which are labelled ϕ.

The complexity of this algorithm is $O(f \cdot V \cdot (V + E))$, where f is the number of connectives in the formula, V is the number of states and E is the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

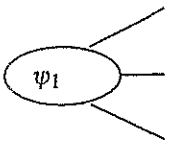

Fig. 3.15. The ite
of the form $\mathrm{E}[\psi_1$

Instead of using
possible to write
probably be mo
different approa
to deal with EG

- $\mathrm{EG}\,\psi_1$:
  - Label *all* tl
  - If any state
  - Repeat: *de*
    is labelled

Here, we label
down this label
the case for EU
for EG and wh
final result is co

We can improve
way of handling
we use EX, EU
take care to sea
ensures that we

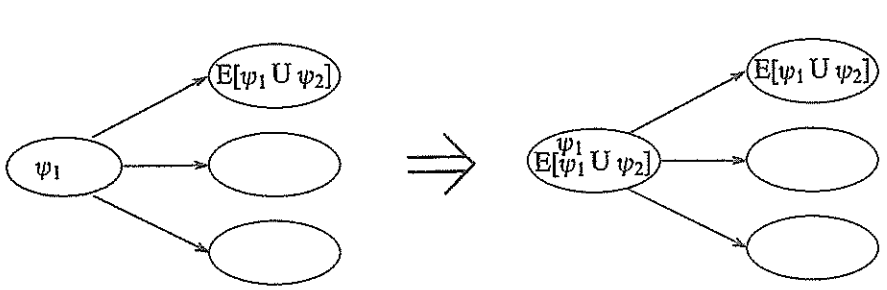

Fig. 3.15. The iteration step of the procedure for labelling states with subformulas of the form $E[\psi_1 \, U \, \psi_2]$.

Handling EG *directly*

Instead of using a minimal adequate set of connectives, it would have been possible to write similar routines for the other connectives. Indeed, this would probably be more efficient. The connectives AG and EG require a slightly different approach from that for the others, however. Here is the algorithm to deal with $EG\,\psi_1$ *directly*:

- $EG\,\psi_1$:
  - Label *all* the states with $EG\,\psi_1$.
  - If any state s is *not* labelled with ψ_1, *delete* the label $EG\,\psi_1$.
  - Repeat: *delete* the label $EG\,\psi_1$ from any state if *none* of its successors is labelled with $EG\,\psi_1$; until there is no change.

Here, we label all the states with the subformula $EG\,\psi_1$ and then whittle down this labelled set, instead of building it up from nothing as we did in the case for EU. Actually, there is no real difference between this procedure for EG and what you would do if you translated it into $\neg AF\neg$ as far as the final result is concerned.

A variant which is more efficient

We can improve the efficiency of our labelling algorithm by using a cleverer way of handling EG. Instead of using EX, EU and AF as the adequate set, we use EX, EU and EG instead. For EX and EU we do as before (but take care to search the model by backwards breadth-first searching, for this ensures that we won't have to pass over any node twice). For the $EG\,\psi$ case:

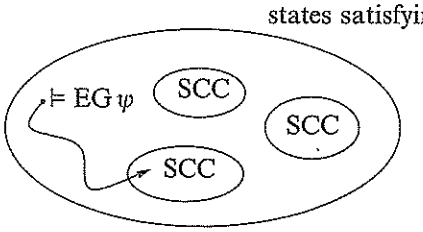

Fig. 3.16. A better way of handling EG.

- restrict the graph to states satisfying ψ, i.e. delete all other states and their transitions;
- find the maximal *strongly connected components* (SCCs); these are maximal regions of the state space in which every state is linked with (= has a finite path to) every other one in that region.
- use backwards breadth-first searching on the restricted graph to find any state that can reach an SCC; see Figure 3.16.

The complexity of this algorithm is $O(f \cdot (V + E))$, i.e. linear both in the size of the model and in the size of the formula.

Example 3.10 We applied the basic algorithm to our second model of mutual exclusion with the formula $\mathrm{E}[\neg c_2 \ \mathrm{U} \ c_1]$; see Figure 3.17. The algorithm labels all states which satisfy c_1 during phase 1 with $\mathrm{E}[\neg c_2 \ \mathrm{U} \ c_1]$. This labels s_2 and s_4. During phase 2, it labels all states which do not satisfy c_2 and have a successor state that is already labelled. This labels states s_1 and s_3. During phase 3, we label s_0 because it does not satisfy c_2 and has a successor state (s_1) which is already labelled. Thereafter, the algorithm terminates because no additional states get labelled: all unlabelled states either satisfy c_2, or must pass through such a state to reach a labelled state.

EXERCISES 3.7

* 1. Apply the labelling algorithm to check the formulas ϕ_1, ϕ_2, ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.12.
2. Apply the labelling algorithm to check the formulas ϕ_1, ϕ_2, ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.13.
3. Explain the construction of formula ϕ_4, used to express that the processes need not enter their critical section in strict sequence. Does it rely on the fact that the safety property ϕ_1 holds?
4. Inspecting the definition of the labelling algorithm, explain what

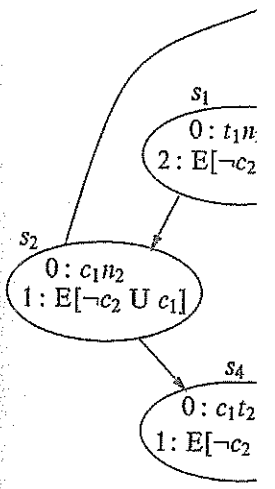

Fig. 3.17. An example
exclusion applied to t

happens if yo
model).

3.5.2 The

We present the pseu
in this chapter. The
CTL formula, i.e. a
The program SAT e
by means of the gr
important *precondit*
the program simply
$\mathrm{X}(\top \wedge \mathrm{EF}\, p_3)$, since
tree according to the
would make sure th
SAT. However, we n
preprocessing becaus

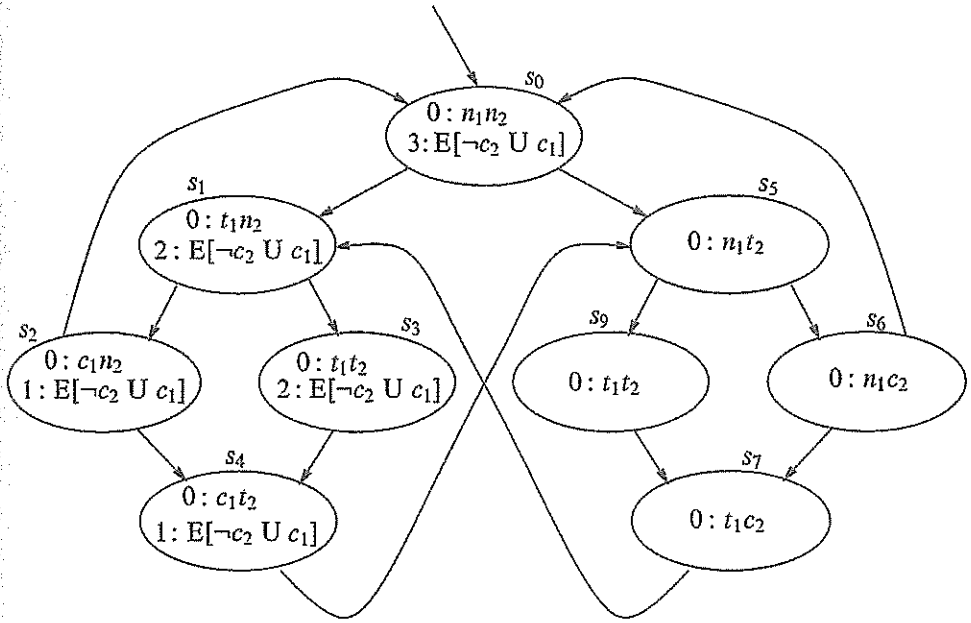

Fig. 3.17. An example run of the labelling algorithm in our second model of mutual exclusion applied to the formula E[$\neg c_2$ U c_1].

happens if you perform it on the formula $p \wedge \neg p$ (in any state, in any model).

—

3.5.2 *The pseudo-code of the model checking algorithm*

We present the pseudo-code for the basic labelling algorithm given earlier in this chapter. The main function SAT (for 'satisfies') takes as input a CTL formula, i.e. a formula over the Backus Naur form in Definition 3.1. The program SAT expects a parse tree of some CTL formula constructed by means of the grammar in Definition 3.1. This expectation reflects an important *precondition* on the correctness of the algorithm SAT. For example, the program simply would not know what to do with an input of the form $X(\top \wedge EF\, p_3)$, since this is not a CTL formula (it does not have a parse tree according to the grammar for CTL). Hence, we also need a parser that would make sure that only well-formed CTL formulas are being fed into SAT. However, we merely focus on the development of SAT without such preprocessing because this is the prime objective of this section.

The pseudo-code we write for SAT looks a bit like fragments of FORTRAN or Pascal code; we use functions with a keyword **return** that indicates which result the function should return. We will also use natural language to indicate the case analysis over the root node of the parse tree of ϕ. The declaration **local var** declares some fresh variables local to the current instance of the procedure in question, whereas **repeat until** executes the command which follows it repeatedly, until the condition becomes true. Additionally, we employ suggestive notation for the operations on sets, like intersection, set complement and so forth. In reality we would need an abstract data type, called SETS, together with implementations of these operations, but for now we are interested only in the mechanism in principle of the algorithm for SAT; any (correct and efficient) implementation of sets would do and we study such an implementation in Chapter 6. We assume that SAT has access to all the relevant parts of the model: S, $\rightarrow$ and L. In particular, we ignore the fact that SAT would require a description of $\mathcal{M}$ as input as well. We simply assume that SAT operates *directly* on any such given model. Note how SAT implicitly translates ϕ into an equivalent formula of the adequate set chosen.

The algorithm is presented in Figure 3.18 and its subfunctions in Figures 3.19-3.21. They use program variables X, Y, V and W of type SETS. The program for SAT simply handles the easy cases directly and passes more complicated cases on to special procedures, which in turn might call SAT *recursively* on subexpressions.

Of course, we still need to make sure that this algorithm is correct. This will be handled in Section 3.9.

3.5.3 *The 'state explosion' problem*

Although the labelling algorithm (with the clever way of handling EG) is linear in the size of the model, unfortunately the size of the model is itself exponential in the number of variables and the number of components of the system which execute in parallel. This means that, for example, adding a boolean variable to your program will *double* the complexity of verifying a property of it.

The tendency of the state space to become very large is known as the *state explosion* problem. A lot of research has gone into finding ways of overcoming it, including the use of:

- Efficient data structures, called *ordered binary decision diagrams* (OBDDs),

```
function SAT(φ)
 /* determines the se
begin
   case
      φ is ⊤ : return
      φ is ⊥ : return
      φ is atomic: r
      φ is ¬φ1 : ret
      φ is φ1 ∧ φ2 :
      φ is φ1 ∨ φ2 :
      φ is φ1 → φ2
      φ is AX φ1 : r
      φ is EX φ1 : r
      φ is A[φ1 U φ
      φ is E[φ1 U φ
      φ is EF φ1 : r
      φ is EG φ1 : r
      φ is AF φ1 : r
      φ is AG φ1 : r
   end case
end function
```

Fig. 3.18. The functic
of states satisfying th
respectively, if EX, E

```
function SAT_EX(φ)
 /* determines the set
local var X, Y
begin
   X := SAT(φ);
   Y := {s0 ∈ S | s0 -
   return Y
end
```

Fig. 3.19. The functio
Then, it looks backwa

which represent s
in Chapter 6 in d

- Abstraction: we
relevant to the fo
- Partial order redu
component traces
to be checked is c
the model-checkin

```
function SAT(φ)
 /* determines the set of states satisfying φ */
begin
   case
      φ is ⊤ : return S
      φ is ⊥ : return ∅
      φ is atomic: return {s ∈ S | φ ∈ L(s)}
      φ is ¬φ1 : return S − SAT(φ1)
      φ is φ1 ∧ φ2 : return SAT(φ1) ∩ SAT(φ2)
      φ is φ1 ∨ φ2 : return SAT(φ1) ∪ SAT(φ2)
      φ is φ1 → φ2 : return SAT(¬φ ∨ φ2)
      φ is AX φ1 : return SAT(¬EX ¬φ1)
      φ is EX φ1 : return SAT_EX(φ1)
      φ is A[φ1 U φ2] : return SAT(¬(E[¬φ2 U (¬φ1 ∧ ¬φ2)] ∨ EG ¬φ2))
      φ is E[φ1 U φ2] : return SAT_EU(φ1, φ2)
      φ is EF φ1 : return SAT(E(⊤ U φ1))
      φ is EG φ1 : return SAT(¬AF ¬φ1)
      φ is AF φ1 : return SAT_AF(φ1)
      φ is AG φ1 : return SAT(¬EF ¬φ1)
   end case
end function
```

Fig. 3.18. The function SAT. It takes a CTL formula as input and returns the set of states satisfying the formula. It calls the functions SAT_{EX}, SAT_{EU} and SAT_{AF}, respectively, if EX, EU or AF is the root of the input's parse tree.

```
function SAT_EX(φ)
 /* determines the set of states satisfying EX φ */
local var X, Y
begin
   X := SAT(φ);
   Y := {s0 ∈ S | s0 → s1 for some s1 ∈ X};
   return Y
end
```

Fig. 3.19. The function SAT_{EX}. It computes the states satisfying ϕ by calling SAT. Then, it looks backwards along $\rightarrow$ to find the states satisfying EX ϕ.

which represent *sets* of states instead of individual states. We study these in Chapter 6 in detail.

- Abstraction: we abstract away variables in the model which are not relevant to the formula being checked.
- Partial order reduction: for asynchronous systems, several interleavings of component traces may be equivalent as far as satisfaction of the formula to be checked is concerned. This can often substantially reduce the size of the model-checking problem.

```
function SAT_AF(φ)
 /* determines the set of states satisfying AF φ */
local var X, Y
begin
    X := S;
    Y := SAT(φ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ {s | for all s' with s → s' we have s' ∈ Y}
    end
    return Y
end
```

Fig. 3.20. The function $\mathrm{SAT}_{\mathrm{AF}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{AF}\,\phi$ in the manner described in the labelling algorithm.

```
function SAT_EU(φ, ψ)
 /* determines the set of states satisfying E[φ U ψ] */
local var W, X, Y
begin
    W := SAT(φ);
    X := S;
    Y := SAT(ψ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ (W ∩ {s | exists s' such that s → s' and s' ∈ Y}
    end
    return Y
end
```

Fig. 3.21. The function $\mathrm{SAT}_{\mathrm{EU}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{E}[\phi\ \mathrm{U}\ \psi]$ in the manner described in the labelling algorithm.

- Induction: model-checking systems with (e.g.) large numbers of identical, or similar, components can often be implemented by 'induction' on this number.
- Composition: break the verification problem down into several simpler verification problems.

The last four issues are beyond the scope of this book, but references may be found at the ends of Chapters 3 and 6.

EXERCISES 3.8

* 1. For mutual
processes to
that ϕ_4 is fa

2. Extend the
subformulas
[Question: v
like that for

* 3. Write the p
of deleting l

So far, this chapte
one continue in thi
checking is that it
implementations w
tion, we look at th
SMV ('symbolic m
models we have be
of CTL formulas o

SMV is freely dis
can be found at th
to SMV, but provid
reader should read

SMV takes as i
and some specifica
word 'true' *if the*
ing why the spe
program.

SMV programs
ming language C,
Modules can decl
give the `initial`
sion in terms of t
non-deterministic
ment at all). Non-
abstraction.

EXERCISES 3.8

* 1. For mutual exclusion, draw a transition system which forces the two processes to enter their critical section in strict sequence and show that ϕ_4 is false of its initial state.

2. Extend the pseudo-code of Section 3.5.2 so that it can deal with subformulas $\mathrm{AG}\,\psi_1$, without rewriting it in terms of other formulas. [Question: will your routine be more like the routine for AF, or more like that for EG on page 175? Why?]

* 3. Write the pseudo-code for $\mathrm{SAT}_{\mathrm{EG}}$, based on the description in terms of deleting labels given in Section 3.5.1.

3.6 The SMV system

So far, this chapter has been quite theoretical; and the sections after this one continue in this vein. However, one of the exciting things about model checking is that it is also a practical subject, for there are several efficient implementations which can check large systems in realistic time. In this section, we look at the best-known of the CTL model checkers, which is called SMV ('symbolic model verifier'). It provides a language for describing the models we have been drawing as diagrams and it directly checks the validity of CTL formulas on those models.

SMV is freely distributed and can be found on the internet. Further details can be found at the end of this chapter. We do not give a full introduction to SMV, but provide just enough information to give the flavour. The serious reader should read more about SMV in the references.

SMV takes as input a text consisting of a program describing a model and some specifications (CTL formulas). It produces as output either the word 'true' *if the specifications hold for all initial states*, or a trace showing why the specification is false for the model determined by our program.

SMV programs consist of one or more modules. As in the programming language C, or Java, one of the modules must be called `main`. Modules can declare variables and assign to them. Assignments usually give the `initial` value of a variable and its `next` value as an expression in terms of the current values of variables. This expression can be non-deterministic (denoted by several expressions in braces, or no assignment at all). Non-determinism is used to model the environment and for abstraction.