

3

Verification by model checking

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. 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 model \mathcal{M} for an appropriate logic. The specification is again represented by a formula ϕ and the verification method consists of computing

whether a model \mathcal{M} satisfies ϕ (written $\mathcal{M} \models \phi$). This computation is usually automatic for finite models.

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

Degree of automation. Approaches differ on how automatic the method is; the extremes are fully automatic and fully manual. Many of the computer-assisted techniques are 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 in the production cycle are less costly to rectify. (It is alleged that Intel lost millions of dollars by releasing their Pentium chip with the FDIV error.)

This chapter concerns a verification method called *model checking*. In 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-reproducible or not covered by test cases, so it is well worth having a verification technique that can help one to find them.

The Alloy system described in Chapter 2 is also an automatic, model-based, property-verification approach. The way models are used is slightly different, however. Alloy finds models which form counterexamples to assertions made by the user. Model checking starts with a model described by the user, and discovers whether hypotheses asserted by the user are valid on the model. If they are not, it can produce counterexamples, consisting of

execution traces. Another difference between Alloy and model checking is that model checking (unlike Alloy) focusses explicitly on temporal properties and the temporal evolution of systems.

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 set of paths, where a path is a sequence of time instances. *Branching-time* logics represent time as a tree, rooted at the present moment and branching out into the future. Branching time appears to make the non-deterministic nature of the future more explicit. 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.

Temporal logics have a dynamic aspect to them, since the truth of a formula is not fixed in a model, as it is in predicate or propositional logic, but depends on the time-point inside the model. In this chapter, we study a logic where time is linear, called *Linear-time Temporal Logic* (LTL), and another where time is branching, namely *Computation Tree Logic* (CTL). These logics have proven to be extremely fruitful in verifying hardware and communication protocols; and people are beginning to apply them 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 one of these logics, \mathcal{M} is an appropriate model of the system under consideration, s is a state of that model and \models is the underlying satisfaction relation.

Models like \mathcal{M} should not be confused 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.

3.2 Linear-time temporal logic

Linear-time temporal logic, or LTL for short, is a temporal logic, with connectives that allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or simply a path. In general, the future is not determined, so we consider several paths, representing different possible futures, any one of which might be the ‘actual’ path that is realised.

We work with a fixed set **Atoms** of atomic formulas (such as p, q, r, \dots , or p_1, p_2, \dots). These atoms stand for atomic facts which may hold of a system, like ‘*Printer Q5 is busy,*’ or ‘*Process 3259 is suspended,*’ or ‘*The content of register R1 is the integer value 6.*’ The choice of atomic descriptions obviously depends on our particular interest in a system at hand.

3.2.1 Syntax of LTL

Definition 3.1 Linear-time temporal logic (LTL) has the following syntax given in Backus Naur form:

$$\begin{aligned} \phi ::= & \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\ & \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi U \phi) \mid (\phi W \phi) \mid (\phi R \phi) \end{aligned} \quad (3.1)$$

where p is any propositional atom from some set \mathbf{Atoms} .

Thus, the symbols \top and \perp are LTL formulas, as are all atoms from \mathbf{Atoms} ; and $\neg\phi$ is an LTL formula if ϕ is one, etc. The connectives X , F , G , U , R , and W are called *temporal connectives*. X means ‘neXt state,’ F means ‘some Future state,’ and G means ‘all future states (Globally).’ The next three, U , W and R are called ‘Until,’ ‘Release’ and ‘Weak-until’ respectively. We will look at the precise meaning of all these connectives in the next section; for now, we concentrate on their syntax.

Here are some examples of LTL formulas:

- $((F p) \wedge (G q)) \rightarrow (p W r)$
- $(F(p \rightarrow (G r)) \vee ((\neg q) U p))$, the parse tree of this formula is illustrated in Figure 3.1.
- $(p W (q W r))$
- $((G(F p)) \rightarrow (F(q \vee s)))$.

It’s boring to write all those brackets, and makes the formulas hard to read. Many of them can be omitted without introducing ambiguities; for example, $(p \rightarrow (F q))$ could be written $p \rightarrow F q$ without ambiguity. Others, however, are required to resolve ambiguities. In order to omit some of those, we assume similar binding priorities for the LTL connectives to those we assumed for propositional and predicate logic.

Convention 3.2 The unary connectives (consisting of \neg and the temporal connectives X , F and G) bind most tightly. Next in the order come U , R and W ; then come \wedge and \vee ; and after that comes \rightarrow .

These binding priorities allow us to drop some brackets without introducing ambiguity. The examples above can be written

- $F p \wedge G q \rightarrow p W r$
- $F(p \rightarrow G r) \vee \neg q U p$
- $p W (q W r)$
- $G F p \rightarrow F(q \vee s)$.

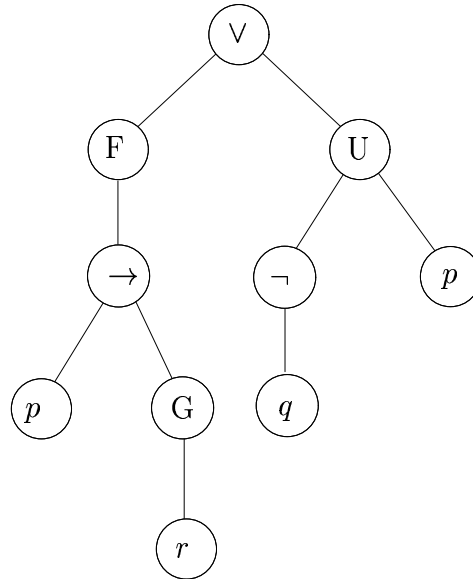


Fig. 3.1. The parse tree of $(F(p \rightarrow G r) \vee (\neg q U p))$.

The brackets we retained were in order to override the priorities of Convention 3.2, or to disambiguate cases which the convention does not resolve. For example, with no brackets at all, the second formula would become $F p \rightarrow G r \vee \neg q U p$, corresponding to the parse tree of Figure 3.2, which is quite different.

The following are *not* well-formed formulas:

- $U r$ — since U is binary, not unary
- $p G q$ — since G is unary, not binary.

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

The subformulas of $p W (q U r)$, e.g., are p , q , r , $q U r$ and $p W (q U r)$.

3.2.2 Semantics of LTL

The kinds of systems we are interested in verifying using LTL may be modelled as transition systems. A transition system models a system by means of *states* (static structure) and *transitions* (dynamic structure). More formally:

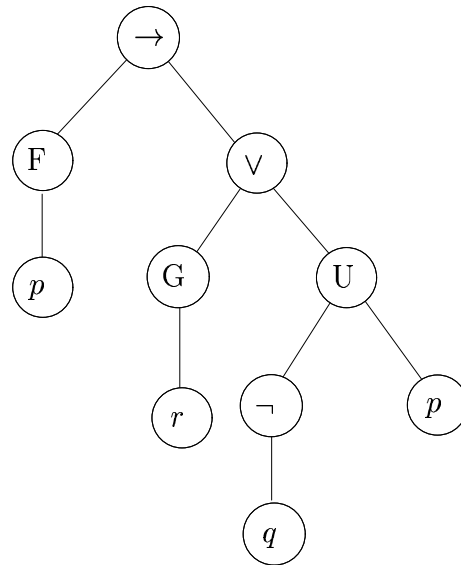


Fig. 3.2. The parse tree of $Fp \rightarrow Gr \vee \neg q U p$, assuming binding priorities of Convention 3.2.

Definition 3.4 A transition system $\mathcal{M} = (S, \rightarrow, L)$ 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 $L: S \rightarrow \mathcal{P}(\text{Atoms})$.

Transition systems are also simply called *models* in this chapter. So a model has a collection of states S , a relation \rightarrow , saying how the system can move from state to state, and, associated with each state s , one has the set of atomic propositions $L(s)$ which are true at that particular state. We write $\mathcal{P}(\text{Atoms})$ 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) transition system \mathcal{M} using directed graphs whose nodes (which we call 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

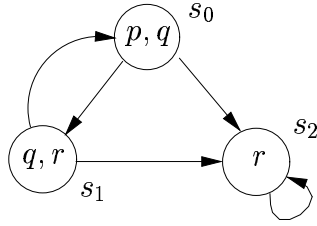


Fig. 3.3. A concise representation of a transition system $\mathcal{M} = (S, \rightarrow, L)$ as a directed graph. We label state s with l iff $l \in L(s)$.

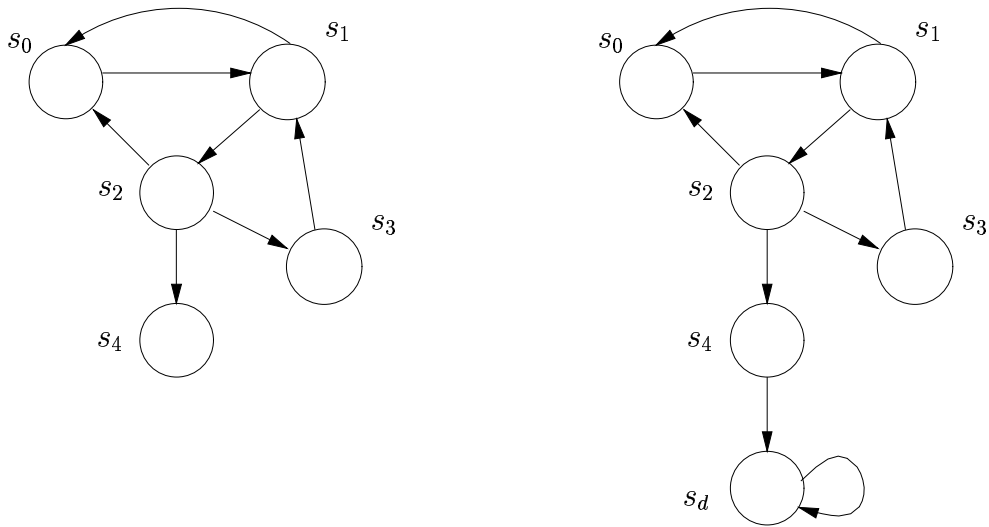


Fig. 3.4. On the left, we have a system with a state s_4 that does not have any further transitions. On the right, we expand that system with a ‘deadlock’ state s_d such that no state can deadlock; of course, it is then our understanding that reaching the ‘deadlock’ state s_d corresponds to deadlock in the original system.

if $L(s_0) = \{p, q\}$, $L(s_1) = \{q, r\}$ and $L(s_2) = \{r\}$, then we can condense all this information into Figure 3.3. We prefer to present models by means of such pictures whenever that is feasible.

The requirement in Definition 3.4 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 a technical convenience, and in fact it does not represent any real restriction on the systems we can model. If a system did deadlock, we could 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 3.4 for such an example.

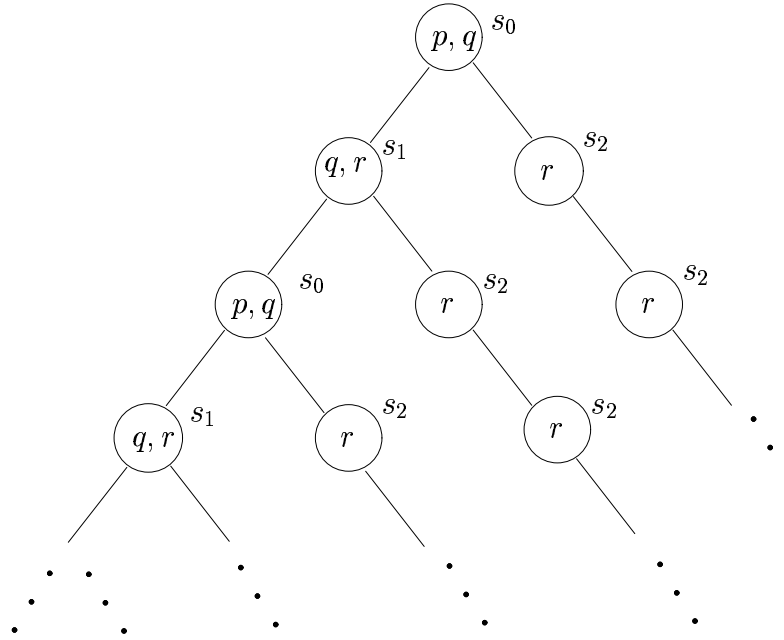


Fig. 3.5. Unwinding the system of Figure 3.3 as an infinite tree of all computation paths beginning in a particular state.

Definition 3.5 A path in a model $\mathcal{M} = (S, \rightarrow, L)$ is an infinite sequence of states s_1, s_2, s_3, \dots in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$. We write the path as $s_1 \rightarrow s_2 \rightarrow \dots$.

Consider the path $\pi = s_1 \rightarrow s_2 \rightarrow \dots$. It represents a possible future of our system: first it is in state s_1 , then it is in state s_2 , and so on. We write π^i for the suffix starting at s_i , e.g. π^3 is $s_3 \rightarrow s_4 \rightarrow \dots$.

It is useful to visualise all possible computation paths from a given state s by unwinding the transition system to obtain an infinite computation tree. For example, if we unwind the state graph of Figure 3.3 for the designated starting state s_0 , then we get the infinite tree in Figure 3.5. The execution paths of a model \mathcal{M} are explicitly represented in the tree obtained by unwinding the model.

Definition 3.6 Let $\mathcal{M} = (S, \rightarrow, L)$ be a model and $\pi = s_1 \rightarrow \dots$ be a path in \mathcal{M} . Whether π satisfies an LTL formula is defined by the satisfaction relation \models as follows:

1. $\pi \models \top$
2. $\pi \not\models \perp$

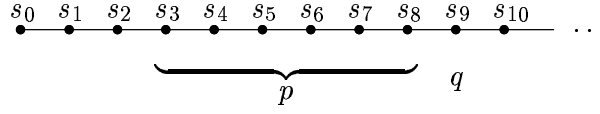


Fig. 3.6. An illustration of the meaning of Until in the semantics of LTL. Suppose p is satisfied at (and only at) $s_3, s_4, s_5, s_6, s_7, s_8$ and q is satisfied at (and only at) s_9 . Only the states s_3 to s_9 each satisfy $p \text{ U } q$ along the path shown.

3. $\pi \models p$ iff $p \in L(s_1)$
4. $\pi \models \neg\phi$ iff $\pi \not\models \phi$
5. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
6. $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
7. $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$
8. $\pi \models X\phi$ iff $\pi^2 \models \phi$
9. $\pi \models G\phi$ iff, for all $i \geq 1$, $\pi^i \models \phi$
10. $\pi \models F\phi$ iff there is some $i \geq 1$ such that $\pi^i \models \phi$
11. $\pi \models \phi \text{ U } \psi$ iff there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, \dots, i - 1$ we have $\pi^j \models \phi$
12. $\pi \models \phi \text{ W } \psi$ iff either there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, \dots, i - 1$ we have $\pi^j \models \phi$; or for all $k \geq 1$ we have $\pi^k \models \phi$
13. $\pi \models \phi \text{ R } \psi$ iff either there is some $i \geq 1$ such that $\pi^i \models \phi$ and for all $j = 1, \dots, i$ we have $\pi^j \models \psi$, or for all $k \geq 1$ we have $\pi^k \models \psi$.

Clauses 1 and 2 reflect the facts that \top is always true, and \perp is always false. Clauses 3–7 are similar to the corresponding clauses we saw in propositional logic. Clause 8 removes the first state from the path, in order to create a path starting at the ‘next’ (second) state.

Notice that clause 3 means that atoms are evaluated in the first state along the path in consideration. However, that doesn’t mean that all the atoms occurring in an LTL formula refer to the first state of the path; if they are in the scope of a temporal connective, e.g. in $G(p \rightarrow Xq)$, then the calculation of satisfaction involves taking suffices of the path in consideration, and the atoms refer to the first state of those suffices.

Let’s now look at clauses 11–13, which deal with the binary temporal connectives. U , which stands for ‘until,’ is the most commonly encountered one of these. The formula $\phi_1 \text{ U } \phi_2$ holds on a path if it is the case that ϕ_1 holds continuously *until* ϕ_2 holds. Moreover, $\phi_1 \text{ U } \phi_2$ actually demands that ϕ_2 *does* hold in some future state. See Figure 3.6 for illustration: each of the states s_3 to s_9 satisfies $p \text{ U } q$ along the path shown, but s_0 to s_2 don’t.

The other binary connectives are W , standing for ‘weak until,’ and R , standing for ‘release.’ Weak-until is just like U , except that $\phi \text{ W } \psi$ does not

require that ψ is eventually satisfied along the path in question, which is required by $\phi \text{ U } \psi$. Release R is the dual of U; that is, $\phi \text{ R } \psi$ is equivalent to $\neg(\neg\phi \text{ U } \neg\psi)$. It is called ‘release’ because clause 11 determines that ψ must remain true up to and including the moment when ϕ becomes true (if there is one); ϕ ‘releases’ ψ . R and W are actually quite similar; the differences are that they swap the roles of ϕ and ψ , and the clause for W has an $i - 1$ where R has i . Since they are similar, why do we need both? We don’t; they are interdefinable, as we will see later. However, it’s useful to have both. R is useful because it is the dual of U, while W is useful because it is a weak form of U.

Note that neither the strong version (U) or the weak version (W) of until says anything about what happens after the until has been realised. This is in contrast with some of the readings of ‘until’ in natural language. 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. We could express the sentence about smoking by combining U with other connectives; for example, by asserting that it was once true that $s \text{ U } (t \wedge \text{G } \neg s)$, where s represents ‘I smoke’ and t represents ‘I am 22.’

Remark 3.7 Notice that, in clauses 9–13 above, the future includes the present. 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. As an exercise, you may consider developing a version of LTL in which the future excludes the present. A consequence of adopting the convention that the future shall include the present is that the formulas $\text{G } p \rightarrow p$, $p \rightarrow q \text{ U } p$ and $p \rightarrow \text{F } p$ are true in every state of every model.

So far we have defined a satisfaction relation between paths and LTL formulas. However, to verify systems, we would like to say that a model as a whole satisfies an LTL formula. This is defined to hold whenever *every* possible execution path of the model satisfies the formula.

Definition 3.8 Suppose $\mathcal{M} = (S, \rightarrow, L)$ is a model, $s \in S$, and ϕ an LTL formula. We write $\mathcal{M}, s \models \phi$ if, for every execution path π of \mathcal{M} starting at s , we have $\pi \models \phi$.

If \mathcal{M} is clear from the context, we may abbreviate $\mathcal{M}, s \models \phi$ by $s \models \phi$.

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. Later in this chapter, we will examine algorithms which implement this calculation. Let us now look at some example checks for the system in Figures 3.3 and 3.5.

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 : $\pi \models p \wedge q$ for *every* path π beginning in 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. $\mathcal{M}, s_0 \models Xr$ holds since all paths from s_0 have either s_1 or s_2 as their next state, and each of those states satisfies r .
5. $\mathcal{M}, s_0 \models X(q \wedge r)$ does not hold since we have the rightmost computation path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ in Figure 3.5, whose second node s_2 contains r , but not q .
6. $\mathcal{M}, s_0 \models G \neg(p \wedge r)$ holds since all computation paths beginning in s_0 satisfy $G \neg(p \wedge r)$, i.e. they satisfy $\neg(p \wedge r)$ in each state along the path. Notice that $G \phi$ holds in a state if, and only if, ϕ holds in all states reachable from the given state.
7. For similar reasons, $\mathcal{M}, s_2 \models Gr$ holds (note the s_2 instead of s_0).
8. For any state s of \mathcal{M} , we have $\mathcal{M}, s \models F(\neg q \wedge r) \rightarrow FG r$. This says that if any path π beginning in s gets to a state satisfying $(\neg q \wedge r)$, then the path π satisfies $FG r$. Indeed this is true, since if the path has a state satisfying $(\neg q \wedge r)$ then (since that state must be s_2) the path does satisfy $FG r$. Notice what $FG r$ says about a path: eventually, you have continuously r .
9. The formula $GF p$ expresses that p occurs along the path in question infinitely often. Intuitively, it's saying: no matter how far along the path you go (that's the G part) you will find you still have a p in front of you (that's the F part). For example, the path $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$ satisfies $GF p$. But the path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ doesn't.
10. In our model, if a path from s_0 has infinitely many ps on it then it must be the path $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$, and in that case it also has infinitely many rs on it. So, $\mathcal{M}, s_0 \models GF p \rightarrow GF r$. But it is not the case the other way around! It is not the case that $\mathcal{M}, s_0 \models GF r \rightarrow GF p$, because we can find a path from s_0 which has infinitely many rs but only one p .

3.2.3 Practical patterns of specifications

What kind of practically relevant properties can we check with formulas of LTL? 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 impossible to get to a state where started holds, but ready does not hold:
 $G\neg(\text{started} \wedge \neg\text{ready})$
 The negation of this formula expresses that it *is possible* to get to such a state, but this is only so if interpreted on paths ($\pi \models \phi$). We cannot assert such a possibility if interpreted on states ($s \models \phi$) since we cannot express the existence of paths; for that interpretation, the negation of the formula above asserts that *all* paths will eventually get to such a state.
- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
 $G(\text{requested} \rightarrow F \text{acknowledged})$.
- A certain process is enabled infinitely often on every computation path:
 $G F \text{enabled}$.
- Whatever happens, a certain process will eventually be permanently deadlocked:
 $F G \text{deadlock}$.
- If the process is enabled infinitely often, then it runs infinitely often.
 $G F \text{enabled} \rightarrow G F \text{running}$.
- An upwards travelling elevator at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
 $G(\text{floor}=2 \wedge \text{direction}=\text{up} \wedge \text{ButtonPressed5} \rightarrow (\text{direction}=\text{up} \cup \text{floor}=5))$
 Here, our atomic descriptions are boolean expressions built from system variables, e.g. $\text{floor}=2$.

There are some things which are *not* possible to say in LTL, however. One big class of such things are statements which assert the existence of a path, such as these ones:

- From any state *it is possible* to get to a restart state (i.e. there is a path from all states to a state satisfying restart).
- The elevator *can* remain idle on the third floor with its doors closed (i.e. from the state in which it is on the third floor, there is a path along which it stays there).

LTL can't express these because it cannot directly assert the existence of paths. In Section 3.4, we look at Computation Tree Logic (CTL) which has operators for quantifying over paths, and can express these properties.

3.2.4 Important equivalences between LTL formulas

Definition 3.9 We say that two LTL formulas ϕ and ψ are semantically equivalent, or simply equivalent, writing $\phi \equiv \psi$, if for all models \mathcal{M} and all paths π in \mathcal{M} : $\pi \models \phi$ iff $\pi \models \psi$.

The equivalence of ϕ and ψ means that ϕ and ψ are semantically interchangeable. If ϕ is a subformula of some bigger formula χ , and $\psi \equiv \phi$, then we can make the substitution of ψ for ϕ in χ without changing the meaning of χ . In propositional logic, we saw that \wedge and \vee are duals of each other, meaning that if you push a \neg past a \wedge , it becomes a \vee , and vice versa:

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi \qquad \neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi .$$

(Because \wedge and \vee are binary, pushing a negation downwards in the parse tree past one of them also has the effect of duplicating that negation.)

Similarly, F and G are duals of each other, and X is dual with itself:

$$\neg G \phi \equiv F \neg\phi \qquad \neg F \phi \equiv G \neg\phi \qquad \neg X \phi \equiv X \neg\phi .$$

Also U and R are duals of each other:

$$\neg(\phi U \psi) \equiv \neg\phi R \neg\psi \qquad \neg(\phi R \psi) \equiv \neg\phi U \neg\psi .$$

We should give formal proofs of these equivalences. But they are easy, so we leave them as an exercise to the reader. 'Morally' there ought to be a dual for W, and you can invent one if you like. Work out what it might mean, and then pick a symbol based on the first letter of the meaning. However, it might not be very useful.

It's also the case that F distributes over \vee and G over \wedge , i.e.

$$\begin{aligned} F(\phi \vee \psi) &\equiv F\phi \vee F\psi \\ G(\phi \wedge \psi) &\equiv G\phi \wedge G\psi . \end{aligned}$$

Compare this with the quantifier equivalences in Section 2.3.2. But F does *not* distribute over \wedge . What this means is that there is a model with a path which distinguishes $F(\phi \wedge \psi)$ and $F\phi \wedge F\psi$, for some ϕ, ψ . Take the path $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$ from the system of Figure 3.3, for example; it satisfies $Fp \wedge Fr$ but it doesn't satisfy $F(p \wedge r)$.

Here are two more equivalences in LTL:

$$F \phi \equiv \top U \phi \qquad G \phi \equiv \perp R \phi .$$

The first one exploits the fact that the clause for Until states two things: the second formula ϕ must become true; and until then, the first formula \top must hold. So, if we put ‘no constraint’ for the first formula, it boils down to asking that the second formula holds, which is what F asks. (The formula \top represent ‘no constraint.’ If you ask me to bring it about that \top holds, I need do nothing, it enforces no constraint. In the same sense, \perp is ‘every constraint.’ If you ask me to bring it about that \perp holds, I’ll have to meet every constraint there is, which is impossible.)

The second formula, that $G \phi \equiv \perp R \phi$, can be obtained from the first by putting a \neg in front of each side, and applying the duality rules. Another more intuitive way of seeing this is to recall the meaning of ‘release:’ \perp releases ϕ , but \perp will never be true, so ϕ doesn’t get released.

Another pair of equivalences relates the strong and weak versions of Until, U and W . Strong until may be seen as weak until plus the constraint that the eventuality must actually occur:

$$\phi U \psi \equiv \phi W \psi \wedge F \psi . \tag{3.2}$$

To prove equivalence (3.2), suppose first that a path satisfies $\phi U \psi$. Then, from clause 11, we have $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, \dots, i - 1$ we have $\pi^j \models \phi$. From clause 12, this proves $\phi W \psi$, and from clause 10 it proves $F \psi$. Thus for all paths π , if $\pi \models \phi U \psi$ then $\pi \models \phi W \psi \wedge F \psi$. As an exercise, the reader can prove it the other way around.

Writing W in terms of U is also possible: W is like U but also allows the possibility of the eventuality never occurring:

$$\phi W \psi \equiv \phi U \psi \vee G \phi . \tag{3.3}$$

Inspection of clauses 12 and 13 reveals that R and W are rather similar. The differences are that they swap the roles of their arguments ϕ and ψ ; and the clause for W has an $i - 1$ where R has i . Therefore, it is not surprising that they are expressible in terms of each other, as follows:

$$\phi W \psi \equiv \psi R (\phi \vee \psi) \tag{3.4}$$

$$\phi R \psi \equiv \psi W (\phi \wedge \psi) . \tag{3.5}$$

3.2.5 Adequate sets of connectives for LTL

Recall that $\phi \equiv \psi$ holds iff any path in any transition system which satisfies ϕ also satisfies ψ , and vice versa. As in propositional logic, there is some

redundancy among the connectives. For example, in Chapter 1 we saw that the set $\{\perp, \wedge, \neg\}$ forms an adequate set of connectives, since the other connectives \vee, \rightarrow, \top , etc., can be written in terms of those three.

Small adequate sets of connectives also exist in LTL. Here is a summary of the situation.

- X is completely orthogonal to the other connectives. That is to say, its presence doesn't help in defining any of the other ones in terms of each other. Moreover, X cannot be derived from any combination of the others.
- Each of the sets $\{U, X\}$, $\{R, X\}$, $\{W, X\}$ is adequate. To see this, we note that
 - R and W may be defined from U, by the duality $\phi R \psi \equiv \neg(\neg\phi U \neg\psi)$ and equivalence (3.4) followed by the duality, respectively.
 - U and W may be defined from R, by the duality $\phi U \psi \equiv \neg(\neg\phi R \neg\psi)$ and equivalence (3.4), respectively.
 - R and U may be defined from W, by equivalence (3.5) and the duality $\phi U \psi \equiv \neg(\neg\phi R \neg\psi)$ followed by equivalence (3.5).

Sometimes it is useful to look at adequate sets of connectives which do not rely on the availability of negation. That's because it is often convenient to assume formulas are written in negation-normal form, where all the negation symbols are applied to proposition atoms (i.e. they are near the leaves of the parse tree). In this case, these sets are adequate for the fragment without X, and no strict subset is: $\{U, R\}$, $\{U, W\}$, $\{U, G\}$, $\{R, F\}$, $\{W, F\}$. But $\{R, G\}$ and $\{W, G\}$ are not adequate. Note that one cannot define G with $\{U, F\}$, and one cannot define F with $\{R, G\}$ or $\{W, G\}$.

We finally state and prove a useful equivalence about U.

Theorem 3.10 The equivalence $\phi U \psi \equiv \neg(\neg\psi U (\neg\phi \wedge \neg\psi)) \wedge F \psi$ holds for all LTL formulas ϕ and ψ .

Proof: Take any path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ in any model.

First, suppose $s_0 \models \phi U \psi$ holds. Let n be the smallest number such that $s_n \models \psi$; such a number has to exist since $s_0 \models \phi U \psi$; then, for each $k < n$, $s_k \models \phi$. We immediately have $s_0 \models F \psi$, so it remains to show $s_0 \models \neg(\neg\psi U (\neg\phi \wedge \neg\psi))$, which, if we expand, means:

- (*) for each $i > 0$, if $s_i \models \neg\phi \wedge \neg\psi$, then there is some $j < i$ with $s_j \models \psi$.

Take any $i > 0$ with $s_i \models \neg\phi \wedge \neg\psi$; $i > n$, so we can take $j \stackrel{\text{def}}{=} n$ and have $s_j \models \psi$.

Conversely, suppose $s_0 \models \neg(\neg\psi \cup (\neg\phi \wedge \neg\psi)) \wedge F\psi$ holds; we prove $s_0 \models \phi \cup \psi$. Since $s_0 \models F\psi$, we have a minimal n as before. We show that, for any $i < n$, $s_i \models \phi$. Suppose $s_i \models \neg\phi$; since n is minimal, we know $s_i \models \neg\psi$, so by (*) there is some $j < i < n$ with $s_j \models \psi$, contradicting the minimality of n . \square

3.3 Model checking: systems, tools, properties

3.3.1 Example: mutual exclusion

Let us now look at a larger example of verification using LTL, 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: Only one process is 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 requests 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 access to the shared resource more often than others, we should make sure our protocol has the property:

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

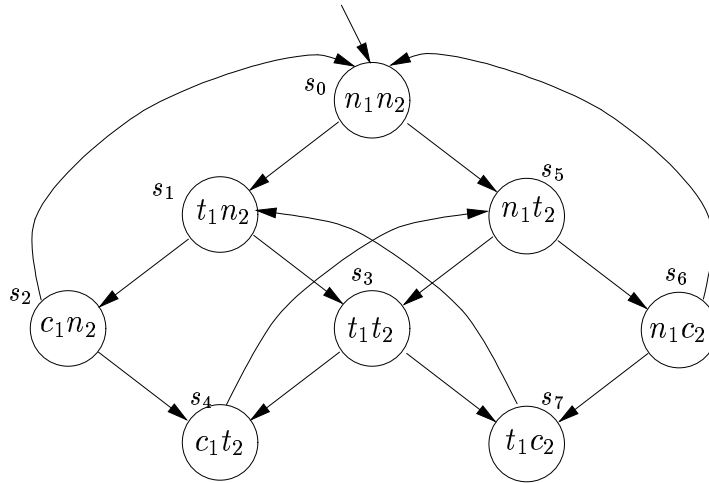


Fig. 3.7. A first-attempt model for mutual exclusion.

3.3.1.1 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 $n \rightarrow t \rightarrow c \rightarrow n \rightarrow \dots$, but the two processes interleave with each other. Consider the protocol given by the transition system \mathcal{M} in Figure 3.7. (As usual, we write $p_1 p_2 \dots p_m$ in a node s to denote that p_1, p_2, \dots, p_m are the only propositional atoms true at s .) 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*). At each step, an (unspecified) scheduler determines which process may run. 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 would like to check the four properties by first describing them as temporal logic formulas. Unfortunately, they are not all expressible as LTL formulas. Let us look at them case-by-case.

Safety: This is expressible in LTL, as $G \neg(c_1 \wedge c_2)$. Clearly, $G \neg(c_1 \wedge c_2)$ is satisfied in the initial state (indeed, in every state).

Liveness: This is also expressible: $G(t_1 \rightarrow F c_1)$. However, it is *not* satisfied by the initial state, for we can find path starting at the initial state along which there is a state, namely s_1 , in which t_1 is true but from there along the path c_1 is false. The path in question is $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow s_1 \rightarrow s_3 \rightarrow s_7 \dots$ on which c_1 is always false.

Non-blocking: Let's just consider process 1. We would like to express the property as: for every state satisfying n_1 , there is a successor satisfying t_1 . Unfortunately, this existence quantifier on paths ('there is a successor satisfying ...') cannot be expressed in LTL. It can be expressed in the logic CTL, which we will turn to in the next section (for the impatient, see page 224).

No strict sequencing: We might consider expressing this as saying: there is a path with two distinct states satisfying c_1 such that no state in between them has that property. However, we cannot express 'there exists a path,' so let us consider the complement formula instead. The complement says that all paths having a c_1 period which ends cannot have a further c_1 state until a c_2 state occurs. We write this as: $G(c_1 \rightarrow c_1 W (\neg c_1 \wedge \neg c_1 W c_2))$. This says that anytime we get into a c_1 state, either that condition persists indefinitely, or it ends with a non- c_1 state and in that case there is no further c_1 state unless and until we obtain a c_2 state.

This formula is false, as exemplified by the path $s_0 \rightarrow s_5 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_3 \rightarrow s_4 \dots$. Therefore the original condition expressing that strict sequencing need not occur, is true.

Before further considering the mutual exclusion example, some comments about expressing properties in LTL are appropriate. Notice that in the no-strict-sequencing property, we overcame the problem of not being able to express the existence of paths by instead expressing the complement property, which of course talks about all paths. Then we can perform our check, and simply reverse the answer; if the complement property is false, we declare our property to be true, and vice versa.

Why was that tactic not available to us to express the non-blocking property? The reason is that it says: every path to a n_1 state may be continued by a one-step path to a t_1 state. The presence of both universal and existential quantifiers is the problem. In the no-strict-sequencing property, we had only an existential quantifier; thus, taking the complement property turned it into a universal path quantifier, which can be expressed in LTL. But where we have alternating quantifiers, taking the complement property doesn't help in general.

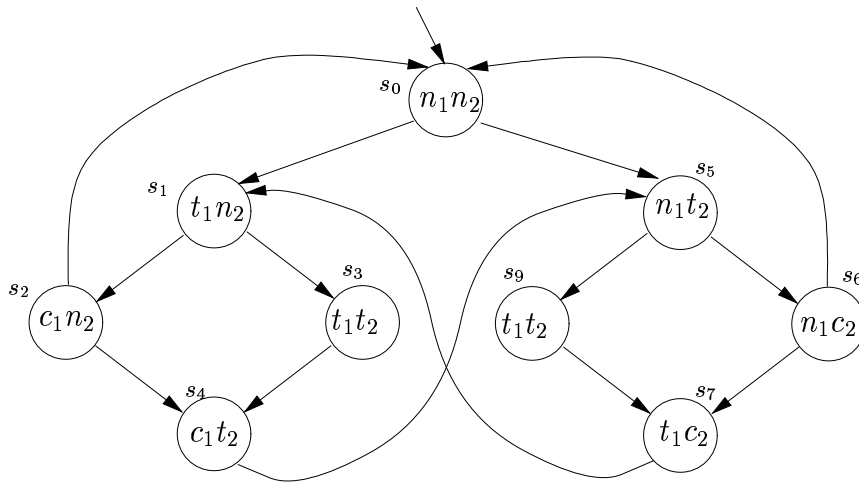


Fig. 3.8. A second-attempt model for mutual exclusion. There are now two states representing t_1t_2 , namely s_3 and s_9 .

Let's go back to the mutual exclusion example. 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.

3.3.1.2 The second modelling attempt

The two states s_3 and s_9 in Figure 3.8 both correspond to the state s_3 in our first modelling attempt. They both record that the two 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_1t_2 ; the definition of transition systems 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 3.11 The four properties of safety, liveness, non-blocking and no strict sequencing are satisfied by the model in Figure 3.8. (Since the non-blocking property has not yet been written in temporal logic, we can only check it informally.)

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_4 , or s_7 , to itself, we will again violate liveness. This problem will be solved later in this chapter when we consider ‘fairness constraints’ (Section 3.6.2).

3.3.2 The NuSMV model checker

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 NuSMV model-checking system. NuSMV stands for ‘New Symbolic Model Verifier.’ NuSMV is an Open Source product, is actively supported and has a substantial user community. For details on how to obtain it, see the bibliographic notes at the end of the chapter.

NuSMV (sometimes called simply SMV) provides a language for describing the models we have been drawing as diagrams and it directly checks the validity of LTL (and also CTL) formulas on those models. SMV takes as input a text consisting of a program describing a model and some specifications (temporal logic formulas). It produces as output either the word ‘true’ if the specifications hold, or a trace showing why the specification is false for the model represented 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.

The following input to SMV:

```

MODULE main
VAR
  request : boolean;
  status : {ready,busy};
ASSIGN
  init(status) := ready;
  next(status) := case
    request : busy;
    1 : {ready,busy};

```

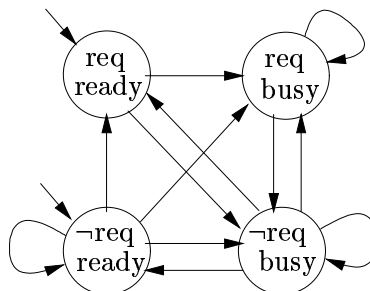


Fig. 3.9. The model corresponding to the SMV program in the text.

```
esac;
```

```
LTLSPEC
```

```
G(request -> F status=busy)
```

consists of a program and a specification. The program has two variables, **request** of type `boolean` and **status** of enumeration type `{ready, busy}`: 0 denotes ‘false’ and 1 represents ‘true.’ The initial and subsequent values of variable **request** are not determined within this program; this conservatively models that these values are determined by an external environment. This under-specification of **request** implies that the value of variable **status** is partially determined: initially, it is ready; and it becomes busy whenever **request** is true. If **request** is false, the next value of **status** is not determined.

Note that the case 1: signifies the default case, and that case statements are evaluated from the top down: if several expressions to the left of a ‘:’ are true, then the command corresponding to the first, top-most true expression will be executed. The program therefore denotes the transition system shown in Figure 3.9; there are four states, each one corresponding to a possible value of the two binary variables. Note that we wrote ‘busy’ as a shorthand for ‘**status=busy**’ and ‘req’ for ‘**request** is true.’

It takes a while to get used to the syntax of SMV and its meaning. Since variable **request** functions as a genuine environment in this model, the program and the transition system are *non-deterministic*: i.e. the ‘next state’ is not uniquely defined. Any state transition based on the behaviour of **status** comes in a pair: to a successor state where **request** is false, or true, respectively. For example, the state ‘¬req, busy’ has four states it can move to (itself and three others).

LTL specifications are introduced by the keyword `LTLSPEC` and are simply LTL formulas. Notice that SMV uses `&`, `|`, `->` and `!` for \wedge , \vee , \rightarrow and

\neg , respectively, since they are available on standard keyboards. We may easily verify that the specification of our module `main` holds of the model in Figure 3.9.

3.3.2.1 Modules in SMV

SMV supports breaking a system description into several *modules*, to aid readability and to verify interaction properties. A module is instantiated when a variable having that module name as its type is declared. This defines a set of variables, one for each one declared in the module description. In the example below, which is one of the ones distributed with SMV, a counter which repeatedly counts from 000 through to 111 is described by three single-bit counters. The module `counter_cell` is instantiated three times, with the names `bit0`, `bit1` and `bit2`. The counter module has one formal parameter, `carry_in`, which is given the actual value 1 in `bit0`, and `bit0.carry_out` in the instance `bit1`. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`. Note that we use the period `.` in `m.v` to access the variable `v` in module `m`. This notation is also used by Alloy (see Chapter 2) and a host of programming languages to access fields in record structures, or methods in objects. The keyword `DEFINE` is used to assign the expression `value & carry_in` to the symbol `carry_out` (such definitions are just a means for referring to the current value of a certain expression).

```

MODULE main
VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
LTLSPEC
    G F bit2.carry_out

MODULE counter_cell(carry_in)
VAR
    value : boolean;
ASSIGN
    init(value) := 0;
    next(value) := (value + carry_in) mod 2;
DEFINE
    carry_out := value & carry_in;

```

The effect of the `DEFINE` statement could have been obtained by declaring a new variable and assigning its value thus:

```

VAR
  carry_out : boolean;
ASSIGN
  carry_out := value & carry_in;

```

Notice that, in this assignment, the *current* value of the variable is assigned. Defined symbols are usually preferable to variables, since they don't increase the state space by declaring new variables. However, they cannot be assigned non-deterministically since they refer only to another expression.

3.3.2.2 Synchronous and asynchronous composition

By default, modules in SMV are composed *synchronously*: this means that there is a global clock and, each time it ticks, each of the modules executes in parallel. By use of the `process` keyword, it is possible to compose the modules asynchronously. In that case, they run at different 'speeds,' interleaving arbitrarily. At each tick of the clock, *one* of them is non-deterministically chosen and executed for one cycle. Asynchronous interleaving composition is useful for describing communication protocols, asynchronous circuits and other systems whose actions are not synchronised to a global clock.

The bit counter above is synchronous, whereas the examples below of mutual exclusion and the alternating bit protocol are asynchronous.

3.3.3 Running NuSMV

The normal use of NuSMV is to run it in batch mode, from a Unix shell or command prompt in Windows. The command line

```
NuSMV counter3.smv
```

will analyse the code in the file `counter3.smv` and report on the specifications it contains. One can also run NuSMV interactively. In that case, the command line

```
NuSMV -int counter3.smv
```

enters NuSMV's command-line interpreter. From there, there is a variety of commands you can use which allow you to compile the description and run the specification checks, as well as inspect partial results and set various parameters. See the NuSMV user manual for more details.

NuSMV also supports *bounded model checking*, invoked by the command-line option `-bmc`. Bounded model checking looks for counterexamples in order of size, starting with counterexamples of length 1, then 2, etc., up to a given threshold (10 by default). Note that bounded model checking

is incomplete: failure to find a counterexample does not mean that there is none, but only that there is none of length up to the threshold. For related reasons, this incompleteness features also in Alloy and its constraint analyzer. Thus, while a negative answer can be relied on (if NuSMV finds a counterexample, it is valid), a positive one cannot. References on bounded model checking can be found in the bibliographic notes on page 264. Later on, we use bounded model checking to prove the optimality of a scheduler.

3.3.4 Mutual exclusion revisited

Figure 3.10 gives the SMV code for a mutual exclusion protocol. This code consists of two modules, `main` and `prc`. The module `main` has the variable `turn`, which determines whose turn it is to enter the critical section if both are trying to enter (recall the discussion about the states s_3 and s_9 in Section 3.3.1.2).

The module `main` also has two instantiations of `prc`. In each of these instantiations, `st` is the status of a process (saying whether it is in its critical section, or not, or trying) and `other-st` is the status of the other process (notice how this is passed as a parameter in the third and fourth lines of `main`).

The value of `st` evolves in the way described in a previous section: when it is n , it may stay as n or move to t . When it is t , if the other one is n , it will go straight to c , but if the other one is t , it will check whose turn it is before going to c . Then, when it is c , it may move back to n . Each instantiation of `prc` gives the turn to the other one when it gets to its critical section.

An important feature of SMV is that we can restrict its search tree to execution paths along which an arbitrary boolean formula about the state ϕ is true infinitely often. Because this is often used to model *fair* access to resources, it is called a *fairness constraint* and introduced by the keyword `FAIRNESS`. Thus, the occurrence of `FAIRNESS ϕ` means that SMV, when checking a specification ψ , will ignore any path along which ϕ is not satisfied infinitely often.

In the module `prc`, we restrict model checks to computation paths along which `st` is infinitely often not equal to c . This is because our code allows the process to stay in its critical section as long as it likes. Thus, there is another opportunity for liveness to fail: if process 2 stays in its critical section forever, process 1 will never be able to enter. Again, we ought not to take this kind of violation into account, since it is patently unfair if a process is allowed to stay in its critical section for ever. We are looking for

```

MODULE main
  VAR
    pr1: process prc(pr2.st, turn, 0);
    pr2: process prc(pr1.st, turn, 1);
    turn: boolean;
  ASSIGN
    init(turn) := 0;
  -- safety
  LTLSPEC G!((pr1.st = c) & (pr2.st = c))
  -- liveness
  LTLSPEC G((pr1.st = t) -> F (pr1.st = c))
  LTLSPEC G((pr2.st = t) -> F (pr2.st = c))
  -- 'negation' of strict sequencing (desired to be false)
  LTLSPEC G(pr1.st=c -> ( G pr1.st=c | (pr1.st=c U
    (!pr1.st=c & G !pr1.st=c | ((!pr1.st=c) U pr2.st=c))))))

MODULE prc(other-st, turn, myturn)
  VAR
    st: {n, t, c};
  ASSIGN
    init(st) := n;
    next(st) :=
      case
        (st = n) : {t,n};
        (st = t) & (other-st = n) : c;
        (st = t) & (other-st = t) & (turn = myturn): c;
        (st = c) : {c,n};
        1 : st;
      esac;
    next(turn) :=
      case
        turn = myturn & st = c : !turn;
        1 : turn;
      esac;
  FAIRNESS running
  FAIRNESS !(st = c)

```

Fig. 3.10. SMV code for mutual exclusion. Because W is not supported by SMV, we had to make use of equivalence (3.3) to write the no-strict-sequencing formula as an equivalent but longer formula involving U .

more subtle violations of the specifications, if there are any. To avoid the one above, we stipulate the fairness constraint $!(st=c)$.

If the module in question has been declared with the `process` keyword, then at each time point SMV will non-deterministically decide whether or not to select it for execution, as explained earlier. We may wish to ignore paths in which a module is starved of processor time. The reserved word `running` can be used instead of a formula in a fairness constraint: writing `FAIRNESS running` restricts attention to execution paths along which the module in which it appears is selected for execution infinitely often.

In `prc`, we restrict ourselves to such paths, since, without this restriction, it would be easy to violate the liveness constraint if an instance of `prc` were *never* selected for execution. We assume the scheduler is fair; this assumption is codified by two `FAIRNESS` clauses. We return to the issue of fairness, and the question of how our model-checking algorithm copes with it, in the next section.

Please run this program in NuSMV to see which specifications hold for it.

The transition system corresponding to this program is shown in Figure 3.11. Each state shows the values of the variables; for example, `ct1` is the state in which process 1 and 2 are critical and trying, respectively, and `turn=1`. The labels on the transitions show which process was selected for execution. In general, each state has several transitions, some in which process 1 moves and others in which process 2 moves.

This model is a bit different from the previous model given for mutual exclusion in Figure 3.8, for these two reasons:

- Because the boolean variable `turn` has been explicitly introduced to distinguish between states s_3 and s_9 of Figure 3.8, we now distinguish between certain states (for example, `ct0` and `ct1`) which were identical before. However, these states are not distinguished if you look just at the transitions *from* them. Therefore, they satisfy the same LTL formulas which don't mention `turn`. Those states are distinguished only by the way they can arise.
- We have eliminated an over-simplification made in the model of Figure 3.8. Recall that we assumed the system would move to a different state on every tick of the clock (there were no transitions from a state to itself). In Figure 3.11, we allow transitions from each state to itself, representing that a process was chosen for execution and did some private computation, but did not move in or out of its critical section. Of course, by doing this we have introduced paths in which one process gets stuck in its critical

section, whence the need to invoke a fairness constraint to eliminate such paths.

3.3.5 *The ferry-man*

You may recall the puzzle of a ferryman, goat, cabbage, and wolf all on one side of a river. The ferryman can cross the river with at most one passenger in his boat. There is a behavioral conflict between

1. the goat and the cabbage; and
2. the goat and the wolf;

if they are on the same river bank but the ferryman crosses the river or stays on the other bank.

Can the ferryman transport all goods to the other side, without any conflicts occurring? This is a *planning problem*, but it can be solved by model checking. We describe a transition system in which the states represent which goods are at which side of the river. Then we ask if the goal state is reachable from the initial state: Is there a path from the initial state such that it has a state along it at which all the goods are on the other side, and during the transitions to that state the goods are never left in an unsafe, conflicting situation?

We model all possible behavior (including that which results in conflicts) as a NuSMV program (Figure 3.12). The location of each agent is modelled as a boolean variable. 0 denotes that the agent is on the initial bank, and 1 the destination bank. Thus, `ferryman = 0` means that the ferryman is on the initial bank, `ferryman = 1` that he is on the destination bank, and similarly for the variables `goat`, `cabbage` and `wolf`.

The variable `carry` takes a value indicating whether the goat, cabbage, wolf or nothing is carried by the ferryman. The definition of `next(carry)` works as follows. It is non-deterministic, but the set from which a value is non-deterministically chosen is determined by the values of `ferryman`, `goat`, etc. If `ferryman = goat` (i.e. they are on the same side) then `g` is a member of the set from which `next(carry)` is chosen. The situation for cabbage and wolf is similar. Thus, if `ferryman = goat = wolf ≠ cabbage` then that set is `{g, w, 0}`. The next value assigned to `ferryman` is non-deterministic: he can choose to cross or not to cross the river. But the next values of `goat`, `cabbage` and `wolf` are deterministic, since whether they are carried or not is determined by the ferryman's choice, represented by the non-deterministic assignment to `carry`; these values follow the same pattern.

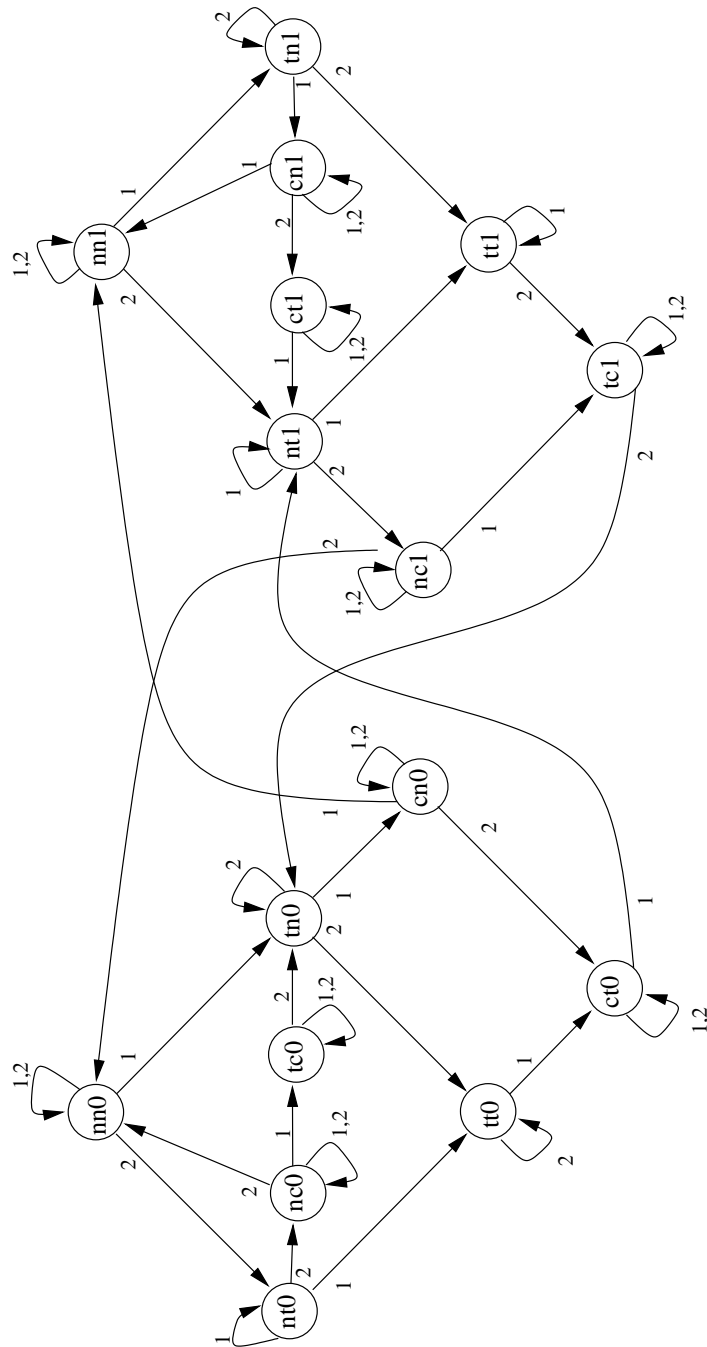


Fig. 3.11. The transition system corresponding to the SMV code in Figure 3.10. The labels on the transitions denote the process which makes the move. The label 1,2 means that either process could make that move.

```

MODULE main
VAR
  ferryman : boolean;
  goat     : boolean;
  cabbage  : boolean;
  wolf     : boolean;
  carry    : {g,c,w,0};
ASSIGN
  init(ferryman) := 0; init(goat)      := 0;
  init(cabbage)  := 0; init(wolf)     := 0;
  init(carry)   := 0;

  next(ferryman) := {0,1};

  next(carry) := case
    ferryman=goat : g;
    1             : 0;
  esac union
  case
    ferryman=cabbage : c;
    1                 : 0;
  esac union
  case
    ferryman=wolf : w;
    1              : 0;
  esac union 0;

  next(goat) := case
    ferryman=goat & next(carry)=g : next(ferryman);
    1                             : goat;
  esac;
  next(cabbage) := case
    ferryman=cabbage & next(carry)=c : next(ferryman);
    1                                 : cabbage;
  esac;
  next(wolf) := case
    ferryman=wolf & next(carry)=w : next(ferryman);
    1                             : wolf;
  esac;

LTLSPEC !(( (goat=cabbage | goat=wolf) -> goat=ferryman)
  U (cabbage & goat & wolf & ferryman))

```

Fig. 3.12. NuSMV code for the ferry-man planning problem.

Note how the boolean guards refer to state bits at the next state. The SMV compiler does a dependency analysis and rejects circular dependencies on next values. (The dependency analysis is rather pessimistic: sometimes NuSMV complains of circularity even in situations when it could be resolved. The original CMU-SMV is more liberal in this respect.)

3.3.5.1 Running NuSMV

We seek a path satisfying $\phi \text{ U } \psi$, where ψ asserts the final goal state, and ϕ expresses the safety condition (if the goat is with the cabbage or the wolf, then the ferryman is there, too, to prevent any untoward behaviour). Thus, we assert that all paths satisfy $\neg(\phi \text{ U } \psi)$, i.e. no path satisfies $\phi \text{ U } \psi$. We hope this is not the case, and NuSMV will give us an example path which does satisfy $\phi \text{ U } \psi$. Indeed, running NuSMV gives us the path of Figure 3.13, which represents a solution to the puzzle.

The beginning of the generated path represents the usual solution to this puzzle: the ferryman takes the goat first, then goes back for the cabbage. To avoid leaving the goat and the cabbage together, he takes the goat back, and picks up the wolf. Now the wolf and the cabbage are on the destination side, and he goes back again to get the goat. This brings us to State 1.9, where the ferryman appears to take a well-earned break. But the path continues. States 1.10 to 1.15 show that he takes his charges back to the original side of the bank; first the cabbage, then the wolf, then the goat. Unfortunately it appears that the ferryman's clever plan up to state 1.9 is now spoiled, because the goat meets an unhappy end in state 1.11.

What went wrong? Nothing, actually. NuSMV has given us an infinite path, which loops around the 15 illustrated states. Along the infinite path, the ferryman repeatedly takes his goods across (safely), and then back again (unsafely). This path does indeed satisfy the specification $\phi \text{ U } \psi$, which asserts the safety of the forward journey but says nothing about what happens after that. In other words, the path is correct; it satisfies $\phi \text{ U } \psi$ (with ψ occurring at state 8). What happens along the path after that has no bearing on $\phi \text{ U } \psi$.

Invoking *bounded model checking* will produce the shortest possible path to violate the property; in this case, it is states 1.1 to 1.8 of the illustrated path. It is the shortest, *optimal* solution to our planning problem since the model check `NuSMV -bmc 7 ferryman.smv` shows that the LTL formula holds in that model, meaning that no solution of length ≤ 7 is possible.

One might wish to verify whether there is a solution which involves three journeys for the goat. This can be done by altering the LTL formula. Instead of seeking a path satisfying $\phi \text{ U } \psi$, where ϕ equals `(goat = cabbage \vee goat =`

```

acws-0116% nusmv ferryman.smv
*** This is NuSMV 2.1.2 (compiled 2002-11-22 12:00:00)
*** For more information of NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv-users@irst.itc.it>.
-- specification !(((goat = cabbage | goat = wolf) -> goat = ferryman)
                    U (((cabbage & goat) & wolf) & ferryman)) is false
-- as demonstrated by the following execution sequence
-- loop starts here --
-> State 1.1 <-
    ferryman = 0
    goat = 0
    cabbage = 0
    wolf = 0
    carry = 0
-> State 1.2 <-
    ferryman = 1
    goat = 1
    carry = g
-> State 1.3 <-
    ferryman = 0
    carry = 0
-> State 1.4 <-
    ferryman = 1
    cabbage = 1
    carry = c
-> State 1.5 <-
    ferryman = 0
    goat = 0
    carry = g
-> State 1.6 <-
    ferryman = 1
    wolf = 1
    carry = w
-> State 1.7 <-
    ferryman = 0
    carry = 0
-> State 1.8 <-
    ferryman = 1
    goat = 1
    carry = g
-> State 1.9 <-
-> State 1.10 <-
    ferryman = 0
    cabbage = 0
    carry = c
-> State 1.11 <-
    ferryman = 1
    carry = 0
-> State 1.12 <-
    ferryman = 0
    wolf = 0
    carry = w
-> State 1.13 <-
    ferryman = 1
    carry = 0
-> State 1.14 <-
    ferryman = 0
    goat = 0
    carry = g
-> State 1.15 <-
    carry = 0

```

Fig. 3.13. A solution path to the ferryman puzzle. It is unnecessarily long. Using bounded model checking will refine it into an optimal solution.

wolf) \rightarrow goat = ferryman and ψ equals $\text{cabbage} \wedge \text{goat} \wedge \text{wolf} \wedge \text{ferryman}$, we now seek a path satisfying $(\phi \cup \psi) \wedge G(\text{goat} \rightarrow G \text{goat})$. The last bit says that once the goat has crossed, he remains across; otherwise, the goat makes at least three trips. NuSMV verifies that the negation of this formula is true, confirming that there is no such solution.

3.3.6 *The alternating bit protocol*

The ABP (alternating bit protocol) is a protocol for transmitting messages along a ‘lossy line,’ i.e. a line which may lose or duplicate messages. The protocol guarantees that, providing the line doesn’t lose infinitely many messages, communication between the sender and the receiver will be successful. (We allow the line to lose or duplicate messages, but it may not corrupt messages; however, there is no way of guaranteeing successful transmission along a line which can corrupt.)

The ABP works as follows. There are four entities, or agents: the sender, the receiver, the message channel and the acknowledgement channel. The sender transmits the first part of the message together with the ‘control’ bit 0. If, and when, the receiver receives a message with the control bit 0, it sends 0 along the acknowledgement channel. When the sender receives this acknowledgement, it sends the next packet with the control bit 1. If and when the receiver receives this, it acknowledges by sending a 1 on the acknowledgement channel. By alternating the control bit, both receiver and sender can guard against duplicating messages and losing messages (i.e. they ignore messages that have the unexpected control bit).

If the sender doesn’t get the expected acknowledgement, it continually resends the message, until the acknowledgement arrives. If the receiver doesn’t get a message with the expected control bit, it continually resends the previous acknowledgement.

Fairness is also important for the ABP. It comes in because, although we want to model the fact that the channel can lose messages, we want to assume that, if we send a message often enough, eventually it will arrive. In other words, the channel cannot lose an infinite sequence of messages. If we did not make this assumption, then the channels could lose all messages and, in that case, the ABP would not work.

Let us see this in the concrete setting of SMV. We may assume that the text to be sent is divided up into single-bit messages, which are sent sequentially. The variable `message1` is the current bit of the message being sent, whereas `message2` is the control bit. The definition of the module `sender` is given in Figure 3.14. This module spends most of its time in `st=sending`, going only briefly to `st=sent` when it receives an acknowledgement corresponding to the control bit of the message it has been sending. The variables `message1` and `message2` represent the actual data being sent and the control bit, respectively. On successful transmission, the module obtains a new message to send and returns to `st=sending`. The new `message1` is obtained non-deterministically (i.e. from the environment); `message2` alternates in

```

MODULE sender(ack)
VAR
  st      : {sending,sent};
  message1 : boolean;
  message2 : boolean;
ASSIGN
  init(st) := sending;
  next(st) := case
    ack = message2 & !(st=sent) : sent;
    1                             : sending;
  esac;
  next(message1) :=
    case
      st = sent : {0,1};
      1         : message1;
    esac;
  next(message2) :=
    case
      st = sent : !message2;
      1         : message2;
    esac;
FAIRNESS running
LTLSPEC G F st=sent

```

Fig. 3.14. The ABP sender in SMV.

value. We impose **FAIRNESS running**, i.e. the sender must be selected to run infinitely often. The **LTLSPEC** tests that we can always succeed in sending the current message. The module **receiver** is programmed in a similar way, in Figure 3.15.

We also need to describe the two channels, in Figure 3.16. The acknowledgement channel is an instance of the one-bit channel **one-bit-chan** below. Its lossy character is specified by the non-deterministic assignment: the **input** may be transmitted to the **output**, but it need not (in which case output retains its old value). However, the second fairness constraint ensures that the channel doesn't continually lose the same message: eventually, a bit will get through (so if **input** is 1, then eventually **output** will be 1 too).

The two-bit channel **two-bit-chan**, used to send messages, is similar. The non-deterministic variable **forget** determines whether the current bit is lost or not. Either both parts of the message get through, or neither of them does (the channel is assumed not to corrupt messages).

A fairness constraint models the fact that, although channels can lose messages, even infinitely often, we assume that they infinitely often transmit

```

MODULE receiver(message1,message2)
VAR
  st      : {receiving,received};
  ack     : boolean;
  expected : boolean;
ASSIGN
  init(st) := receiving;
  next(st) := case
    message2=expected & !(st=received) : received;
    1                                     : receiving;
  esac;
  next(ack) :=
    case
      st = received : message2;
      1             : ack;
    esac;
  next(expected) :=
    case
      st = received : !expected;
      1             : expected;
    esac;
FAIRNESS running
LTLSPEC G F st=received

```

Fig. 3.15. The ABP receiver in SMV.

the message correctly. (If this were not the case, then we could find an uninteresting violation of the liveness constraint, for example a path along which all messages from a certain time onwards get lost.)

Finally, we tie it all together with the module `main` (Figure 3.17). Its role is to connect together the components of the system, which it does by instantiating the four processes. It also specifies the initial values. Since the first control bit is 0, we also initialise the receiver to expect a 0. The receiver should start off by sending 1 as its acknowledgement, so that `sender` does not think that its very first message is being acknowledged before anything has happened. For the same reason, the output of the channels is initialised to 1.

The specifications for ABP. Our SMV program satisfies the following specifications:

Safety: If the message bit 1 has been sent and the correct acknowledgement has been returned, then a 1 was received by the receiver: $G(S.st=sent \ \& \ S.message1=1 \ \rightarrow \ msg_chan.output1=1)$.

```

MODULE one-bit-chan(input)
VAR
  output : boolean;
ASSIGN
  next(output) := {input,output};
FAIRNESS running
FAIRNESS (input=0 -> AF output=0) & (input=1 -> AF output=1)

MODULE two-bit-chan(input1,input2)
VAR
  forget : boolean;
  output1 : boolean;
  output2 : boolean;
ASSIGN
  next(output1) := case
    forget : output1;
    1:      input1;
  esac;
  next(output2) := case
    forget : output2;
    1:      input2;
  esac;
FAIRNESS running
FAIRNESS !forget

```

Fig. 3.16. The two modules for the two ABP channels in SMV.

```

MODULE main
VAR
  S : process sender(ack_chan.output);
  R : process receiver(msg_chan.output1,msg_chan.output2);
  msg_chan : process two-bit-chan(S.message1,S.message2);
  ack_chan : process one-bit-chan(R.ack);
ASSIGN
  init(S.message2) := 0;
  init(R.expected) := 0;
  init(R.ack)      := 1;
  init(msg_chan.output2) := 1;
  init(ack_chan.output) := 1;

LTLSPEC G (S.st=sent & S.message1=1 -> msg_chan.output1=1)

```

Fig. 3.17. The main ABP module.

Liveness: Messages get through eventually. Thus, for any state there is inevitably a future state in which the current message has got through. In the module `sender`, we specified $\mathbf{G F st=sent}$. (This specification could equivalently have been written in the main module, as $\mathbf{G F S.st=sent}$.)

Similarly, acknowledgements get through eventually. In the module `receiver`, we write $\mathbf{G F st=received}$.

3.4 Branching-time logic

In our analysis of LTL (*linear-time temporal logic*) in the preceding sections, we noted that LTL formulas are evaluated on *paths*. We defined that a *state* of a system satisfies an LTL formula if *all paths* from the given state satisfy it. Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL. This problem can partly be alleviated by considering the negation of the property in question, and interpreting the result accordingly. To check whether there exists a path from s satisfying the LTL formula ϕ , we check whether all paths satisfy $\neg\phi$; a positive answer to this is a negative answer to our original question, and vice versa. We used this approach when analysing the ferryman puzzle in the previous section. However, as already noted, properties which *mix* universal and existential path quantifiers cannot in general be model checked using this approach, because the complement formula still has a mix.

Branching-time logics solve this problem by allowing us to quantify explicitly over paths. We will examine a logic known as *Computation Tree Logic*, or CTL. In CTL, as well as the temporal operators U, F, G and X of LTL we also have quantifiers A and E which express ‘all paths’ and ‘exists a path’ respectively. For example, we can write

- There is a reachable state satisfying q : this is written $\mathbf{EF} q$
- From all reachable states satisfying p , it is possible to maintain p continuously until reaching a state satisfying q : this is written $\mathbf{AG} (p \rightarrow \mathbf{E}[p \mathbf{U} q])$.
- Whenever a state satisfying p is reached, the system can exhibit q continuously forevermore: $\mathbf{AG} (p \rightarrow \mathbf{EG} q)$.
- There is a reachable state from which all reachable states satisfy p : $\mathbf{EF} \mathbf{AG} p$.

3.4.1 Syntax of CTL

Computation tree logic, or CTL for short, is 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.

As before, we work with a fixed set of atomic formulas/descriptions (such as p, q, r, \dots , or p_1, p_2, \dots).

Definition 3.12 We define CTL formulas inductively via a Backus Naur form as done for LTL:

$$\begin{aligned} \phi ::= & \perp \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \text{AX } \phi \mid \text{EX } \phi \mid \\ & \text{AF } \phi \mid \text{EF } \phi \mid \text{AG } \phi \mid \text{EG } \phi \mid \text{A}[\phi \text{ U } \phi] \mid \text{E}[\phi \text{ U } \phi] \end{aligned}$$

where p ranges over a set of atomic formulas.

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 symbols in $\text{E}[\phi_1 \text{ U } \phi_2]$, for example, is EU. In CTL, pairs of symbols like EU are indivisible. 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.

Usually weak-until (W) and release (R) are not included in CTL, but they are derivable (see Section 3.4.5).

Convention 3.13 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 can 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:

- $\text{AG}(q \rightarrow \text{EG } r)$, note that this is not the same as $\text{AG } q \rightarrow \text{EG } r$, for according to Convention 3.13, the latter formula means $(\text{AG } q) \rightarrow (\text{EG } r)$
- $\text{EF } \text{E}[r \text{ U } q]$
- $\text{A}[p \text{ U } \text{EF } r]$

- $EF EG p \rightarrow AF r$, again, note that this binds as $(EF EG p) \rightarrow AF r$, not $EF (EG p \rightarrow AF r)$ or $EF EG (p \rightarrow AF r)$
- $A[p_1 U A[p_2 U p_3]]$
- $E[A[p_1 U p_2] U p_3]$
- $AG (p \rightarrow A[p U (\neg p \wedge A[\neg p U q])])$.

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:

- $EF G r$
- $A \neg G \neg p$
- $F [r U q]$
- $EF (r U q)$
- $A EF r$
- $A[(r U q) \wedge (p U r)]$.

It is especially worth understanding why the syntax rules don't allow us to construct these. For example, take $EF (r U q)$. The problem with this string 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 $EF E[r U q]$ or $EF A[r U q]$.

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 SMV insists on it.

The reason $A[(r U q) \wedge (p U r)]$ 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 $A[\phi U \psi]$. Now, the ϕ and the ψ may contain \wedge , since they are arbitrary formulas; so $A[(p \wedge q) U (\neg r \rightarrow q)]$ 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 $\phi_1 AU \phi_2$, whereas in pure prefix we would write $AU(\phi_1, \phi_2)$.

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 $A[AX \neg p U E[EX (p \wedge q) U \neg p]]$ is shown in Figure 3.18.

Definition 3.14 A subformula of a CTL formula ϕ is any formula ψ whose parse tree is a subtree of ϕ 's parse tree.

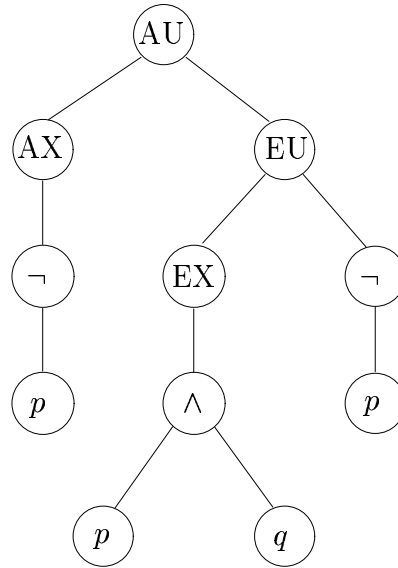


Fig. 3.18. The parse tree of a CTL formula without infix notation.

3.4.2 Semantics of computation tree logic

CTL formulas are interpreted over transition systems (Definition 3.4). Let $\mathcal{M} = (S, \rightarrow, L)$ be such a model, $s \in S$ and ϕ a CTL formula. The definition of whether $\mathcal{M}, s \models \phi$ holds is recursive on the structure of ϕ , and can be roughly understood as follows:

- If ϕ is atomic, satisfaction is determined by L .
- If the top-level connective of ϕ (i.e. the connective occurring top-most in the parse tree of ϕ) is a boolean connective (\wedge, \vee, \neg, \top etc.) then the satisfaction question is answered by the usual truth-table definition and further recursion down ϕ .
- If the top level connective is an operator beginning A, then satisfaction holds if all paths from s satisfy the ‘LTL formula’ resulting from removing the A symbol.
- Similarly, if the top level connective begins with E, then satisfaction holds if some path from s satisfy the ‘LTL formula’ resulting from removing the E.

In the last two cases, the result of removing A or E is not strictly an LTL formula, for it may contain further As or Es below. However, these will be dealt with by the recursion.

The formal definition of $\mathcal{M}, s \models \phi$ is a bit more verbose:

Definition 3.15 Let $\mathcal{M} = (S, \rightarrow, L)$ be a model for CTL, s in S , ϕ a CTL formula. The relation $\mathcal{M}, s \models \phi$ is defined by structural induction on ϕ :

1. $\mathcal{M}, s \models \top$ and $\mathcal{M}, s \not\models \perp$.
2. $\mathcal{M}, s \models p$ iff $p \in L(s)$.
3. $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$.
4. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$.
5. $\mathcal{M}, s \models \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \models \phi_1$ or $\mathcal{M}, s \models \phi_2$.
6. $\mathcal{M}, s \models \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \not\models \phi_1$ or $\mathcal{M}, s \models \phi_2$.
7. $\mathcal{M}, s \models AX \phi$ iff for all s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \models \phi$. Thus, AX says: ‘in every next state.’
8. $\mathcal{M}, s \models EX \phi$ iff for some s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \models \phi$. 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. $\mathcal{M}, s \models AG \phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, 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. $\mathcal{M}, s \models EG \phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, 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. $\mathcal{M}, s \models AF \phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow \dots$, 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. $\mathcal{M}, s \models EF \phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, 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. $\mathcal{M}, s \models A[\phi_1 U \phi_2]$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , that path satisfies $\phi_1 U \phi_2$, 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. $\mathcal{M}, s \models E[\phi_1 U \phi_2]$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , and that path satisfies $\phi_1 U \phi_2$ as specified in 13. Mnemonically: there Exists a computation path beginning in s such that ϕ_1 Until ϕ_2 holds on it.

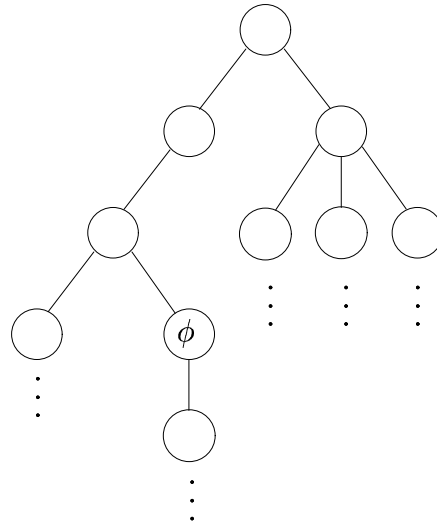
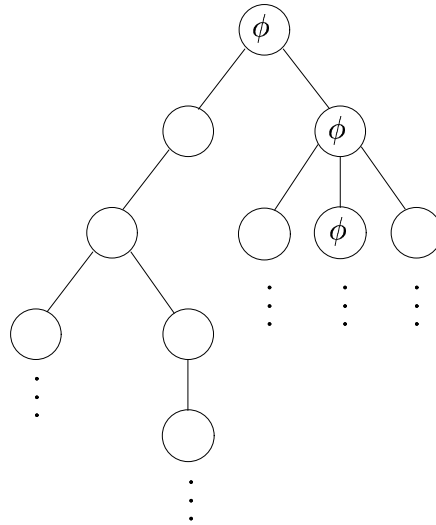
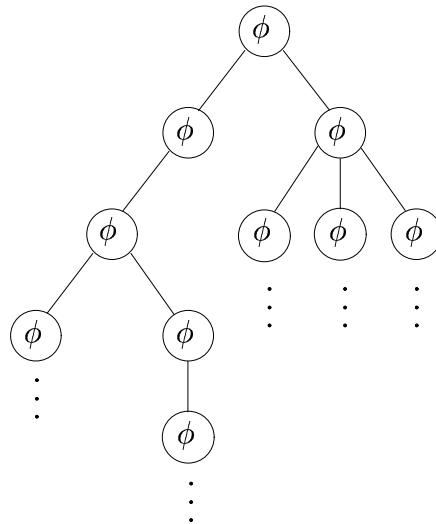


Fig. 3.19. A system whose starting state satisfies $EF \phi$.

Clauses 8–14 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.’ The diagrams in Figures 3.19–3.22 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.

Recall the transition system of Figure 3.3 (page 187) for the designated starting state s_0 , and the infinite tree illustrated in Figure 3.5. Let us now look at some example checks for this system.

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. $\mathcal{M}, s_0 \models EX (q \wedge r)$ holds since we have the leftmost computation path $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$ in Figure 3.5, whose second node s_1 contains q and r .
5. $\mathcal{M}, s_0 \models \neg AX (q \wedge r)$ holds since we have the rightmost computation path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ in Figure 3.5, whose second node s_2 only contains r , but *not* q .

Fig. 3.20. A system whose starting state satisfies EG ϕ .Fig. 3.21. A system whose starting state satisfies AG ϕ .

6. $\mathcal{M}, s_0 \models \neg \text{EF}(p \wedge r)$ 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. $\mathcal{M}, s_2 \models \text{EG} r$ holds since there is a computation path $s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ beginning in s_2 such that r holds in all future states of

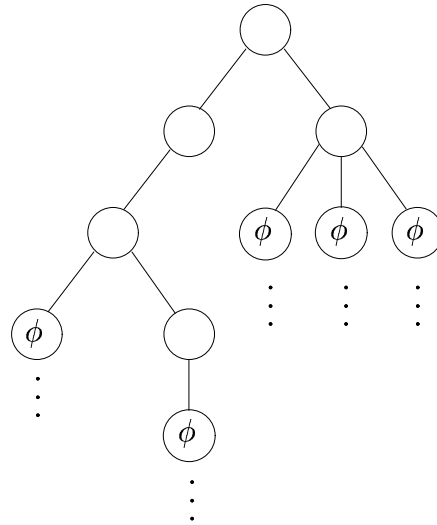


Fig. 3.22. A system whose starting state satisfies $\text{AF } \phi$.

that path; this is the only computation path beginning at s_2 and so $\mathcal{M}, s_2 \models \text{AG } r$ holds as well.

8. $\mathcal{M}, s_0 \models \text{AF } r$ holds since, for all computation paths beginning in s_0 , the system reaches a state (s_1 or s_2) such that r holds.
9. $\mathcal{M}, s_0 \models \text{E}[(p \wedge q) \text{ U } r]$ holds since we have the rightmost computation path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ in Figure 3.5, whose second node s_2 ($i = 1$) satisfies r , but all previous nodes (only $j = 0$, i.e. node s_0) satisfy $p \wedge q$.
10. $\mathcal{M}, s_0 \models \text{A}[p \text{ U } r]$ holds since p holds at s_0 and r holds in any possible successor state of s_0 , so $p \text{ U } r$ is true for all computation paths beginning in s_0 (so we may choose $i = 1$ independently of the path).
11. $\mathcal{M}, s_0 \models \text{AG } (p \vee q \vee r \rightarrow \text{EF EG } r)$ holds since in all states reachable from s_0 and satisfying $p \vee q \vee r$ (all states in this case) the system can reach a state satisfying $\text{EG } r$ (in this case state s_2).

3.4.3 Practical patterns of specifications

It's useful to look at some typical examples of formulas, and compare the situation with LTL (Section 3.2.3). Suppose atomic descriptions include some words such as busy and requested.

- It is possible to get to a state where started holds, but ready doesn't: $\text{EF}(\text{started} \wedge \neg \text{ready})$. To express impossibility, we simply negate the formula.

- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
AG (requested \rightarrow AF acknowledged).
 - The property that if the process is enabled infinitely often, then it runs infinitely often, is not expressible in CTL. In particular, it is not expressed by AG AF enabled \rightarrow AG AF running, or indeed any other insertion of A or E into the corresponding LTL formula. The CTL formula just given expresses that if every path has infinitely often enabled, then every path is infinitely often taken; this is much weaker than asserting that every path which has infinitely often enabled is infinitely often taken.
 - 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 \wedge direction=up \wedge ButtonPressed5 \rightarrow
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 \wedge idle \wedge door=closed \rightarrow EG (floor=3 \wedge idle \wedge door=closed)).
 - A process can always request to enter its critical section. Recall that this was not expressible in LTL. Using the propositions of Figure 3.8, this may be written AG ($n_1 \rightarrow$ EX t_1) in CTL.
 - Processes need not enter their critical section in strict sequence. This was also not expressible in LTL, though we expressed its negation. CTL allows us to express it directly: EF ($c_1 \wedge$ E[c_1 U ($\neg c_1 \wedge$ E[$\neg c_2$ U c_1]))).

3.4.4 Important equivalences between CTL formulas

Definition 3.16 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 $\phi \equiv \psi$.

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 uni-

versal 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:

$$\begin{aligned}\neg\text{AF } \phi &\equiv \text{EG } \neg\phi & (3.6) \\ \neg\text{EF } \phi &\equiv \text{AG } \neg\phi \\ \neg\text{AX } \phi &\equiv \text{EX } \neg\phi .\end{aligned}$$

We also have the equivalences

$$\text{AF } \phi \equiv \text{A}[\top \text{ U } \phi] \qquad \text{EF } \phi \equiv \text{E}[\top \text{ U } \phi]$$

which are similar to the corresponding equivalences in LTL.

3.4.5 Adequate sets of CTL connectives

As in propositional logic and in LTL, there is some redundancy among the CTL connectives. For example, the connective AX can be written $\neg\text{EX } \neg$; and AG, AF, EG and EF can be written in terms of AU and EU as follows: first, write $\text{AG } \phi$ as $\neg\text{EF } \neg\phi$ and $\text{EG } \phi$ as $\neg\text{AF } \neg\phi$, using (3.6), and then use $\text{AF } \phi \equiv \text{A}[\top \text{ U } \phi]$ and $\text{EF } \phi \equiv \text{E}[\top \text{ U } \phi]$. 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

$$\text{A}[\phi \text{ U } \psi] \equiv \neg(\text{E}[\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)] \vee \text{EG } \neg\psi) \quad (3.7)$$

which can be proved as follows:

$$\begin{aligned}\text{A}[\phi \text{ U } \psi] &\equiv \text{A}[\neg(\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)) \wedge \text{F } \psi] \\ &\equiv \neg\text{E}\neg[\neg(\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)) \wedge \text{F } \psi] \\ &\equiv \neg\text{E}[(\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)) \vee \text{G } \neg\psi] \\ &\equiv \neg(\text{E}[\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)] \vee \text{EG } \neg\psi) .\end{aligned}$$

The first line is by Theorem 3.10, and the remainder by elementary manipulation. (This proof involves intermediate formulas which violate the syntactic formation rules of CTL; however, it is valid in the logic CTL* introduced in the next section.) More generally, we have:

Theorem 3.17 A set of temporal connectives in CTL is adequate if, and only if, it contains at least one of {AX, EX}, at least one of {EG, AF, AU} and EU.

This theorem is proved in a paper referenced in the bibliographic notes at the end of the chapter. The connective EU plays a special role in that theorem because neither weak-until W nor release R are primitive in CTL (Definition 3.12). The temporal connectives AR, ER, AW and EW are all definable in CTL:

- $A[\phi R \psi] = \neg E[\neg\phi U \neg\psi]$
- $E[\phi R \psi] = \neg A[\neg\phi U \neg\psi]$
- $A[\phi W \psi] = A[\psi R (\phi \vee \psi)]$, and then use the first equation above
- $E[\phi W \psi] = E[\psi R (\phi \vee \psi)]$, and then use the second one.

These definitions are justified by LTL equivalences in Sections 3.2.4 and 3.2.5. Some other noteworthy equivalences in CTL are the following:

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

For example, the intuition for the third one is the following: in order to have $AF \phi$ 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 $AF \phi$ 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 3.6.1; and we return to it later (Section 3.7).

3.5 CTL* and the expressive powers of LTL and CTL

CTL allows explicit quantification over paths, and in this respect it is more expressive than LTL, as we have seen. However, it does not allow one to select a range of paths by describing them with a formula, as LTL does. In that respect, LTL is more expressive. For example, in LTL we can say ‘all paths which have a p along them also have a q along them,’ by writing $Fp \rightarrow Fq$. It is not possible to write this in CTL because of the constraint that every F has an associated A or E. The formula $AF p \rightarrow AF q$ means something quite different: it says ‘if all paths have a p along them, then

all paths have a q along them.’ One might write $AG(p \rightarrow AF q)$, which is closer, since it says that every way of extending every path to a p eventually meets a q , but that is still not capturing the meaning of $Fp \rightarrow Fq$.

CTL* is a logic which combines the expressive powers of LTL and CTL, by dropping the CTL constraint that every temporal operator (X, U, F, G) has to be associated with a unique path quantifier (A, E). It allows us to write formulas such as

- $A[(p \text{ U } r) \vee (q \text{ U } r)]$: along all paths, either p is true until r , or q is true until r .
- $A[Xp \vee XXp]$: along all paths, p is true in the next state, or the next but one.
- $E[GFp]$: there is a path along which p is infinitely often true.

These formulas are *not* equivalent to, respectively, $A[(p \vee q) \text{ U } r]$, $AXp \vee AXAXp$ and $EGEFp$. It turns out that the first of them can be written as a (rather long) CTL formula. The second and third do not have a CTL equivalent.

The syntax of CTL* involves two classes of formulas:

- *state formulas*, which are evaluated in states:

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \psi) \mid A[\alpha] \mid E[\alpha]$$

where p is any atomic formula and α any path formula; and

- *path formulas*, which are evaluated along paths:

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \beta) \mid (\alpha \text{ U } \beta) \mid (G\alpha) \mid (F\alpha) \mid (X\alpha)$$

where ϕ is any state formula. This is an example of an inductive definition which is *mutually recursive*: the definition of each class depends upon the definition of the other, with base cases p and \top .

3.5.0.1 LTL and CTL as subsets of CTL*

Although the syntax of LTL does not include A and E, the semantic viewpoint of LTL is that we consider all paths. Therefore, the LTL formula α is equivalent to the CTL* formula $A[\alpha]$. Thus, LTL can be viewed as a subset of CTL*.

CTL is also a subset of CTL*, since it is the fragment of CTL* in which we restrict the form of path formulas to

$$\alpha ::= (\phi \text{ U } \psi) \mid (G\phi) \mid (F\phi) \mid (X\phi)$$

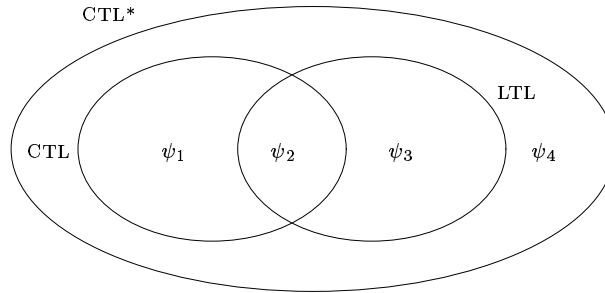
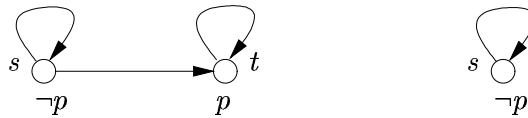


Fig. 3.23. The expressive powers of CTL, LTL and CTL*.

Figure 3.23 shows the relationship among the expressive powers of CTL, LTL and CTL*. Here are some examples of formulas in each of the subsets shown:

In CTL but not in LTL: $\psi_1 \stackrel{\text{def}}{=} \text{AG EF } p$. This expresses: wherever we have got to, we can always get to a state in which p is true. This is also useful, e.g. in finding deadlocks in protocols.

The proof that $\text{AG EF } p$ is not expressible in LTL is as follows. Let ϕ be an LTL formula such that $\text{A}[\phi]$ is allegedly equivalent to $\text{AG EF } p$. Since $\mathcal{M}, s \models \text{AG EF } p$ in the left-hand diagram below, we have $\mathcal{M}, s \models \text{A}[\phi]$. Now let \mathcal{M}' be as shown in the right-hand diagram. The paths from s in \mathcal{M}' are a subset of those from s in \mathcal{M} , so we have $\mathcal{M}', s \models \text{A}[\phi]$. Yet, it is *not* the case that $\mathcal{M}', s \models \text{AG EF } p$; a contradiction.



In CTL*, but neither in CTL nor in LTL: $\psi_4 \stackrel{\text{def}}{=} \text{E}[\text{G F } p]$, saying that there is a path with infinitely many p .

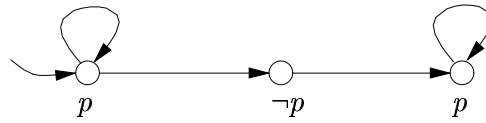
The proof that this is not expressible in CTL is quite complex and may be found in the papers co-authored by E. A. Emerson with others, given in the references. (Why is it not expressible in LTL?)

In LTL but not in CTL: $\psi_3 \stackrel{\text{def}}{=} \text{A}[\text{G F } p \rightarrow \text{F } q]$, saying that if there are infinitely many p along the path, then there is an occurrence of q . This is an interesting thing to be able to say; for example, many fairness constraints are of the form ‘infinitely often requested implies eventually acknowledged.’

In LTL and CTL: $\psi_2 \stackrel{\text{def}}{=} \text{AG}(p \rightarrow \text{AF} q)$ in CTL, or $\text{G}(p \rightarrow \text{F} q)$ in LTL: any p is eventually followed by a q .

Remark 3.18 We just saw that some (but not all) LTL formulas can be converted into CTL formulas by adding an A to each temporal operator. For a positive example, the LTL formula $\text{G}(p \rightarrow \text{F} q)$ is equivalent to the CTL formula $\text{AG}(p \rightarrow \text{AF} q)$. We discuss two more negative examples:

- $\text{F} \text{G} p$ and $\text{AF} \text{AG} p$ are not equivalent, since $\text{F} \text{G} p$ is satisfied, whereas $\text{AF} \text{AG} p$ is not satisfied, in the model



In fact, $\text{AF} \text{AG} p$ is strictly stronger than $\text{F} \text{G} p$.

- While the LTL formulas $\text{XF} p$ and $\text{FX} p$ are equivalent, and they are equivalent to the CTL formula $\text{AX} \text{AF} p$, they are not equivalent to $\text{AF} \text{AX} p$. The latter is strictly stronger, and has quite a strange meaning (try working it out).

Remark 3.19 There is a considerable literature comparing linear-time and branching-time logics. The question of which one is ‘better’ has been debated for about 20 years. We have seen that they have incomparable expressive powers. CTL* is more expressive than either of them, but is computationally much more expensive (as will be seen in Section 3.6). The choice between LTL and CTL depends on the application at hand, and on personal preference. LTL lacks CTL’s ability to quantify over paths, and CTL lacks LTL’s finer-grained ability to describe individual paths. To many people, LTL appears to be more straightforward to use; as noted above, CTL formulas like $\text{AF} \text{AX} p$ seem hard to understand.

3.5.1 Boolean combinations of temporal formulas in CTL

Compared with CTL*, the syntax of CTL is restricted in two ways: it does not allow boolean combinations of path formulas and it does not allow nesting of the path modalities X, F and G. Indeed, we have already seen examples of the inexpressibility in CTL of nesting of path modalities, namely the formulas ψ_3 and ψ_4 above.

In this section, we see that the first of these restrictions is only apparent; we can find equivalents in CTL for formulas having boolean combinations

of path formulas. The idea is to translate any CTL formula having boolean combinations of path formulas into a CTL formula that doesn't. For example, we may see that $E[F p \wedge F q] \equiv EF[p \wedge EF q] \vee EF[q \wedge EF p]$ since, if we have $F p \wedge F q$ along any path, then either the p must come before the q , or the other way around, corresponding to the two disjuncts on the right. (If the p and q occur simultaneously, then both disjuncts are true.)

Since U is like F (only with the extra complication of its first argument), we find the following equivalence:

$$E[(p_1 U q_1) \wedge (p_2 U q_2)] \equiv E[(p_1 \wedge p_2) U (q_1 \wedge E[p_2 U q_2])] \\ \vee E[(p_1 \wedge p_2) U (q_2 \wedge E[p_1 U q_1])] .$$

And from the CTL equivalence $A[p U q] \equiv \neg(E[\neg q U (\neg p \wedge \neg q)] \vee EG \neg q)$ (see Theorem 3.10) we can obtain $E[\neg(p U q)] \equiv E[\neg q U (\neg p \wedge \neg q)] \vee EG \neg q$. Other identities we need in this translation include $E[\neg X p] \equiv EX \neg p$.

3.5.2 Past operators in LTL

The temporal operators X , U , F , etc. which we have seen so far refer to the future. Sometimes we want to encode properties that refer to the past, such as: 'whenever q occurs, then there was some p in the past.' To do this, we may add the operators Y , S , O , H . They stand for *yesterday*, *since*, *once*, and *historically*, and are the past analogues of X , U , F , G , respectively. Thus, the example formula may be written $G(q \rightarrow Op)$.

NuSMV supports past operators in LTL. One could also add past operators to CTL (AY, ES etc.) but NuSMV does not support them.

Somewhat counter-intuitively, past operators do not increase the expressive power of LTL. That is to say, every LTL formula with past operators can be written equivalently without them. The example formula above can be written $\neg p W q$, or equivalently $\neg(\neg q U (p \wedge \neg q))$ if one wants to avoid W . This result is surprising, because it seems that being able to talk about the past as well as the future allows more expressivity than talking about the future alone. However, recall that LTL equivalence is quite crude: it says that the two formulas are satisfied by exactly the same set of paths. The past operators allow us to travel backwards along the path, but only to reach points we could have reached by travelling forwards from its beginning. In contrast, adding past operators to CTL does increase its expressive power, because they can allow us to examine states not forward-reachable from the present one.

3.6 Model checking algorithms

The semantic definitions for LTL and CTL presented in Sections 3.2 and 3.4 allow us to test whether the initial states of a given system satisfy an LTL or CTL formula. This is the basic model-checking question. In general, interesting transition systems will have a huge number of states and the formula we are interested in checking may be quite long. It is therefore well worth trying to find efficient algorithms.

Although LTL is generally preferred by specifiers, as already noted, we start with CTL model checking because its algorithm is simpler.

3.6.1 The CTL model-checking algorithm

Humans may find it easier to do model checks on the unwindings of models into infinite trees, given a designated initial state, for then 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 infinite trees. We need to do checks on *finite* data structures. For this reason, 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 case that ϕ is not satisfied, such an algorithm can be augmented to produce an actual path (= run) of the system demonstrating that \mathcal{M} cannot satisfy ϕ . That way, we may *debug* a system by trying to fix what enables runs which refute ϕ .

There are various ways in which one could consider

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

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

It turns out that it is easier to provide an algorithm for solving the second of these two problems. This automatically gives us a solution to the first one, since we can simply check whether s_0 is an element of the output set.

3.6.1.1 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 \perp , \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 in order to write it in an equivalent form in terms of the adequate set of connectives, and then call the model-checking algorithm. Here is the algorithm:

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula ϕ .

OUTPUT: the set of states of \mathcal{M} which satisfy ϕ .

First, change ϕ to the output of $\text{TRANSLATE}(\phi)$, i.e. we write ϕ in terms of the connectives AF, EU, EX, \wedge , \neg and \perp using the equivalences given earlier in the chapter. Next, label the states of \mathcal{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. We determine by a case analysis which states to label with ψ . If ψ is

- \perp : then no states are labelled with \perp .
- 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 .
- AF ψ_1 :
 - If any state s is labelled with ψ_1 , label it with AF ψ_1 .
 - Repeat: label any state with AF ψ_1 if all successor states are labelled with AF ψ_1 , until there is no change. This step is illustrated in Figure 3.24.
- E[ψ_1 U ψ_2]:
 - If any state s is labelled with ψ_2 , label it with E[ψ_1 U ψ_2].
 - Repeat: label any state with E[ψ_1 U ψ_2] if it is labelled with ψ_1 and at least one of its successors is labelled with E[ψ_1 U ψ_2], until there is no change. This step is illustrated in Figure 3.25.
- EX ψ_1 : label any state with EX ψ_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

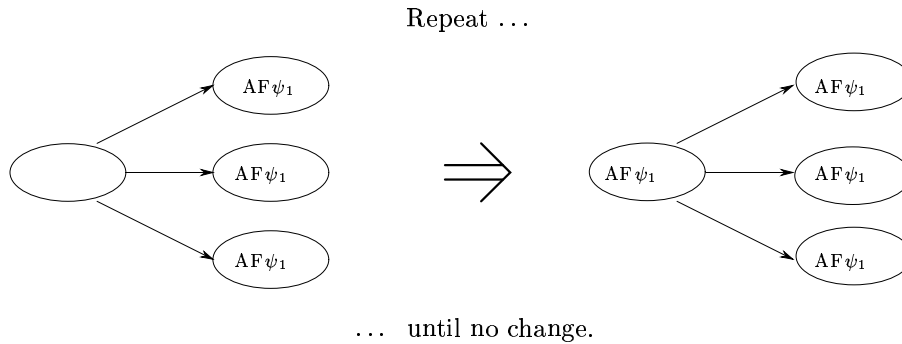


Fig. 3.24. The iteration step of the procedure for labelling states with subformulas of the form $AF \psi_1$.

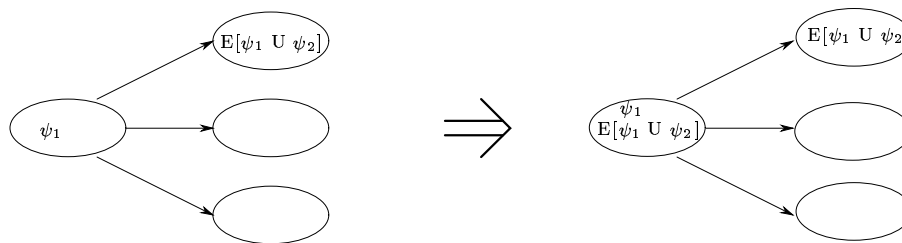


Fig. 3.25. The iteration step of the procedure for labelling states with subformulas of the form $E[\psi_1 U \psi_2]$.

the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

3.6.1.2 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

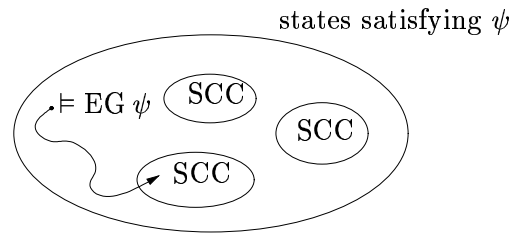


Fig. 3.26. A better way of handling EG.

the case for EU. Actually, there is no real difference between this procedure for $EG \psi$ and what you would do if you translated it into $\neg AF \neg \psi$ as far as the final result is concerned.

3.6.1.3 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 search, for this ensures that we won't have to pass over any node twice). For the $EG \psi$ case:

- 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 search on the restricted graph to find any state that can reach an SCC; see Figure 3.26.

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 3.20 We applied the basic algorithm to our second model of mutual exclusion with the formula $E[\neg c_2 \cup c_1]$; see Figure 3.27. The algorithm labels all states which satisfy c_1 during phase 1 with $E[\neg c_2 \cup c_1]$. 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.

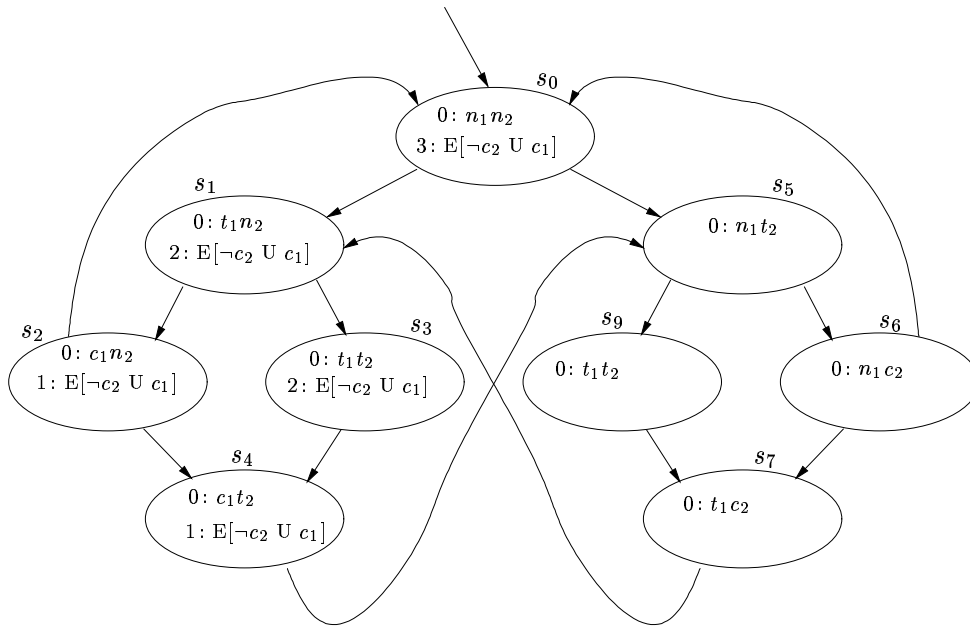


Fig. 3.27. An example run of the labelling algorithm in our second model of mutual exclusion applied to the formula $E[\neg c_2 \cup c_1]$.

3.6.1.4 The pseudo-code of the CTL model checking algorithm

We present the pseudo-code for the basic labelling algorithm. The main function **SAT** (for ‘satisfies’) takes as input a CTL formula. The program **SAT** expects a parse tree of some CTL formula constructed by means of the grammar in Definition 3.12. 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.

The pseudo-code we write for **SAT** looks a bit like fragments of C or Java 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, together with implementations of these operations, but for now we


```

function SAT ( $\phi$ )
  /* determines the set of states satisfying  $\phi$  */
begin
  case
     $\phi$  is  $\top$  : return  $S$ 
     $\phi$  is  $\perp$  : return  $\emptyset$ 
     $\phi$  is atomic: return  $\{s \in S \mid \phi \in L(s)\}$ 
     $\phi$  is  $\neg\phi_1$  : return  $S - \text{SAT}(\phi_1)$ 
     $\phi$  is  $\phi_1 \wedge \phi_2$  : return  $\text{SAT}(\phi_1) \cap \text{SAT}(\phi_2)$ 
     $\phi$  is  $\phi_1 \vee \phi_2$  : return  $\text{SAT}(\phi_1) \cup \text{SAT}(\phi_2)$ 
     $\phi$  is  $\phi_1 \rightarrow \phi_2$  : return  $\text{SAT}(\neg\phi_1 \vee \phi_2)$ 
     $\phi$  is  $\text{AX } \phi_1$  : return  $\text{SAT}(\neg\text{EX } \neg\phi_1)$ 
     $\phi$  is  $\text{EX } \phi_1$  : return  $\text{SAT}_{\text{EX}}(\phi_1)$ 
     $\phi$  is  $\text{A}[\phi_1 \text{ U } \phi_2]$  : return  $\text{SAT}(\neg(\text{E}[\neg\phi_2 \text{ U } (\neg\phi_1 \wedge \neg\phi_2)] \vee \text{EG } \neg\phi_2))$ 
     $\phi$  is  $\text{E}[\phi_1 \text{ U } \phi_2]$  : return  $\text{SAT}_{\text{EU}}(\phi_1, \phi_2)$ 
     $\phi$  is  $\text{EF } \phi_1$  : return  $\text{SAT}(\text{E}(\top \text{ U } \phi_1))$ 
     $\phi$  is  $\text{EG } \phi_1$  : return  $\text{SAT}(\neg\text{AF } \neg\phi_1)$ 
     $\phi$  is  $\text{AF } \phi_1$  : return  $\text{SAT}_{\text{AF}}(\phi_1)$ 
     $\phi$  is  $\text{AG } \phi_1$  : return  $\text{SAT}(\neg\text{EF } \neg\phi_1)$ 
  end case
end function

```

Fig. 3.28. 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  $\text{SAT}_{\text{EX}}(\phi)$ 
  /* determines the set of states satisfying  $\text{EX } \phi$  */
  local var  $X, Y$ 
begin
     $X := \text{SAT}(\phi)$ ;
     $Y := \text{pre}_{\exists}(X)$ ;
    return  $Y$ 
end

```

Fig. 3.29. The function SAT_{EX} . It computes the states satisfying ϕ by calling SAT . Then, it looks backwards along \rightarrow to find the states satisfying $\text{EX } \phi$.

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 translates ϕ into an equivalent formula of the adequate set chosen.

The algorithm is presented in Figure 3.28 and its subfunctions in Fig-

```

function SATAF( $\phi$ )
  /* determines the set of states satisfying AF  $\phi$  */
  local var X, Y
  begin
    X := S;
    Y := SAT( $\phi$ );
    repeat until X = Y
      begin
        X := Y;
        Y := Y  $\cup$  pre $_{\forall}$ (Y)
      end
    return Y
  end

```

Fig. 3.30. The function SAT_{AF}. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying AF ϕ in the manner described in the labelling algorithm.

```

function SATEU( $\phi, \psi$ )
  /* determines the set of states satisfying E[ $\phi \cup \psi$ ] */
  local var W, X, Y
  begin
    W := SAT( $\phi$ );
    X := S;
    Y := SAT( $\psi$ );
    repeat until X = Y
      begin
        X := Y;
        Y := Y  $\cup$  (W  $\cap$  pre $_{\exists}$ (Y))
      end
    return Y
  end

```

Fig. 3.31. The function SAT_{EU}. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying E[$\phi \cup \psi$] in the manner described in the labelling algorithm.

ures 3.29–3.31. They use program variables X , Y , V and W which are sets of states. The program for SAT handles the easy cases directly and passes more complicated cases on to special procedures, which in turn might call SAT *recursively* on subexpressions. These special procedures rely on implementations of the functions

$$\begin{aligned} \text{pre}_{\exists}(Y) &= \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in Y)\} \\ \text{pre}_{\forall}(Y) &= \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in Y)\} . \end{aligned}$$

‘Pre’ denotes travelling backwards along the transition relation. Both functions compute a pre-image of a set of states. The function pre_{\exists} (instrumental in SAT_{EX} and SAT_{EU}) takes a subset Y of states and returns the set of states which *can* make a transition into Y . The function pre_{\forall} , used in SAT_{AF} , takes a set Y and returns the set of states which make transitions *only* into Y . Observe that pre_{\forall} can be expressed in terms of complementation and pre_{\exists} , as follows:

$$\text{pre}_{\forall}(Y) = S - \text{pre}_{\exists}(S - Y) \quad (3.8)$$

where we write $S - Y$ for the set of all $s \in S$ which are not in Y .

The correctness of this pseudocode and the model checking algorithm is discussed in Section 3.7.

3.6.1.5 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 more often than not 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 state spaces 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), which represent *sets* of states instead of individual states. We study these in Chapter 6 in detail. SMV is implemented using OBDDs.
- Abstraction: one may interpret a model abstractly, uniformly or for a specific property.
- 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.
- 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 end of this chapter.

3.6.2 CTL model checking with fairness

The verification of $\mathcal{M}, s_0 \models \phi$ might fail because the model \mathcal{M} may contain behaviour which is unrealistic, or guaranteed not to occur in the actual system being analysed. For example, in the mutual exclusion case, we expressed that the process `prc` can stay in its critical section (`st=c`) as long as it needs. We modelled this by the non-deterministic assignment

```

next(st) :=
  case
    ...
    (st = c)  : {c,n};
    ...
  esac;

```

However, if we really allow process 2 to stay in its critical section as long as it likes, then we have a path which violates the liveness constraint $\text{AG}(t_1 \rightarrow \text{AF } c_1)$, since, if process 2 stays forever in its critical section, t_1 can be true without c_1 ever becoming true.

We would like to ignore this path, i.e. we would like to assume that the process can stay in its critical section as long as it needs, *but will eventually exit from its critical section* after some finite time.

In LTL, we could handle this by verifying a formula like $\text{FG}\neg c_2 \rightarrow \phi$, where ϕ is the formula we actually want to verify. This whole formula asserts that all paths which satisfy infinitely often $\neg c_2$ also satisfy ϕ . However, we cannot do this in CTL because we cannot write formulas of the form $\text{FG}\neg c_2 \rightarrow \phi$ in CTL. The logic CTL is not expressive enough to allow us to pick out the “fair” paths, i.e., those in which process 2 always eventually leaves its critical section.

It is for that reason that SMV allows us to impose *fairness constraints* on top of the transition system it describes. These assumptions state that a given formula is true infinitely often along every computation path. We call such paths *fair computation paths*. The presence of fairness constraints means that, when evaluating the truth of CTL formulas in specifications, the connectives A and E range only over fair paths.

We therefore impose the fairness constraint that `!st=c` be true infinitely often. This means that, whatever state the process is in, there will be a state in the future in which it is not in its critical section. Similar fairness constraints were used for the Alternating Bit Protocol.

Fairness constraints of the form (where ϕ is a state formula)

Property ϕ is true infinitely often.

are known as *simple* fairness constraints. Other types include those of the form

If ϕ is true infinitely often, then ψ is also true infinitely often.

SMV can deal only with simple fairness constraints; but how does it do that? To answer that, we now explain how we may adapt our model-checking algorithm so that A and E are assumed to range only over fair computation paths.

Definition 3.21 Let $C \stackrel{\text{def}}{=} \{\psi_1, \psi_2, \dots, \psi_n\}$ be a set of n fairness constraints. A computation path $s_0 \rightarrow s_1 \rightarrow \dots$ is fair with respect to these fairness constraints iff for each i there are infinitely many j such that $s_j \models \psi_i$, that is, each ψ_i is true infinitely often along the path. Let us write A_C and E_C for the operators A and E restricted to fair paths.

For example, $\mathcal{M}, s_0 \models A_C G \phi$ iff ϕ is true in every state along all fair paths; and similarly for $A_C F$, $A_C U$, etc. Notice that these operators explicitly depend on the chosen set C of fairness constraints. We already know that $E_C U$, $E_C G$ and $E_C X$ form an adequate set; this can be shown in the same manner as was done for the temporal connectives without fairness constraints (Section 3.4.4). We also have that

$$\begin{aligned} E_C[\phi U \psi] &\equiv E[\phi U (\psi \wedge E_C G \top)] \\ E_C X \phi &\equiv EX(\phi \wedge E_C G \top) . \end{aligned}$$

To see this, observe that a computation path is fair iff any suffix of it is fair. Therefore, we need only provide an algorithm for $E_C G \phi$. It is similar to Algorithm 2 for EG, given earlier in this chapter:

- Restrict the graph to states satisfying ϕ ; of the resulting graph, we want to know from which states there is a fair path.
- Find the maximal *strongly connected components* (SCCs) of the restricted graph;
- Remove an SCC if, for some ψ_i , it does not contain a state satisfying ψ_i . The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach one has a fair path from it.
- Use backwards breadth-first search to find the states on the restricted graph that can reach a fair SCC.

See Figure 3.32. The complexity of this algorithm is $O(n \cdot f \cdot (V + E))$, i.e. still linear in the size of the model and formula.

It should be noted that writing fairness conditions using SMV's FAIRNESS keyword is necessary only for CTL model checking. In the case of LTL,

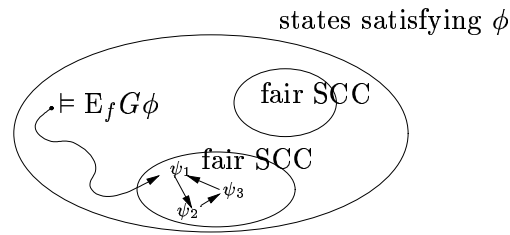


Fig. 3.32. Computing the states satisfying $E_C G \phi$. A state satisfies $E_C G \phi$ iff, in the graph resulting from the restriction to states satisfying ϕ , the state has a fair path from it. A fair path is one which leads to an SCC with a cycle passing through at least one state that satisfies each fairness constraint; in the example, C equals $\{\psi_1, \psi_2, \psi_3\}$.

we can assert the fairness condition as part of the formula to be checked. For example, if we wish to check the LTL formula ψ under the assumption that ϕ is infinitely often true, we check $G F \phi \rightarrow \psi$. This means: all paths satisfying infinitely often ϕ also satisfy ψ . It is not possible to express this in CTL. In particular, any way of adding As or Es to $G F \phi \rightarrow \psi$ will result in a formula with a different meaning from the intended one. For example, $AG AF \phi \rightarrow \psi$ means that if all paths are fair then ψ holds, rather than what was intended: ψ holds along all paths which are fair.

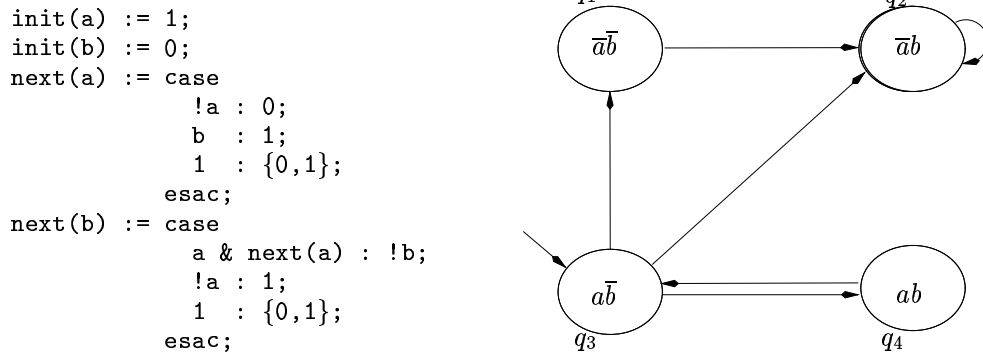
3.6.3 The LTL model checking algorithm

The algorithm presented in the sections above for CTL model checking is quite intuitive: given a system and a CTL formula, it labels states of the system with the subformulas of the formula which are satisfied there. The state-labelling approach is appropriate because subformulas of the formula may be evaluated in states of the system. This is not the case for LTL: subformulas of the formula must be evaluated not in states but *along paths* of the system. Therefore, LTL model checking has to adopt a different strategy.

There are several algorithms for LTL model checking described in the literature. Although they differ in detail, nearly all of them adopt the same basic strategy. We explain that strategy first; then, we describe some algorithms in more detail.

3.6.3.1 The basic strategy

Let $\mathcal{M} = (S, \rightarrow, L)$ be a model, $s \in S$, and ϕ an LTL formula. We determine whether $\mathcal{M}, s \models \phi$, i.e. whether ϕ is satisfied along all paths of \mathcal{M} starting

Fig. 3.33. An SMV program and its model \mathcal{M} .

at s . Almost all LTL model checking algorithms proceed along the following three steps.

Step 1. Construct an automaton, also known as a tableau, for the formula $\neg\phi$. The automaton for ψ is called A_ψ . Thus, we construct $A_{\neg\phi}$. The automaton has a notion of *accepting a trace*. A trace is a sequence of valuations of the propositional atoms. From a path, we can abstract its trace. The construction has the property that for all paths π : $\pi \models \psi$ iff the trace of π is accepted by A_ψ . In other words, the automaton A_ψ encodes precisely the traces which satisfy ψ .

Thus, the automaton $A_{\neg\phi}$ which we construct for $\neg\phi$ has the property that it encodes all the traces satisfying $\neg\phi$; i.e. all the traces which do not satisfy ϕ .

Step 2. Combine the automaton $A_{\neg\phi}$ with the model \mathcal{M} of the system. The combination operation results in a transition system whose paths are *both* paths of the automaton *and* paths of the system.

Step 3. Discover whether there is any path from a state derived from s in the combined transition system. Such a path, if there is one, can be interpreted as a path in \mathcal{M} beginning at s which does not satisfy ϕ .

If there was *no such path*, then output: ‘Yes, $\mathcal{M}, s \models \phi$.’ Otherwise, if there *is such a path*, output ‘No, $\mathcal{M}, s \not\models \phi$.’ In the latter case, the counterexample can be extracted from the path found.

Let us consider an example. The system is described by the SMV program and its model \mathcal{M} , shown in Figure 3.33. We consider the formula $\neg(a \cup b)$. Since it is not the case that all paths of \mathcal{M} satisfy the formula (for example, the path $q_3, q_2, q_2 \dots$ does not satisfy it) we expect the model check to fail.

In accordance with Step 1, we construct an automaton A_{aUb} which characterises precisely the traces which satisfy $a U b$. (We use the fact that $\neg\neg(a U b)$ is equivalent to $a U b$.) Such an automaton is shown in Figure 3.34. We will look at how to construct it later; for now, we just try to understand how and why it works.

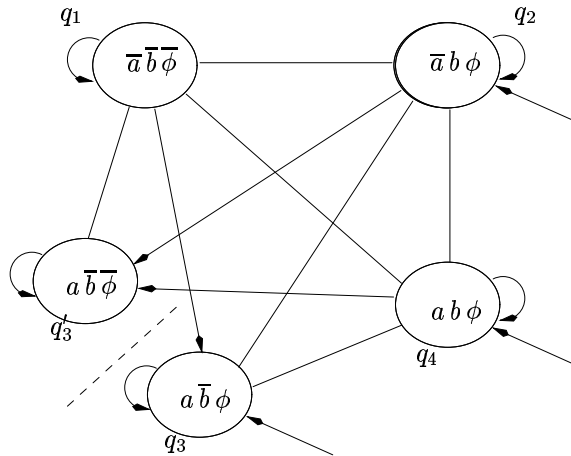


Fig. 3.34. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} a U b$. The transitions with no arrows can be taken in either direction. The acceptance condition is that the path of the automaton cannot loop indefinitely through q_3 .

A trace t is accepted by an automaton like the one of Figure 3.34 if there exists a path π through the automaton such that:

- π starts in an initial state (i.e. one containing ϕ);
- it respects the transition relation of the automaton;
- t is the trace of π ; matches the corresponding state of π ;
- the path respects a certain ‘accepting condition.’ For the automaton of Figure 3.34, the accepting condition is that the path should not end $q_3, q_3, q_3 \dots$, indefinitely.

For example, suppose t is $a\bar{b}, a\bar{b}, a\bar{b}, ab, ab, ab, a\bar{b}, a\bar{b}, a\bar{b}, \dots$, eventually repeating forevermore the state $a\bar{b}$. Then we choose the path $q_3, q_3, q_3, q_4, q_4, q_1, q_3', q_3' \dots$. We start in q_3 because the first state is $a\bar{b}$ and it is an initial state. The next states we choose just follow the valuation of the states of π . For example, at q_1 the next valuation is $a\bar{b}$ and the transitions allow us to choose q_3 or q_3' . We choose q_3' , and loop there forevermore. This path meets the conditions, and therefore the trace t is accepted. Observe that the definition states ‘there exists a path.’ In the example above, there are also paths which don’t meet the conditions:

- Any path beginning q_3, q'_3, \dots doesn't meet the condition that we have to respect the transition relation.
- The path $q_3, q_3, q_3, q_4, q_4, q_1, q_3, q_3 \dots$ doesn't meet the condition that we must not end on a loop of q_3 .

These paths need not bother us, because it is sufficient to find one which does meet the conditions in order to declare that π is accepted.

Why does the automaton of Figure 3.34 work as intended? To understand it, observe that it has enough states to distinguish the values of the propositions – that is, a state for each of the valuations $\{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$, and in fact two states for the valuation $a\bar{b}$. One state for each of $\{\bar{a}\bar{b}, \bar{a}b, ab\}$ is intuitively enough, because those valuations determine whether $a \text{ U } b$ holds. But $a \text{ U } b$ could be false or true in $a\bar{b}$, so we have to consider the two cases. The presence of $\phi \stackrel{\text{def}}{=} a \text{ U } b$ in a state indicates that either we are still expecting ϕ to become true, or we have just obtained it. Whereas $\bar{\phi}$ indicates we no longer expect ϕ , and have not just obtained it. The transitions of the automaton are such that the only way out of q_3 is to obtain b , i.e. to move to q_2 or q_4 . Apart from that, the transitions are liberal, allowing any path to be followed; each of q_1, q_2, q_3 can transition to any valuation, and so can q_3, q'_3 taken together, provided we are careful to choose the right one to enter. The acceptance condition, which allows any path except one looping indefinitely on q_3 , guarantees that the promise of $a \text{ U } b$ to deliver b is eventually fulfilled.

Using this automaton $A_{a \text{ U } b}$, we proceed to step 2. To combine the automaton $A_{a \text{ U } b}$ with the model of the system \mathcal{M} shown in Figure 3.33, it is convenient first to redraw \mathcal{M} with two versions of q_3 ; see Figure 3.35(left). It is an equivalent system; all ways into q_3 now non-deterministically choose q_3 or q'_3 , and whichever one we choose leads to the same successors. But it allows us to superimpose it on $A_{a \text{ U } b}$ and select the transitions common to both, obtaining the combined system of Figure 3.35(right).

Step 3 now asks whether there is a path from q of the combined automaton. As can be seen, there are two kinds of path in the combined system: $q_3, (q_4, q_3)^* q_2, q_2 \dots$, and $q_3, q_4, (q_3, q_4)^* q'_3, q_1, q_2, q_2, \dots$ where $(q_3, q_4)^*$ denotes either the empty string or q_3, q_4 or q_3, q_4, q_3, q_4 etc. Thus, according to Step 3, and as we expected, $\neg(a \text{ U } b)$ is not satisfied in all paths of the original system \mathcal{M} .

3.6.3.2 Constructing the automaton

Let us look in more detail at how the automaton is constructed. Given an LTL formula ϕ , we wish to construct an automaton A_ϕ such that A_ϕ accepts

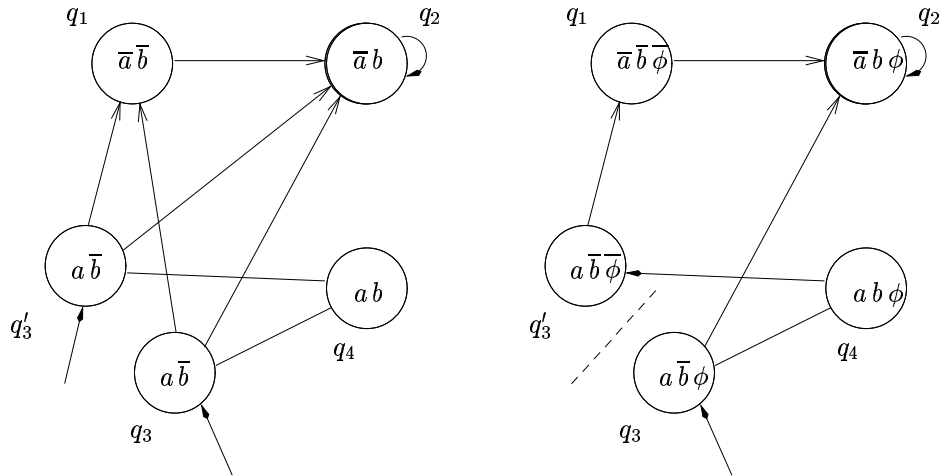


Fig. 3.35. Left: the system \mathcal{M} of Figure 3.33, redrawn with an expanded state space; right: the expanded \mathcal{M} and A_{aUb} combined.

precisely those runs on which ϕ holds. We assume that ϕ contains only the temporal connectives U and X; recall that the other temporal connectives can be written in terms of these two.

Define the *closure* $\mathcal{C}(\phi)$ of formula ϕ as the set of subformulas of ϕ and their complements, identifying $\neg\neg\psi$ and ψ . For example, $\mathcal{C}(a \text{ U } b) = \{a, b, \neg a, \neg b, a \text{ U } b, \neg(a \text{ U } b)\}$. The states of A_ϕ , denoted by q, q' etc, are the maximal subsets of $\mathcal{C}(\phi)$ which satisfy the following conditions:

- For all (non-negated) $\psi \in \mathcal{C}(\phi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- $\psi_1 \vee \psi_2 \in q$ holds iff $\psi_1 \in q$ or $\psi_2 \in q$, whenever $\psi_1 \vee \psi_2 \in \mathcal{C}(\phi)$.
- Conditions for other boolean combinations are similar.
- If $\psi_1 \text{ U } \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1 \text{ U } \psi_2) \in q$, then $\neg\psi_2 \in q$.

Intuitively, these conditions imply that the states of A_ϕ are capable of saying which subformulas of ϕ are true.

The initial states of A_ϕ are those states containing ϕ . For transition relation δ of A_ϕ we have $(q, q') \in \delta$ iff all of the following conditions hold:

- If $X\psi \in q$ then $\psi \in q'$
- If $\neg X\psi \in q$ then $\neg\psi \in q'$
- If $\psi_1 \text{ U } \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \text{ U } \psi_2 \in q'$.
- If $\neg(\psi_1 \text{ U } \psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1 \text{ U } \psi_2) \in q'$.

These last two conditions are justified by the recursion laws

$$\begin{aligned}\psi_1 \text{ U } \psi_2 &= \psi_2 \vee (\psi_1 \wedge \text{X}(\psi_1 \text{ U } \psi_2)) \\ \neg(\psi_1 \text{ U } \psi_2) &= \neg\psi_2 \wedge (\neg\psi_1 \vee \text{X}\neg(\psi_1 \text{ U } \psi_2)) .\end{aligned}$$

In particular, they ensure that whenever some state contains $\psi_1 \text{ U } \psi_2$, subsequent states contain ψ_1 for as long as they do not contain ψ_2 .

As we have defined A_ϕ so far, not all paths through A_ϕ satisfy ϕ . We use additional *acceptance conditions* to guarantee the ‘eventualities’ ψ promised by the formula $\phi \text{ U } \psi$, namely that A_ϕ cannot stay for ever in states satisfying ϕ without ever obtaining ψ . Recall that, for the automaton of Figure 3.34 for $a \text{ U } b$, we stipulated the acceptance condition that the path through the automaton should not end q_3, q_3, \dots .

The acceptance conditions of A_ϕ are defined so that they ensure that every state containing some formula $\chi \text{ U } \psi$ will eventually be followed by some state containing ψ . Let $\chi_1 \text{ U } \psi_1, \dots, \chi_k \text{ U } \psi_k$ be all subformulas of this form in $\mathcal{C}(\phi)$. We stipulate the following acceptance condition: a run is accepted if, for every i such that $1 \leq i \leq k$, the run has infinitely many states satisfying $\neg(\chi_i \text{ U } \psi_i) \vee \psi_i$. To understand why this condition has the desired effect, imagine the circumstances in which it is false. Suppose we have a run having only finitely many states satisfying $\neg(\chi_i \text{ U } \psi_i) \vee \psi_i$. Let us advance through all those finitely many states, taking the suffix of the run none of whose states satisfies $\neg(\chi_i \text{ U } \psi_i) \vee \psi_i$, i.e. all of whose states satisfy $(\chi_i \text{ U } \psi_i) \wedge \neg\psi_i$. That is precisely the sort of run we want to eliminate.

If we carry out this construction on $a \text{ U } b$, we obtain the automaton shown in Figure 3.34. Another example is shown in Figure 3.36, for the formula $(p \text{ U } q) \vee (\neg p \text{ U } q)$. Since that formula has two U subformulas, there are two sets specified in the acceptance condition, namely, the states satisfying $p \text{ U } q$ and the states satisfying $\neg p \text{ U } q$.

3.6.3.3 How LTL model checking is implemented in NuSMV

In the sections above, we described an algorithm for LTL model checking. Given an LTL formula ϕ and a system \mathcal{M} and a state s of \mathcal{M} , we may check whether $\mathcal{M}, s \models \phi$ holds by constructing the automaton $A_{\neg\phi}$, combining it with \mathcal{M} , and checking whether there is a path of the resulting system which satisfies the acceptance condition of $A_{\neg\phi}$.

It is possible to implement the check for such a path in terms of CTL model checking, and this is in fact what NuSMV does. The combined system $\mathcal{M} \times A_{\neg\phi}$ is represented as the system to be model checked in NuSMV, and the formula to be checked is simply $\text{EG} \top$. Thus, we ask the question: does the combined system have a path. The acceptance conditions of $A_{\neg\phi}$

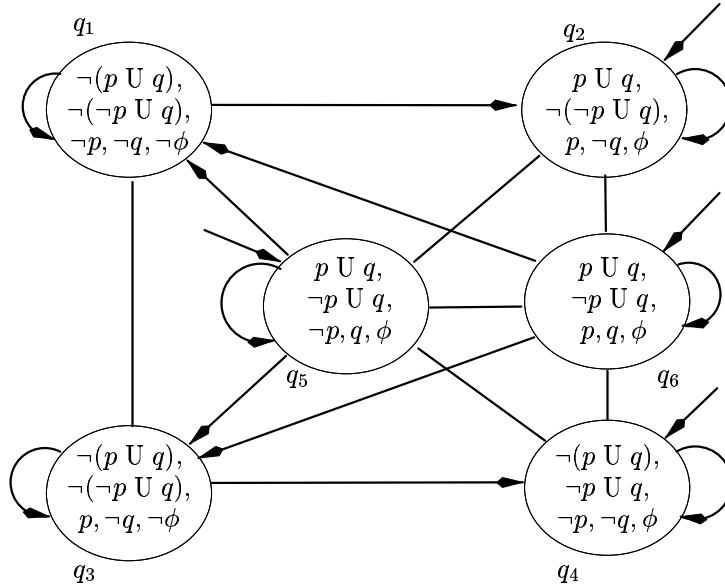


Fig. 3.36. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} (p U q) \vee (\neg p U q)$. The transitions with no arrows can be taken in either direction. The acceptance condition asserts that every run must pass infinitely often through the set $\{q_1, q_3, q_4, q_5, q_6\}$, and also the set $\{q_1, q_2, q_3, q_5, q_6\}$.

are represented as implicit fairness conditions for the CTL model checking procedure. Explicitly, this amounts to asserting ‘FAIRNESS $\neg(\chi U \psi) \vee \psi$ ’ for each formula $\chi U \psi$ occurring in $\mathcal{C}(\phi)$.

3.7 The fixed-point characterisation of CTL

On page 236, we presented an algorithm which, given a CTL formula ϕ and a model $\mathcal{M} = (S, \rightarrow, L)$, computes the set of states $s \in S$ satisfying ϕ . We write this set as $\llbracket \phi \rrbracket$. The algorithm works recursively on the structure of ϕ . For formulas ϕ of height 1 (\perp , \top or p), $\llbracket \phi \rrbracket$ is computed directly. Other formulas are composed of smaller subformulas combined by a connective of CTL. For example, if ϕ is $\psi_1 \vee \psi_2$, then the algorithm computes the sets $\llbracket \psi_1 \rrbracket$ and $\llbracket \psi_2 \rrbracket$ and combines them in a certain way (in this case, by taking the union) in order to obtain $\llbracket \psi_1 \vee \psi_2 \rrbracket$.

The more interesting cases arise when we deal with a formula such as $\text{EX } \psi$, involving a temporal operator. The algorithm computes the set $\llbracket \psi \rrbracket$ and then computes the set of all states which have a transition to a state in $\llbracket \psi \rrbracket$. This is in accord with the semantics of $\text{EX } \psi$: $\mathcal{M}, s \models \text{EX } \psi$ iff there is a state s' with $s \rightarrow s'$ and $\mathcal{M}, s' \models \psi$.

```

function SATEG( $\phi$ )
/* determines the set of states satisfying EG  $\phi$  */
local var X, Y
begin
  Y := SAT( $\phi$ );
  X :=  $\emptyset$ ;
  repeat until X = Y
  begin
    X := Y;
    Y := Y  $\cap$  pre $\exists$ (Y)
  end
  return Y
end

```

Fig. 3.37. The pseudo-code for SAT_{EG}.

For most of these logical operators, we may easily continue this discussion to see that the algorithms work just as expected. However, the cases EU, AF and EG (where we needed to iterate a certain labelling policy until it stabilised) are not so obvious to reason about. The topic of this section is to develop the semantic insights into these operators that allow us to provide a complete proof for their termination and correctness. Inspecting the pseudo-code in Figure 3.28, we see that most of these clauses just do the obvious and correct thing according to the semantics of CTL. For example, try out what SAT does when you call it with $\phi_1 \rightarrow \phi_2$.

Our aim in this section is to prove the termination and correctness of SAT_{AF} and SAT_{EU}. In fact, we will also write a procedure SAT_{EG} and prove its termination and correctness¹. The procedure SAT_{EG} is given in Figure 3.37 and based on the intuitions given in Section 3.6.1.2: note how *deleting* the label if none of the successor states is labelled is coded as *intersecting* the labelled set with the set of states which have a labelled successor.

The semantics of EG ϕ says that $s_0 \models \text{EG } \phi$ holds iff there exists a computation path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ such that $s_i \models \phi$ holds for all $i \geq 0$. We could instead express it as follows: EG ϕ holds if ϕ holds and EG ϕ holds in one of the successor states to the current state. This suggests the equivalence $\text{EG } \phi \equiv \phi \wedge \text{EX EG } \phi$ which can easily be proved from the semantic definitions of the connectives.

Observing that $\llbracket \text{EX } \psi \rrbracket = \text{pre}_{\exists}(\llbracket \psi \rrbracket)$ we see that the equivalence above can be written as $\llbracket \text{EG } \phi \rrbracket = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(\llbracket \text{EG } \phi \rrbracket)$. This does not look like a very promising way of calculating EG ϕ , because we need to know EG ϕ in

¹ Section 3.6.1.4 handles EG ϕ by translating it into $\neg \text{AF } \neg \phi$, but we already noted in Section 3.6.1.2 that EG could be handled directly.

order to work out the right-hand side. Fortunately, there is a way around this apparent circularity, known as computing fixed points, and that is the subject of this section.

3.7.1 Monotone functions

Definition 3.22 Let S be a set of states and $F: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ a function on the power set of S .

1. We say that F is monotone iff $X \subseteq Y$ implies $F(X) \subseteq F(Y)$ for all subsets X and Y of S .
2. A subset X of S is called a fixed point of F iff $F(X) = X$.

For an example, let $S \stackrel{\text{def}}{=} \{s_0, s_1\}$ and $F(Y) \stackrel{\text{def}}{=} Y \cup \{s_0\}$ for all subsets Y of S . Since $Y \subseteq Y'$ implies $Y \cup \{s_0\} \subseteq Y' \cup \{s_0\}$, we see that F is monotone. The fixed points of F are all subsets of S containing s_0 . Thus, F has two fixed points, the sets $\{s_0\}$ and $\{s_0, s_1\}$. Notice that F has a least ($= \{s_0\}$) and a greatest ($= \{s_0, s_1\}$) fixed point.

An example of a function $G: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, which is *not* monotone, is given by

$$G(Y) \stackrel{\text{def}}{=} \text{if } Y = \{s_0\} \text{ then } \{s_1\} \text{ else } \{s_0\} .$$

So G maps $\{s_0\}$ to $\{s_1\}$ and *all other sets* to $\{s_0\}$. The function G is not monotone since $\{s_0\} \subseteq \{s_0, s_1\}$ but $G(\{s_0\}) = \{s_1\}$ is *not* a subset of $G(\{s_0, s_1\}) = \{s_0\}$. Note that G has *no* fixed points whatsoever.

The reasons for exploring monotone functions on $\mathcal{P}(S)$ in the context of proving the correctness of SAT are

1. that monotone functions *always* have a least and a greatest fixed point,
2. that the meanings of EG, AF and EU can be expressed via greatest, respectively least, fixed points of monotone functions on $\mathcal{P}(S)$,
3. that these fixed-points can be easily computed and
4. that the procedures SAT_{EU} and SAT_{AF} code up such fixed-point computations, and are correct by item 2.

Notation 3.23 $F^i(X)$ means

$$\underbrace{F(F(\dots F(X)\dots))}_{i \text{ times}}$$

Thus, the function F^i is just ' F applied i many times.'

For example, for the function $F(Y) \stackrel{\text{def}}{=} Y \cup \{s_0\}$, we obtain $F^2(Y) = F(F(Y)) = (Y \cup \{s_0\}) \cup \{s_0\} = Y \cup \{s_0\} = F(Y)$. In this case, $F^2 = F$ and therefore $F^i = F$ for all $i \geq 1$. It is not always the case that the sequence of functions (F^1, F^2, F^3, \dots) stabilises in such a way. For example, this won't happen for the function G defined above (see exercise 1(d) on page 263). The following fact is a special case of a fundamental insight, often referred to as the Knaster-Tarski Theorem.

Theorem 3.24 Let S be a set $\{s_0, s_1, \dots, s_n\}$ with $n + 1$ elements. If $F: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is a monotone function, then $F^{n+1}(\emptyset)$ is the least fixed point of F and $F^{n+1}(S)$ is the greatest fixed point of F .

Proof: Since $\emptyset \subseteq F(\emptyset)$, we get $F(\emptyset) \subseteq F(F(\emptyset))$, i.e. $F^1(\emptyset) \subseteq F^2(\emptyset)$, for F is monotone. We can now use mathematical induction to show that

$$F^1(\emptyset) \subseteq F^2(\emptyset) \subseteq F^3(\emptyset) \subseteq \dots \subseteq F^i(\emptyset)$$

for all $i \geq 1$. In particular, taking $i \stackrel{\text{def}}{=} n + 1$, we claim that one of the expressions $F^k(\emptyset)$ above is already a fixed point of F . Otherwise, $F^1(\emptyset)$ needs to contain at least one element (for then $\emptyset \neq F(\emptyset)$). By the same token, $F^2(\emptyset)$ needs to have at least two elements since it must be bigger than $F^1(\emptyset)$. Continuing this argument, we see that $F^{n+2}(\emptyset)$ would have to contain at least $n + 2$ many elements. The latter is impossible since S has only $n + 1$ elements. Therefore, $F(F^k(\emptyset)) = F^k(\emptyset)$ for some $0 \leq k \leq n + 1$, which readily implies that $F^{n+1}(\emptyset)$ is a fixed point of F as well.

Now suppose that X is another fixed point of F . We need to show that $F^{n+1}(\emptyset)$ is a subset of X ; but, since $\emptyset \subseteq X$, we conclude $F(\emptyset) \subseteq F(X) = X$, for F is monotone and X a fixed point of F . By induction, we obtain $F^i(\emptyset) \subseteq X$ for all $i \geq 0$. So, for $i \stackrel{\text{def}}{=} n + 1$, we get $F^{n+1}(\emptyset) \subseteq X$.

The proof of the statements about the greatest fixed point is dual to the one above. Simply replace \subseteq by \supseteq , \emptyset by S and 'bigger' by 'smaller.' \square

This theorem about the existence of least and greatest fixed points of monotone functions $F: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ not only asserted the existence of such fixed points; it also provided a recipe for computing them, and correctly so. For example, in computing the least fixed point of F , all we have to do is apply F to the empty set \emptyset and keep applying F to the result until the latter becomes invariant under the application of F . The theorem above further ensures that this process is *guaranteed to terminate*. Moreover, we can specify an upper bound $n + 1$ to the worst-case number of

iterations necessary for reaching this fixed point, assuming that S has $n + 1$ elements.

3.7.2 The correctness of SAT_{EG}

We saw at the end of the last section that $\llbracket \text{EG } \phi \rrbracket = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(\llbracket \text{EG } \phi \rrbracket)$. This implies that $\text{EG } \phi$ is a fixed point of the function $F(X) = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(X)$. In fact, F is monotone, $\text{EG } \phi$ is its greatest fixed point and therefore $\text{EG } \phi$ can be computed using Theorem 3.24.

Theorem 3.25 Let F be as defined above and let S have $n + 1$ elements. Then F is monotone, $\llbracket \text{EG } \phi \rrbracket$ is the greatest fixed-point of F , and $\llbracket \text{EG } \phi \rrbracket = F^{n+1}(S)$.

Proof:

1. In order to show that F is monotone, we take any two subsets X and Y of S such that $X \subseteq Y$ and we need to show that $F(X)$ is a subset of $F(Y)$. Given s_0 such that there is some $s_1 \in X$ with $s_0 \rightarrow s_1$, we certainly have $s_0 \rightarrow s_1$, where $s_1 \in Y$, for X is a subset of Y . Thus, we showed $\text{pre}_{\exists}(X) \subseteq \text{pre}_{\exists}(Y)$ from which we readily conclude that $F(X) = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(X) \subseteq \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(Y) = F(Y)$.
2. We have already seen that $\llbracket \text{EG } \phi \rrbracket$ is a fixed point of F . To show that it is the greatest fixed point, it suffices to show here that any set X with $F(X) = X$ has to be contained in $\llbracket \text{EG } \phi \rrbracket$. So let s_0 be an element of such a fixed point X . We need to show that s_0 is in $\llbracket \text{EG } \phi \rrbracket$ as well. For that we use the fact that

$$s_0 \in X = F(X) = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(X)$$

to infer that $s_0 \in \llbracket \phi \rrbracket$ and $s_0 \rightarrow s_1$ for some $s_1 \in X$; but, since s_1 is in X , we may apply that same argument to $s_1 \in X = F(X) = \llbracket \phi \rrbracket \cap \text{pre}_{\exists}(X)$ and we get $s_1 \in \llbracket \phi \rrbracket$ and $s_1 \rightarrow s_2$ for some $s_2 \in X$. By mathematical induction, we can therefore construct an infinite path $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n \rightarrow s_{n+1} \rightarrow \dots$ such that $s_i \in \llbracket \phi \rrbracket$ for all $i \geq 0$. By the definition of $\llbracket \text{EG } \phi \rrbracket$, this entails $s_0 \in \llbracket \text{EG } \phi \rrbracket$.

3. The last item is now immediately accessible from the previous one and Theorem 3.24.

□

Now we can see that the procedure SAT_{EG} is correctly coded and terminates. First, note that the line $Y := Y \cap \text{pre}_{\exists}(Y)$ in the procedure SAT_{EG}

(Figure 3.37) could be changed to $Y := \text{SAT}(\phi) \cap \text{pre}_\exists(Y)$ without changing the effect of the procedure. To see this, note that the first time round the loop, Y is $\text{SAT}(\phi)$; and in subsequent loops, $Y \subseteq \text{SAT}(\phi)$, so it doesn't matter whether we intersect with Y or $\text{SAT}(\phi)$ ¹. With the change, it is clear that SAT_{EG} is calculating the greatest fixed point of F ; therefore its correctness follows from Theorem 3.25.

3.7.3 The correctness of SAT_{EU}

Proving the correctness of SAT_{EU} is similar. We start by noting the equivalence $\text{E}[\phi \cup \psi] \equiv \psi \vee (\phi \wedge \text{EX E}[\phi \cup \psi])$ and we write it as $\llbracket \text{E}[\phi \cup \psi] \rrbracket = \llbracket \psi \rrbracket \cup (\llbracket \phi \rrbracket \cap \text{pre}_\exists \llbracket \text{E}[\phi \cup \psi] \rrbracket)$. That tells us that $\llbracket \text{E}[\phi \cup \psi] \rrbracket$ is a fixed point of the function $G(X) = \llbracket \psi \rrbracket \cup (\llbracket \phi \rrbracket \cap \text{pre}_\exists(X))$. As before, we can prove that this function is monotone. It turns out that $\llbracket \text{E}[\phi \cup \psi] \rrbracket$ is its *least* fixed point and that the function SAT_{EU} is actually computing it in the manner of Theorem 3.24.

Theorem 3.26 Let G be defined as above and let S have $n + 1$ elements. Then G is monotone, $\llbracket \text{E}[\phi \cup \psi] \rrbracket$ is the least fixed-point of G , and we have $\llbracket \text{E}[\phi \cup \psi] \rrbracket = G^{n+1}(\emptyset)$.

Proof:

1. Again, we need to show that $X \subseteq Y$ implies $G(X) \subseteq G(Y)$; but that is essentially the same argument as for F , since the function

¹ If you are sceptical, try computing the values Y_0, Y_1, Y_2, \dots , where Y_i represents the value of Y after i iterations round the loop. The program before the change computes as follows:

$$\begin{aligned} Y_0 &= \text{SAT}(\phi) \\ Y_1 &= Y_0 \cap \text{pre}_\exists(Y_0) \\ Y_2 &= Y_1 \cap \text{pre}_\exists(Y_1) \\ &= Y_0 \cap \text{pre}_\exists(Y_0) \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0)) \\ &= Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0)). \end{aligned}$$

The last of these equalities follows from the monotonicity of pre_\exists .

$$\begin{aligned} Y_3 &= Y_2 \cap \text{pre}_\exists(Y_2) \\ &= Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0)) \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0))) \\ &= Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0))). \end{aligned}$$

Again the last one follows by monotonicity. Now look at what the program does after the change:

$$\begin{aligned} Y_0 &= \text{SAT}(\phi) \\ Y_1 &= \text{SAT}(\phi) \cap \text{pre}_\exists(Y_0) \\ &= Y_0 \cap \text{pre}_\exists(Y_0) \\ Y_2 &= Y_0 \cap \text{pre}_\exists(Y_1) \\ Y_3 &= Y_0 \cap \text{pre}_\exists(Y_1) \\ &= Y_0 \cap \text{pre}_\exists(Y_0 \cap \text{pre}_\exists(Y_0)). \end{aligned}$$

A formal proof would follow by induction on i .

which sends X to $\text{pre}_\exists(X)$ is monotone and all that G now does is to perform the intersection and union of that set with constant sets $\llbracket\phi\rrbracket$ and $\llbracket\psi\rrbracket$.

2. If S has $n+1$ elements, then the least fixed point of G equals $G^{n+1}(\emptyset)$ by Theorem 3.24. Therefore it suffices to show that this set equals $\llbracket\text{E}(\phi \text{ U } \psi)\rrbracket$. Simply observe what kind of states we obtain by iterating G on the empty set \emptyset : $G^1(\emptyset) = \llbracket\psi\rrbracket \cup (\llbracket\phi\rrbracket \cap \text{pre}_\exists(\llbracket\phi\rrbracket)) = \llbracket\psi\rrbracket \cup (\llbracket\phi\rrbracket \cap \emptyset) = \llbracket\psi\rrbracket \cup \emptyset = \llbracket\psi\rrbracket$, which are all states $s_0 \in \llbracket\text{E}(\phi \text{ U } \psi)\rrbracket$, where we chose $i = 0$ according to the definition of Until. Now,

$$G^2(\emptyset) = \llbracket\psi\rrbracket \cup (\llbracket\phi\rrbracket \cap \text{pre}_\exists(G^1(\emptyset)))$$

tells us that the elements of $G^2(\emptyset)$ are all those $s_0 \in \llbracket\text{E}(\phi \text{ U } \psi)\rrbracket$ where we chose $i \leq 1$. By mathematical induction, we see that $G^{k+1}(\emptyset)$ is the set of all states s_0 for which we chose $i \leq k$ to secure $s_0 \in \llbracket\text{E}(\phi \text{ U } \psi)\rrbracket$. Since this holds for all k , we see that $\llbracket\text{E}(\phi \text{ U } \psi)\rrbracket$ is nothing but the union of all sets $G^{k+1}(\emptyset)$ with $k \geq 0$; but, since $G^{n+1}(\emptyset)$ is a fixed point of G , we see that this union is just $G^{n+1}(\emptyset)$.

□

The correctness of the coding of SAT_{EU} follows similarly to that of SAT_{EG} . We change the line $Y := Y \cup (W \cap \text{pre}_\exists(Y))$ into $Y := \text{SAT}(\psi) \cup (W \cap \text{pre}_\exists(Y))$ and observe that this does not change the result of the procedure, because the first time round the loop, Y is $\text{SAT}(\psi)$; and, since Y is always increasing, it makes no difference whether we perform a union with Y or with $\text{SAT}(\psi)$. Having made that change, it is then clear that SAT_{EU} is just computing the least fixed point of G using Theorem 3.24.

We illustrate these results about the functions F and G above through an example. Consider the system in Figure 3.38. We begin by computing the set $\llbracket\text{EF } p\rrbracket$. By the definition of EF this is just $\llbracket\text{E}(\top \text{ U } p)\rrbracket$. So we have $\phi_1 \stackrel{\text{def}}{=} \top$ and $\phi_2 \stackrel{\text{def}}{=} p$. From Figure 3.38, we obtain $\llbracket p\rrbracket = \{s_3\}$ and of course $\llbracket\top\rrbracket = S$. Thus, the function G above equals $G(X) = \{s_3\} \cup \text{pre}_\exists(X)$. Since $\llbracket\text{E}(\top \text{ U } p)\rrbracket$ equals the least fixed point of G , we need to iterate G on \emptyset until this process stabilises. First, $G^1(\emptyset) = \{s_3\} \cup \text{pre}_\exists(\emptyset) = \{s_3\}$. Second, $G^2(\emptyset) = G(G^1(\emptyset)) = \{s_3\} \cup \text{pre}_\exists(\{s_3\}) = \{s_1, s_3\}$. Third, $G^3(\emptyset) = G(G^2(\emptyset)) = \{s_3\} \cup \text{pre}_\exists(\{s_1, s_3\}) = \{s_0, s_1, s_2, s_3\}$. Fourth, $G^4(\emptyset) = G(G^3(\emptyset)) = \{s_3\} \cup \text{pre}_\exists(\{s_0, s_1, s_2, s_3\}) = \{s_0, s_1, s_2, s_3\}$. Therefore, $\{s_0, s_1, s_2, s_3\}$ is the least fixed point of G , which equals $\llbracket\text{E}(\top \text{ U } p)\rrbracket$ by Theorem 3.20. But then $\llbracket\text{E}(\top \text{ U } p)\rrbracket = \llbracket\text{EF } p\rrbracket = \llbracket\text{EF } p\rrbracket$.

The other example we study is the computation of the set $\llbracket\text{EG } q\rrbracket$. By Theorem 3.25, that set is the greatest fixed point of the function F above,

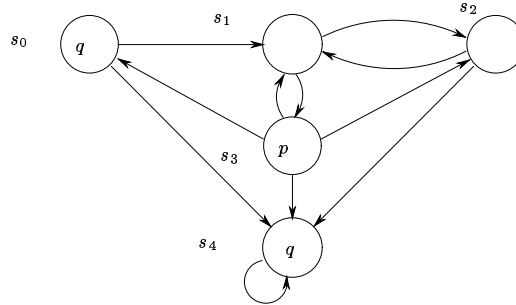


Fig. 3.38. A system for which we compute invariants.

where $\phi \stackrel{\text{def}}{=} q$. From Figure 3.38 we see that $\llbracket q \rrbracket = \{s_0, s_4\}$ and so $F(X) = \llbracket q \rrbracket \cap \text{pre}_{\exists}(X) = \{s_0, s_4\} \cap \text{pre}_{\exists}(X)$. Since $\llbracket \text{EG } q \rrbracket$ equals the greatest fixed point of F , we need to iterate F on S until this process stabilises. First, $F^1(S) = \{s_0, s_4\} \cap \text{pre}_{\exists}(S) = \{s_0, s_4\} \cap S$ since every s has some s' with $s \rightarrow s'$. Thus, $F^1(S) = \{s_0, s_4\}$.

Second, $F^2(S) = F(F^1(S)) = \{s_0, s_4\} \cap \text{pre}_{\exists}(\{s_0, s_4\}) = \{s_0, s_4\}$. Therefore, $\{s_0, s_4\}$ is the greatest fixed point of F , which equals $\llbracket \text{EG } q \rrbracket$ by Theorem 3.25.

3.8 Exercises

Exercises 3.1

1. Read Section 2.7 in case you have not yet done so and classify Alloy and its constraint analyzer according to the classification criteria for formal methods proposed on page 180.
2. Visit and browse the web sites ¹ and ² to find formal methods that interest you for whatever reason. Then classify them according to the criteria from page 180.

Exercises 3.2

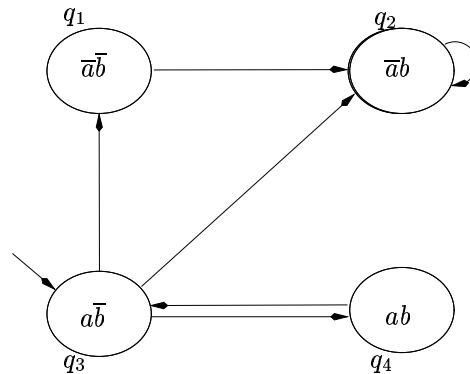
1. Draw parse trees for the LTL formulas:

- (a) $F p \wedge G q \rightarrow p W r$
- (b) $F (p \rightarrow G r) \vee \neg q U p$
- (c) $p W (q W r)$
- (d) $G F p \rightarrow F (q \vee s)$

2. Consider the system of Figure 3.39. For each of the formulas ϕ :

¹ www.afm.sbu.ac.uk

² www.cs.indiana.edu/formal-methods-education/

Fig. 3.39. A model \mathcal{M} .

- (a) $G a$
- (b) $a U b$
- (c) $a U X (a \wedge \neg b)$
- (d) $X \neg b \wedge G (\neg a \vee \neg b)$
- (e) $X (a \wedge b) \wedge F (\neg a \wedge \neg b)$

- (i) Find a path from the initial state q_3 which satisfies ϕ .
- (ii) Determine whether $\mathcal{M}, q_3 \models \phi$.

3. Working from the clauses of Definition 3.1 (page 184), prove the equivalences

$$\begin{aligned} \phi U \psi &\equiv \phi W \psi \wedge F \psi \\ \phi W \psi &\equiv \phi U \psi \vee G \phi \\ \phi W \psi &\equiv \psi R (\phi \vee \psi) \\ \phi R \psi &\equiv \psi W (\phi \wedge \psi). \end{aligned}$$

- 4. Prove that $\phi U \psi \equiv \psi R (\phi \vee \psi) \wedge F \psi$.
- 5. List all subformulas of the LTL formula $\neg p U (F r \vee G \neg q \rightarrow q W \neg r)$.
- 6. ‘Morally’ there ought to be a dual for W. Work out what it might mean, and then pick a symbol based on the first letter of the meaning.
- 7. Prove that for all paths π of all models, $\pi \models \phi W \psi \wedge F \psi$ implies $\pi \models \phi U \psi$. That is, prove the remaining half of equivalence (3.2) on page 194.
- 8. Recall the algorithm NNF on page 64 which computes the negation normal form of propositional logic formulas. Extend this algorithm to LTL: you need to add program clauses for the additional connectives

X, F, G and U, R and W; these clauses have to animate the semantic equivalences that we presented in this section.

Exercises 3.3

1. Consider the model in Figure 3.9 (page 201).
 - * (a) Verify that $G(\text{req} \rightarrow F \text{ busy})$ holds in all initial states.
 - (b) Does $\neg(\text{req} U \neg \text{busy})$ hold in all initial states of that model?
 - (c) NuSMV has the capability of referring to the next value of a declared variable v by writing `next(v)`. Consider the model obtained from Figure 3.9 by removing the self-loop on state `!req & busy`. Use the NuSMV feature `next(...)` to code that modified model as an NuSMV program with the specification $G(\text{req} \rightarrow F \text{ busy})$. Then run it.
2. Verify Remark 3.11 from page 199.
- * 3. Draw the transition system described by the ABP program.

Remarks: There are 28 reachable states of the ABP program. (Looking at the program, you can see that the state is described by nine boolean variables, namely `S.st`, `S.message1`, `S.message2`, `R.st`, `R.ack`, `R.expected`, `msg_chan.output1`, `msg_chan.output2` and finally `ack_chan.output`. Therefore, there are $2^9 = 512$ states in total. However, only 28 of them can be reached from the initial state by following a finite path.)

If you abstract away from the contents of the message (e.g. by setting `S.message1` and `msg_chan.output1` to be constant 0), then there are only 12 reachable states. This is what you are asked to draw.

Exercises 3.4

1. Write the parse trees for the following CTL formulas:
 - * (a) $EG r$
 - * (b) $AG (q \rightarrow EG r)$
 - * (c) $A[p U EF r]$
 - * (d) $EF EG p \rightarrow AF r$, recall Convention 3.13
 - (e) $A[p U A[q U r]]$
 - (f) $E[A[p U q] U r]$
 - (g) $AG (p \rightarrow A[p U (\neg p \wedge A[\neg p U q]))]$.
2. Explain why the following are not well-formed CTL formulas:
 - * (a) $FG r$

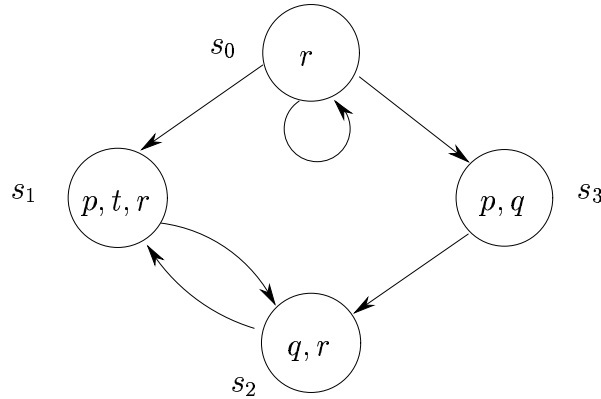


Fig. 3.40. A model with four states.

- (b) XXr
 - (c) $A\neg G\neg p$
 - (d) $F[rUq]$
 - (e) $EXXr$
 - * (f) $AEF r$
 - * (g) $AF[(rUq) \wedge (pUr)]$.
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) $\neg(\neg p) \vee (r \wedge s)$
 - (b) Xq
 - * (c) $\neg AXq$
 - (d) $pU(AX\perp)$
 - * (e) $E[(AXq)U(\neg(\neg p) \vee (\top \wedge s))]$
 - * (f) $(Fr) \wedge (AGq)$
 - (g) $\neg(AGq) \vee (EGq)$.
- * 4. List all subformulas of the formula $AG(p \rightarrow A[pU(\neg p \wedge A[\neg pUq])])$.
5. Does $E[\text{req}U\neg\text{busy}]$ hold in all initial states of the model in Figure 3.9 on page 201?
6. Consider the system \mathcal{M} in Figure 3.40.
- (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) Determine whether $\mathcal{M}, s_0 \models \phi$ and $\mathcal{M}, s_2 \models \phi$ hold and justify your answer, where ϕ is the LTL or CTL formula:

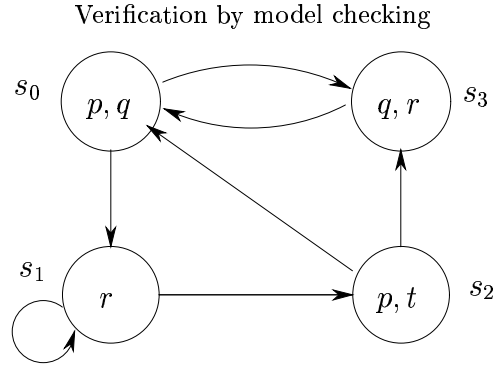


Fig. 3.41. Another model with four states.

- * (i) $\neg p \rightarrow r$
- (ii) $F t$
- * (iii) $\neg EG r$
- (iv) $E (t U q)$
- (v) $F q$
- (vi) $EF q$
- (vii) $EG r$
- (viii) $G (r \vee q)$.

7. Let $\mathcal{M} = (S, \rightarrow, L)$ be any model for CTL and let $\llbracket \phi \rrbracket$ denote the set of all $s \in S$ such that $\mathcal{M}, s \models \phi$. Prove the following set identities by inspecting the clauses of Definition 3.15 from page 220.

- * (a) $\llbracket \top \rrbracket = S$,
- (b) $\llbracket \perp \rrbracket = \emptyset$
- (c) $\llbracket \neg \phi \rrbracket = S - \llbracket \phi \rrbracket$,
- (d) $\llbracket \phi_1 \wedge \phi_2 \rrbracket = \llbracket \phi_1 \rrbracket \cap \llbracket \phi_2 \rrbracket$
- (e) $\llbracket \phi_1 \vee \phi_2 \rrbracket = \llbracket \phi_1 \rrbracket \cup \llbracket \phi_2 \rrbracket$
- * (f) $\llbracket \phi_1 \rightarrow \phi_2 \rrbracket = (S - \llbracket \phi_1 \rrbracket) \cup \llbracket \phi_2 \rrbracket$
- * (g) $\llbracket AX \phi \rrbracket = S - \llbracket EX \neg \phi \rrbracket$
- (h) $\llbracket A(\phi_2 U \phi_2) \rrbracket = \llbracket \neg(E(\neg \phi_1 U (\neg \phi_1 \wedge \neg \phi_2)) \vee EG \neg \phi_2) \rrbracket$.

8. Consider the model \mathcal{M} in Figure 3.41. Check whether $\mathcal{M}, s_0 \models \phi$ and $\mathcal{M}, s_2 \models \phi$ hold for the CTL formulas ϕ :

- (a) $AF q$
- (b) $AG (EF (p \vee r))$
- (c) $EX (EX r)$
- (d) $AG (AF q)$.

- * 9. The meaning of the temporal operators F, G and U in LTL and AU, EU, AG, EG, AF and EF in CTL 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 on page 217 to define such (six) modified connectives as derived operators in CTL.
10. 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) $EF \phi$ and $EG \phi$
 - * (b) $EF \phi \vee EF \psi$ and $EF (\phi \vee \psi)$
 - * (c) $AF \phi \vee AF \psi$ and $AF (\phi \vee \psi)$
 - (d) $AF \neg \phi$ and $\neg EG \phi$
 - * (e) $EF \neg \phi$ and $\neg AF \phi$
 - (f) $A[\phi_1 U A[\phi_2 U \phi_3]]$ and $A[A[\phi_1 U \phi_2] U \phi_3]$, hint: it might make it simpler if you think first about models that have just one path
 - (g) \top and $AG \phi \rightarrow EG \phi$
 - * (h) \top and $EG \phi \rightarrow AG \phi$.
11. Find operators to replace the ? marks, to make the following equivalences.
- * (a) $AG (\phi \wedge \psi) \equiv AG \phi ? AG \psi$.
 - (b) $EF \neg \phi \equiv \neg ?? \phi$
12. State explicitly the meaning of the temporal connectives AR etc., as defined on page 226.
13. Prove the equivalences (3.6) on page 225.
- * 14. 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 $\{\perp, \neg, \wedge, AF, EU, EX\}$.

Exercises 3.5

1. Express the following properties in CTL and LTL whenever possible. If neither is possible, try to express the property in CTL*:
- * (a) Whenever p is followed by q (after finitely many steps), then the system enters an ‘interval’ in which no r occurs until t .

- (b) Event p precedes s and t on all computation paths. (You may find it easier to code the negation of that specification first.)
 - (c) After p , q is never true. (Where this constraint is meant to apply on all computation paths.)
 - (d) Between the events q and r , event p is never true.
 - (e) Transitions to states satisfying p occur at most twice.
 - * (f) Property p is true for every second state along a path.
2. Explain in detail why the LTL and CTL formulas for the practical specification patterns of pages 192 and 223 capture the stated ‘informal’ properties expressed in plain English.
 3. Consider the set of LTL/CTL formulas $\mathcal{F} = \{F p \rightarrow F q, AF p \rightarrow AF q, AG (p \rightarrow AF q)\}$.
 - (a) Is there a model such that all formulas hold in it?
 - (b) For each $\phi \in \mathcal{F}$, is there a model such that ϕ is the only formula in \mathcal{F} satisfied in that model?
 - (c) Find a model in which no formula of \mathcal{F} holds.
 4. Consider the CTL formula $AG (p \rightarrow AF (s \wedge AX (AF t)))$. Explain what exactly it expresses in terms of the order of occurrence of events p , s and t .
 5. Extend the algorithm NNF from page 64 which computes the negation normal form of propositional logic formulas to CTL*. Since CTL* is defined in terms of two syntactic categories (state formulas and path formulas), this requires two separate versions of NNF which call each other in a way that is reflected by the syntax of CTL* given on page 227.
 6. Find a transition system which distinguishes the following pairs of CTL* formulas, i.e. show that they are not equivalent:
 - (a) $AF G p$ and $AF AG p$
 - * (b) $AG F p$ and $AG EF p$
 - (c) $A[(p U r) \vee (q U r)]$ and $A[(p \vee q) U r]$
 - * (d) $A[X p \vee XX p]$ and $AX p \vee AX AX p$
 - (e) $E[G F p]$ and $EG EF p$.
 7. The translation from CTL with boolean combinations of path formulas to plain CTL introduced in Section 3.5.1 is not complete. Invent CTL equivalents for:
 - * (a) $E[F p \wedge (q U r)]$
 - * (b) $E[F p \wedge G q]$.

In this way, we have dealt with all formulas of the form $E[\phi \wedge \psi]$. Formulas of the form $E[\phi \vee \psi]$ can be rewritten as $E[\phi] \vee E[\psi]$ and $A[\phi]$ can be written $\neg E[\neg\phi]$.

Use this translation to write the following in CTL:

(c) $E[(p \text{ U } q) \wedge F p]$

* (d) $A[(p \text{ U } q) \wedge G p]$

* (e) $A[F p \rightarrow F q]$.

8. The aim of this exercise is to demonstrate the expansion given for AW at the end of the last section, i.e. $A[p \text{ W } q] \equiv \neg E[\neg q \text{ U } \neg(p \vee q)]$.

- (a) Show that the following LTL formulas are valid (i.e. true in any state of any model):

(i) $\neg q \text{ U } (\neg p \wedge \neg q) \rightarrow \neg G p$

(ii) $G \neg q \wedge F \neg p \rightarrow \neg q \text{ U } (\neg p \wedge \neg q)$.

- (b) Expand $\neg((p \text{ U } q) \vee G p)$ using de Morgan rules and the LTL equivalence $\neg(\phi \text{ U } \psi) \equiv (\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)) \vee \neg F \psi$.

- (c) Using your expansion and the facts (i) and (ii) above, show $\neg((p \text{ U } q) \vee G p) \equiv \neg q \text{ U } \neg(p \wedge q)$ and hence show that the desired expansion of AW above is correct.

Exercises 3.6

- * 1. Verify ϕ_1 to ϕ_4 for the transition system given in Figure 3.11 on page 208. Which of them require the fairness constraints of the SMV program in Figure 3.10?
2. Try to write a CTL formula that enforces non-blocking and no strict sequencing at the same time, for the SMV program in Figure 3.10 (page 205).
- * 3. Apply the labelling algorithm to check the formulas ϕ_1 , ϕ_2 , ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.7 (page 197).
4. Apply the labelling algorithm to check the formulas ϕ_1 , ϕ_2 , ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.8 (page 199).
5. Prove that (3.8) on page 238 holds in all models. Does your proof require that for every state s there is some state s' with $s \rightarrow s'$?
6. Inspecting the definition of the labelling algorithm, explain what happens if you perform it on the formula $p \wedge \neg p$ (in any state, in any model).

7. Modify the pseudo-code for **SAT** on page 236 by writing a special procedure for $\text{AG } \psi_1$, without rewriting it in terms of other formulas¹.
- * 8. Write the pseudo-code for SAT_{EG} , based on the description in terms of deleting labels given in Section 3.6.1.2.
- * 9. 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.
10. Use the definition of \models between states and CTL formulas to explain why $s \models \text{AG AF } \phi$ means that ϕ is true infinitely often along every path starting at s .
- * 11. Show that a CTL formula ϕ is true on infinitely many states of a computation path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ iff for all $n \geq 0$ there is some $m \geq n$ such that $s_m \models \phi$.
12. Run the NuSMV system on some examples. Try commenting out, or deleting, some of the fairness constraints, if applicable, and see the counter examples NuSMV generates. NuSMV is very easy to run.
13. In the one-bit channel, there are two fairness constraints. We could have written this as a single one, inserting '&' between **running** and the long formula, or we could have separated the long formula into two and made it into a total of three fairness constraints.

In general, what is the difference between the single fairness constraint $\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_n$ and the n fairness constraints $\phi_1, \phi_2, \dots, \phi_n$? Write an SMV program with a fairness constraint **a & b** which is not equivalent to the two fairness constraints **a** and **b**. (You can actually do it in four lines of SMV.)

14. 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?
- * 15. Compute the $\text{E}_C\text{G } \top$ labels for Figure 3.11, given the fairness constraints of the code in Figure 3.10 on page 205.

Exercises 3.7

1. Consider the functions

$$H_1, H_2, H_3: \mathcal{P}(\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}) \rightarrow \mathcal{P}(\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\})$$

¹ Question: will your routine be more like the routine for AF, or more like that for EG on page 233? Why?

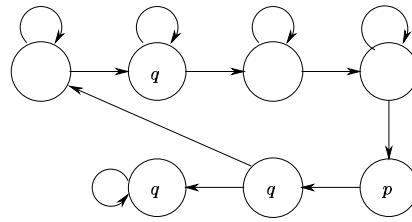


Fig. 3.42. Another system for which we compute invariants.

defined by

$$H_1(Y) \stackrel{\text{def}}{=} Y - \{1, 4, 7\}$$

$$H_2(Y) \stackrel{\text{def}}{=} \{2, 5, 9\} - Y$$

$$H_3(Y) \stackrel{\text{def}}{=} \{1, 2, 3, 4, 5\} \cap (\{2, 4, 8\} \cup Y)$$

for all $Y \subseteq \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

- * (a) Which of these functions are monotone; which ones aren't? Justify your answer in each case.
- * (b) Compute the least and greatest fixed points of H_3 using the iterations H_3^i with $i = 1, 2, \dots$ and Theorem 3.24.
- (c) Does H_2 have any fixed points?
- (d) Recall $G: \mathcal{P}(\{s_0, s_1\}) \rightarrow \mathcal{P}(\{s_0, s_1\})$ with

$$G(Y) \stackrel{\text{def}}{=} \text{if } Y = \{s_0\} \text{ then } \{s_1\} \text{ else } \{s_0\}.$$

Use mathematical induction to show that G^i equals G for all odd numbers $i \geq 1$. What does G^i look like for even numbers i ?

- * 2. Let A and B be two subsets of S and let $F: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ be a monotone function. Show that
 - (a) $F_1: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ with $F_1(Y) \stackrel{\text{def}}{=} A \cap F(Y)$ is monotone
 - (b) $F_2: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ with $F_2(Y) \stackrel{\text{def}}{=} A \cup (B \cap F(Y))$ is monotone.
- 3. Use Theorems 3.25 and 3.26 to compute the following sets (the underlying model is in Figure 3.42):
 - (a) $\llbracket \text{EF } p \rrbracket$
 - (b) $\llbracket \text{EG } q \rrbracket$.

4. Using the function $F(X) = \llbracket \phi \rrbracket \cup \text{pre}_V(X)$ prove that $\llbracket \text{AF } \phi \rrbracket$ is the least fixed point of F . Hence argue that the procedure SAT_{AF} is correct and terminates.
- * 5. One may also compute $\text{AG } \phi$ directly as a fixed point. Consider the function $H: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ with $H(X) = \llbracket \phi \rrbracket \cap \text{pre}_V(X)$. Show that H is monotone and that $\llbracket \text{AG } \phi \rrbracket$ is the greatest fixed point of H . Use that insight to write a procedure SAT_{AG} .
6. Similarly, one may compute $\text{A}[\phi_1 \text{ U } \phi_2]$ directly as a fixed point, using $K: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, where $K(X) = \llbracket \phi_2 \rrbracket \cup (\llbracket \phi_1 \rrbracket \cap \text{pre}_V(X))$. Show that K is monotone and that $\llbracket \text{A}[\phi_1 \text{ U } \phi_2] \rrbracket$ is the least fixed point of K . Use that insight to write a procedure SAT_{AU} . Can you use that routine to handle all calls of the form $\text{AF } \phi$ as well?
7. Prove that $\llbracket \text{A}[\phi_1 \text{ U } \phi_2] \rrbracket = \llbracket \phi_2 \vee (\phi_1 \wedge \text{AX}(\text{A}[\phi_1 \text{ U } \phi_2])) \rrbracket$.
8. Prove that $\llbracket \text{AG } \phi \rrbracket = \llbracket \phi \wedge \text{AX}(\text{AG } \phi) \rrbracket$.
9. Show that the repeat-statements in the code for SAT_{EU} and SAT_{EG} always terminate. Use this fact to reason informally that the main program SAT terminates for all valid CTL formulas ϕ . Note that some subclauses, like the one for AU , call SAT recursively and with a more complex formula. Why does this not affect termination?

3.9 Bibliographic notes

Temporal logic was invented by the philosopher A. Prior in the 1960s; his logic was similar to what we now call LTL. The first use of temporal logic for reasoning about concurrent programs was by A. Pnueli [Pnu81]. The logic CTL was invented by E. Clarke and E. A. Emerson (during the early 1980s); and CTL* was invented by E. A. Emerson and J. Halpern (in 1986) to unify CTL and LTL.

CTL model checking was invented by E. Clarke and E. A. Emerson [CE81] and by J. Quielle and J. Sifakis [QS81]. The technique we described for LTL model checking was invented by M. Vardi and P. Wolper [VW84]. Surveys of some of these ideas can be found in [CGL93] and [CGP99]. The theorem about adequate sets of CTL connectives is proved in [Mar01].

The original SMV system was written by K. McMillan [McM93] and is available with source code from Carnegie Mellon University¹. NuSMV² is a reimplementaion, developed in Trento by A. Cimatti, and M. Roveri and is aimed at being customisable and extensible. Extensive documentation

¹ www.cs.cmu.edu/~modelcheck/

² nusmv.iirst.itc.it

about NuSMV can be found at that site. NuSMV supports essentially the same system description language as CMU SMV, but it has an improved user interface and a greater variety of algorithms. For example, whereas CMU SMV checks only CTL specification, NuSMV supports LTL and CTL. NuSMV implements bounded model checking [BCCZ99]. Cadence SMV³ is an entirely new model checker focused on compositional systems and abstraction as ways of addressing the state explosion problem. It was also developed by K. McMillan and its description language resembles but much extends the original SMV.

A WWW site which gathers frequently used specification patterns in various frameworks (such as CTL, LTL and regular expressions) is maintained by M. Dwyer, G. Avrunin, J. Corbett and L. Dillon¹.

Current research in model checking includes attempts to exploit abstractions, symmetries and compositionality [CGL94, Lon83, Dam96] in order to reduce the impact of the state explosion problem.

The model checker Spin, which is geared towards asynchronous systems and is based on the temporal logic LTL, can be found at the Spin website². A model checker called FDR2 based on the process algebra CSP is available³. The Edinburgh Concurrency Workbench⁴ and the Concurrency Workbench of North Carolina⁵ are similar software tools for the design and analysis of concurrent systems. An example of a customisable and extensible modular model checking frameworks for the verification of concurrent software is Bogor⁶.

There are many textbooks about verification of reactive systems; we mention [MP91, MP95, Ros97, Hol90]. The SMV code contained in this chapter can be downloaded from www.cs.bham.ac.uk/research/lics/.

³ www-cad.eecs.berkeley.edu/~kenmcmil/

¹ www.cis.ksu.edu/~dwyer/spec-patterns.html

² netlib.bell-labs.com/netlib/spin/whatispin.html

³ www.formal.demon.co.uk/FDR2.html

⁴ www.dcs.ed.ac.uk/home/cwb

⁵ www.csc.ncsu.edu/eos/users/r/rance/WWW/cwb-nc.html

⁶ <http://bogor.projects.cis.ksu.edu/>