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

$$\neg \mathrm{G}\,\phi \equiv \mathrm{F}\,\neg\phi \qquad \neg \mathrm{F}\,\phi \equiv \mathrm{G}\,\neg\phi \qquad \neg \mathrm{X}\,\phi \equiv \mathrm{X}\,\neg\phi.$$

Also U and R are duals of each other:

$$\neg(\phi \mathrel{\mathrm{U}} \psi) \equiv \neg\phi \mathrel{\mathrm{R}} \neg\psi \qquad \neg(\phi \mathrel{\mathrm{R}} \psi) \equiv \neg\phi \mathrel{\mathrm{U}} \neg\psi.$$

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.,

$$\mathrm{F}\,(\phi \vee \psi) \equiv \mathrm{F}\,\phi \vee \mathrm{F}\,\psi$$
$$\mathrm{G}\,(\phi \wedge \psi) \equiv \mathrm{G}\,\phi \wedge \mathrm{G}\,\psi.$$

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 $\mathrm{F}\,(\phi \wedge \psi)$ and $\mathrm{F}\,\phi \wedge \mathrm{F}\,\psi$, for some ϕ, ψ. Take the path $s_0 \to s_1 \to s_0 \to s_1 \to \dots$ from the system of Figure 3.3, for example; it satisfies $\mathrm{F}\,p \wedge \mathrm{F}\,r$ but it doesn't satisfy $\mathrm{F}\,(p \wedge r)$.

Here are two more equivalences in LTL:

$$\mathrm{F}\,\phi \equiv \top \mathrel{\mathrm{U}} \phi \qquad \mathrm{G}\,\phi \equiv \bot \mathrel{\mathrm{R}} \phi.$$

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, $\bot$ is 'every constraint.' If you ask me to bring it about that $\bot$ holds, I'll have to meet every constraint there is, which is impossible.)

The second formula, that $\mathrm{G}\,\phi \equiv \bot \mathrel{\mathrm{R}} \phi$, 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:' $\bot$ releases ϕ, but $\bot$ 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:

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

To prove equivalence (3.2), suppose first that a path satisfies $\phi \mathrm{~U~} \psi$. Then, from clause 11, we have $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi^j \vDash \phi$. From clause 12, this proves $\phi \mathrm{~W~} \psi$, and from clause 10 it proves $\mathrm{F}\,\psi$. Thus for all paths π, if $\pi \vDash \phi \mathrm{~U~} \psi$ then $\pi \vDash \phi \mathrm{~W~} \psi \wedge \mathrm{F}\,\psi$. 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:

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

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:

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

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

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 $\{\bot, \wedge, \neg\}$ forms an adequate set of connectives, since the other connectives $\vee$, $\rightarrow$, $\top$, etc., can be written in terms of those three.

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 $\{\mathrm{U}, \mathrm{X}\}$, $\{\mathrm{R}, \mathrm{X}\}$, $\{\mathrm{W}, \mathrm{X}\}$ is adequate. To see this, we note that
  - R and W may be defined from U, by the duality $\phi \mathrm{~R~} \psi \equiv \neg(\neg\phi \mathrm{~U~} \neg\psi)$ and equivalence (3.4) followed by the duality, respectively.
  - U and W may be defined from R, by the duality $\phi \mathrm{~U~} \psi \equiv \neg(\neg\phi \mathrm{~R~} \neg\psi)$ and equivalence (3.4), respectively.
  - R and U may be defined from W, by equivalence (3.5) and the duality $\phi \mathrm{~U~} \psi \equiv \neg(\neg\phi \mathrm{~R~} \neg\psi)$ 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 propositional 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: $\{\mathrm{U},\mathrm{R}\}$, $\{\mathrm{U},\mathrm{W}\}$, $\{\mathrm{U},\mathrm{G}\}$, $\{\mathrm{R},\mathrm{F}\}$, $\{\mathrm{W},\mathrm{F}\}$. But $\{\mathrm{R},\mathrm{G}\}$ and $\{\mathrm{W},\mathrm{G}\}$ 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 $\phi \,\mathrm{U}\, \psi \;\equiv\; \neg(\neg\psi \,\mathrm{U}\, (\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi$ holds for all LTL formulas ϕ and ψ.

PROOF: Take any path $s_0 \to s_1 \to s_2 \to \dots$ in any model.

First, suppose $s_0 \vDash \phi \,\mathrm{U}\, \psi$ holds. Let n be the smallest number such that $s_n \vDash \psi$; such a number has to exist since $s_0 \vDash \phi \,\mathrm{U}\, \psi$; then, for each $k < n$, $s_k \vDash \phi$. We immediately have $s_0 \vDash \mathrm{F}\,\psi$, so it remains to show $s_0 \vDash \neg(\neg\psi \,\mathrm{U}\, (\neg\phi \wedge \neg\psi))$, which, if we expand, means:
$(*)$ for each $i > 0$, if $s_i \vDash \neg\phi \wedge \neg\psi$, then there is some $j < i$ with $s_j \vDash \psi$.
Take any $i > 0$ with $s_i \vDash \neg\phi \wedge \neg\psi$; $i > n$, so we can take $j \stackrel{\mathrm{def}}{=} n$ and have $s_j \vDash \psi$.
Conversely, suppose $s_0 \vDash \neg(\neg\psi \,\mathrm{U}\, (\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi$ holds; we prove $s_0 \vDash \phi \,\mathrm{U}\, \psi$. Since $s_0 \vDash \mathrm{F}\,\psi$, we have a minimal n as before. We show that, for any $i < n$, $s_i \vDash \phi$. Suppose $s_i \vDash \neg\phi$; since n is minimal, we know $s_i \vDash \neg\psi$, so by $(*)$ there is some $j < i < n$ with $s_j \vDash \psi$, contradicting the minimality of n. □

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.

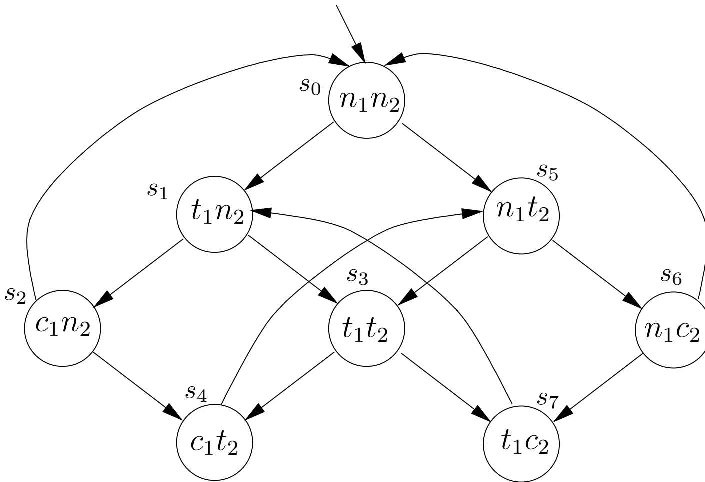


Figure 3.7. A first-attempt model for mutual exclusion.

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.

The first modelling attempt We will model two processes, each of which is in its non-critical state (n), or trying to enter its critical state (t), or in its critical state (c). Each individual process undergoes transitions in the cycle $n \to t \to c \to n \to \dots$, 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 $p_1p_2 \dots p_m$ 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 $\mathrm{G}\,\neg(c_1 \wedge c_2)$. Clearly, $\mathrm{G}\,\neg(c_1 \wedge c_2)$ is satisfied in the initial state (indeed, in every state).

Liveness: This is also expressible: $\mathrm{G}\,(t_1 \rightarrow \mathrm{F}\,c_1)$. However, it is *not* satisfied by the initial state, for we can find a 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 215).

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: $\mathrm{G}\,(c_1 \rightarrow c_1\ \mathrm{W}\ (\neg c_1 \wedge \neg c_1\ \mathrm{W}\ c_2))$. 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.

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.

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 $t_1 t_2$; 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).

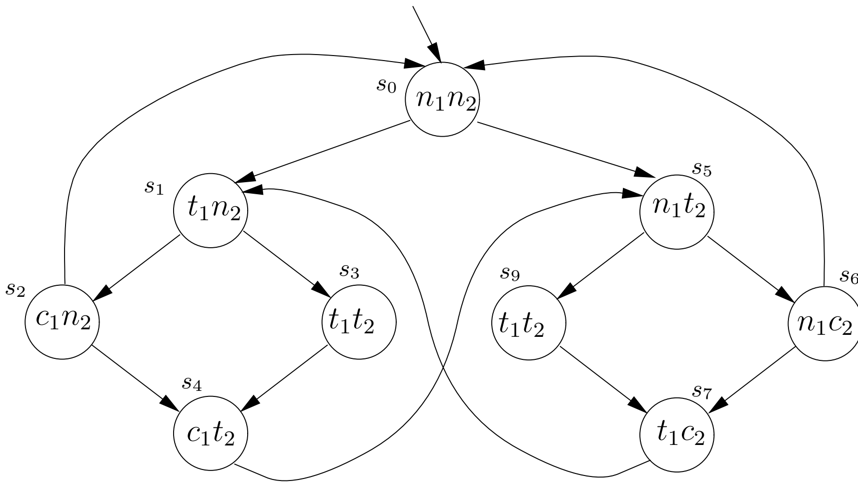


Figure 3.8. A second-attempt model for mutual exclusion. There are now two states representing t_1t_2, namely s_3 and s_9.

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 `init`ial 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};
                  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

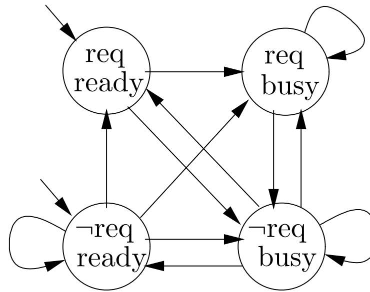


Figure 3.9. The model corresponding to the SMV program in the text.

easily verify that the specification of our module `main` holds of the model in Figure 3.9.

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.

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 254. 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).

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

```
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)
```

Figure 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.

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 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

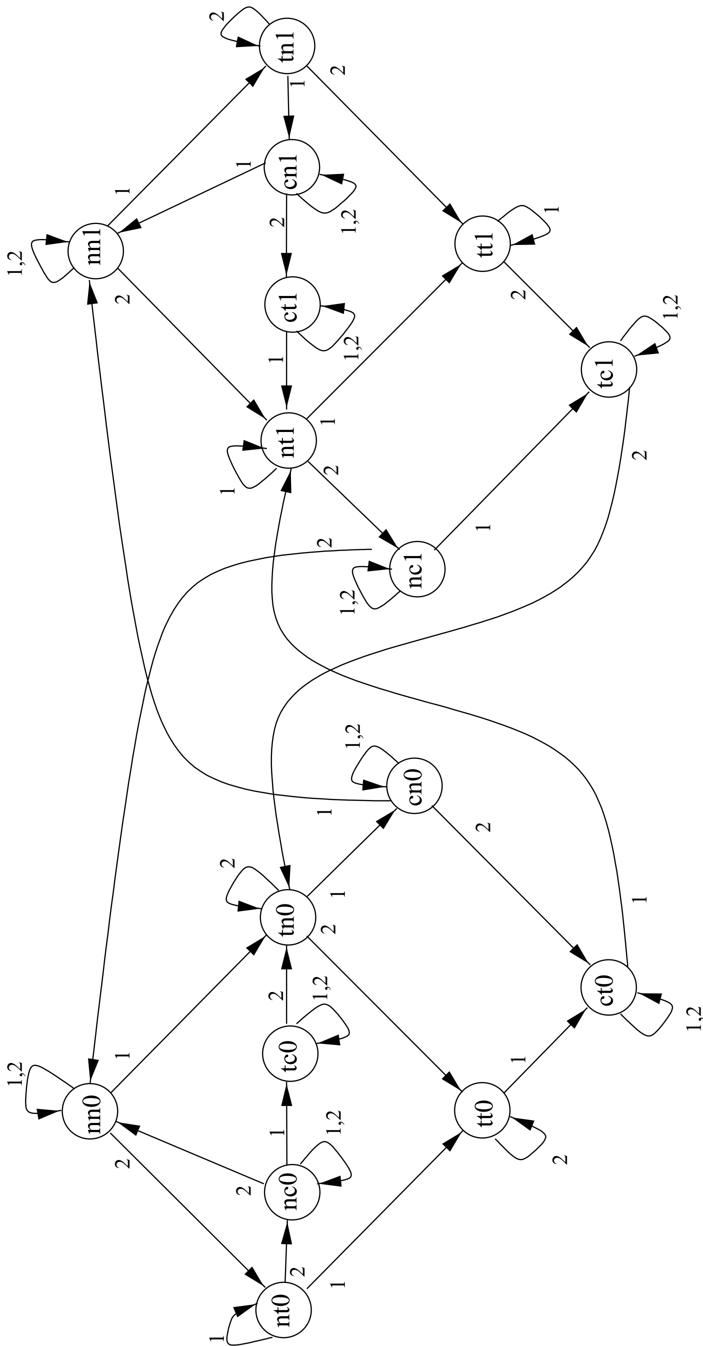


Figure 3.11. The transition system corresponding to the SMV code in Figure 3.10. The labels on the transitions denote the process which makes the move. The label $1, 2$ means that either process could make that move.

(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 ferryman

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 behavioural 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 behaviour (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,

```
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))
```

Figure 3.12. NuSMV code for the ferryman planning problem.

etc., and always includes 0. 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` $\neq$ `cabbage` then that set is $\{\mathtt{g}, \mathtt{w}, \mathtt{0}\}$. 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.

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.)

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 $\phi \text{ U } \psi$.

```
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
```

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

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 -bmc_length 7 ferryman.smv` shows that the LTL formula holds in that model, meaning that no solution with fewer than seven transitions 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 $\phi \, \mathrm{U} \, \psi$, where ϕ equals $(\texttt{goat} = \texttt{cabbage} \lor \texttt{goat} = \texttt{wolf}) \to \texttt{goat} = \texttt{ferryman}$ and ψ equals $\texttt{cabbage} \land \texttt{goat} \land \texttt{wolf} \land \texttt{ferryman}$, we now seek a path satisfying $(\phi \, \mathrm{U} \, \psi) \land \mathrm{G} \, (\texttt{goat} \to \mathrm{G} \, \texttt{goat})$. 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 alternating bit protocol (ABP) 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

```
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
```

Figure 3.14. The ABP sender in SMV.

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 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 assignment to `forget`. The value of

```
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
```

Figure 3.15. The ABP receiver in SMV.

`input` should be transmitted to `output`, unless `forget` is true. The two-bit channel `two-bit-chan`, used to send messages, is similar. Again, 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).

The channels have fairness constraint which are intended to model the fact that, although channels can lose messages, we assume that they infinitely often transmit the message correctly. (If this were not the case, then we could find an uninteresting violation of the liveness property, for example a path along which all messages from a certain time onwards get lost.)

It is interesting to note that the fairness constraint 'infinitely often `!forget`' is not sufficient to prove the desired properties, for although it forces the channel to transmit infinitely often, it doesn't prevent it from (say) dropping all the 0 bits and transmitting all the 1 bits. That is why we use the stronger fairness constraints shown. Some systems allow fairness

```
MODULE one-bit-chan(input)
VAR
   output : boolean;
   forget : boolean;
ASSIGN
   next(output) := case
                      forget : output;
                      1:       input;
                    esac;
FAIRNESS running
FAIRNESS input & !forget
FAIRNESS !input & !forget

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 input1 & !forget
FAIRNESS !input1 & !forget
FAIRNESS input2 & !forget
FAIRNESS !input2 & !forget
```

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

contraints of the form 'infinitely often p implies infinitely often q', which would be more satisfactory here, but is not allowed by SMV.

Finally, we tie it all together with the module `main` (Figure 3.17). Its role is to connect together the components of the system, and giving them initial values of their parameters. 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

```
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)
```

Figure 3.17. The `main` ABP module.

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 indeed received by the receiver: `G (S.st=sent & S.message1=1 -> msg_chan.output1=1)`.

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 `G F st=sent`. (This specification could equivalently have been written in the main module, as `G F S.st=sent`.) Similarly, acknowledgements get through eventually. In the module `receiver`, we write `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 $\mathrm{EF}\, q$.
- From all reachable states satisfying p, it is possible to maintain p continuously until reaching a state satisfying q: this is written $\mathrm{AG}\,(p \to \mathrm{E}[p\ \mathrm{U}\ q])$.
- Whenever a state satisfying p is reached, the system can exhibit q continuously forevermore: $\mathrm{AG}\,(p \to \mathrm{EG}\, q)$.
- There is a reachable state from which all reachable states satisfy p: $\mathrm{EF}\,\mathrm{AG}\, p$.

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:

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

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 $\mathrm{E}[\phi_1\ \mathrm{U}\ \phi_2]$, 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:

- $\mathrm{AG}\,(q \rightarrow \mathrm{EG}\,r)$, note that this is not the same as $\mathrm{AG}\,q \rightarrow \mathrm{EG}\,r$, for according to Convention 3.13, the latter formula means $(\mathrm{AG}\,q) \rightarrow (\mathrm{EG}\,r)$
- $\mathrm{EF}\,\mathrm{E}[r\ \mathrm{U}\ q]$
- $\mathrm{A}[p\ \mathrm{U}\ \mathrm{EF}\,r]$
- $\mathrm{EF}\,\mathrm{EG}\,p \rightarrow \mathrm{AF}\,r$, again, note that this binds as $(\mathrm{EF}\,\mathrm{EG}\,p) \rightarrow \mathrm{AF}\,r$, not $\mathrm{EF}\,(\mathrm{EG}\,p \rightarrow \mathrm{AF}\,r)$ or $\mathrm{EF}\,\mathrm{EG}\,(p \rightarrow \mathrm{AF}\,r)$
- $\mathrm{A}[p_1\ \mathrm{U}\ \mathrm{A}[p_2\ \mathrm{U}\ p_3]]$
- $\mathrm{E}[\mathrm{A}[p_1\ \mathrm{U}\ p_2]\ \mathrm{U}\ p_3]$
- $\mathrm{AG}\,(p \rightarrow \mathrm{A}[p\ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p\ \mathrm{U}\ q])])$.

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:

- $\mathrm{EF}\,\mathrm{G}\,r$
- $\mathrm{A}\neg\mathrm{G}\,\neg p$
- $\mathrm{F}\,[r\ \mathrm{U}\ q]$
- $\mathrm{EF}\,(r\ \mathrm{U}\ q)$
- $\mathrm{AEF}\,r$
- $\mathrm{A}[(r\ \mathrm{U}\ q) \wedge (p\ \mathrm{U}\ r)]$.

It is especially worth understanding why the syntax rules don't allow us to construct these. For example, take $\mathrm{EF}\,(r\ \mathrm{U}\ q)$. 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 $\mathrm{EF}\,\mathrm{E}[r\ \mathrm{U}\ q]$ or $\mathrm{EF}\,\mathrm{A}[r\ \mathrm{U}\ q]$.

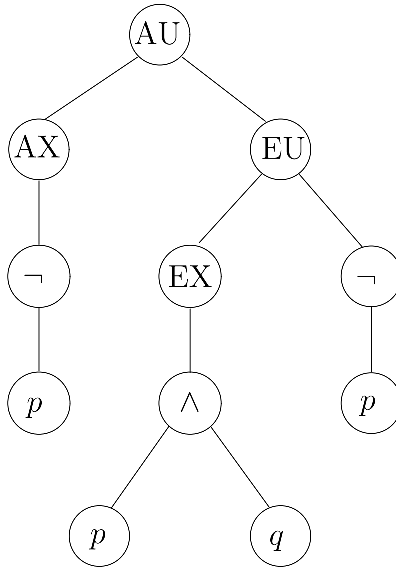


Figure 3.18. The parse tree of a CTL formula without infix notation.

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

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

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

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 $\mathcal{M}, s \vDash \phi$ 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 $\mathcal{M}, s \vDash \phi$ 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 $\mathcal{M}, s \vDash \phi$ is defined by structural induction on ϕ:

1. $\mathcal{M}, s \vDash \top$ and $\mathcal{M}, s \nvDash \bot$
2. $\mathcal{M}, s \vDash p$ iff $p \in L(s)$
3. $\mathcal{M}, s \vDash \neg\phi$ iff $\mathcal{M}, s \nvDash \phi$
4. $\mathcal{M}, s \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ and $\mathcal{M}, s \vDash \phi_2$
5. $\mathcal{M}, s \vDash \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$
6. $\mathcal{M}, s \vDash \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \nvDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
7. $\mathcal{M}, s \vDash \mathrm{AX}\,\phi$ iff for all s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. Thus, AX says: 'in every next state.'
8. $\mathcal{M}, s \vDash \mathrm{EX}\,\phi$ iff for some s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. 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. $\mathcal{M}, s \vDash \mathrm{AG}\,\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$. 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. $\mathcal{M}, s \vDash \mathrm{EG}\,\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and for all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$. Mnemonically: there Exists a path beginning in s such that ϕ holds Globally along the path.

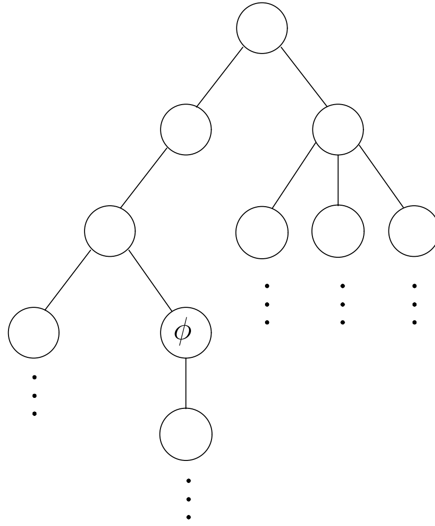


Figure 3.19. A system whose starting state satisfies EF ϕ.

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

Clauses 9–14 above refer to computation paths in models. It is therefore useful to visualise all possible computation paths from a given state s by unwinding the transition system to obtain an infinite computation tree, whence 'computation tree logic.' The diagrams in Figures 3.19–3.22 show schematically systems whose starting states satisfy the formulas EF ϕ, EG ϕ, AG ϕ and AF ϕ, 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.

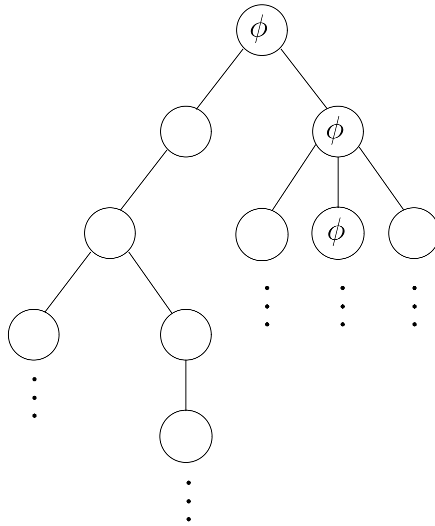


Figure 3.20. A system whose starting state satisfies EG ϕ.

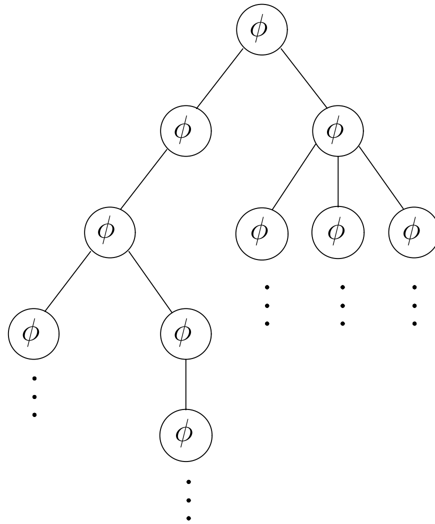


Figure 3.21. A system whose starting state satisfies AG ϕ.

Recall the transition system of Figure 3.3 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. $\mathcal{M}, s_0 \vDash p \wedge q$ holds since the atomic symbols p and q are contained in the node of s_0.
2. $\mathcal{M}, s_0 \vDash \neg r$ holds since the atomic symbol r is *not* contained in node s_0.

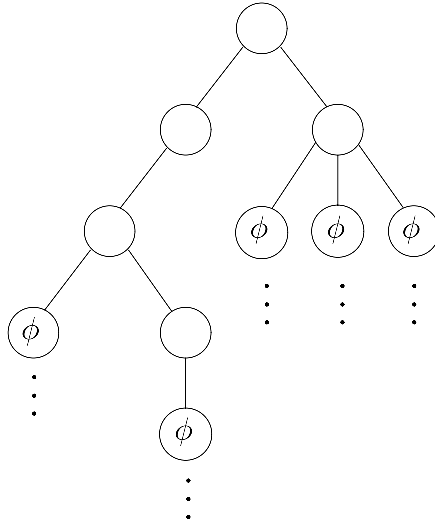


Figure 3.22. A system whose starting state satisfies AF ϕ.

3. $\mathcal{M}, s_0 \vDash \top$ holds by definition.
4. $\mathcal{M}, s_0 \vDash \text{EX}\ (q \wedge r)$ 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. $\mathcal{M}, s_0 \vDash \neg\text{AX}\ (q \wedge r)$ 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.
6. $\mathcal{M}, s_0 \vDash \neg\text{EF}\,(p \wedge r)$ 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. $\mathcal{M}, s_2 \vDash \text{EG}\, r$ 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 that path; this is the only computation path beginning at s_2 and so $\mathcal{M}, s_2 \vDash \text{AG}\, r$ holds as well.
8. $\mathcal{M}, s_0 \vDash \text{AF}\, r$ 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. $\mathcal{M}, s_0 \vDash \text{E}[(p \wedge q)\ \text{U}\ r]$ 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. $\mathcal{M}, s_0 \vDash \text{A}[p\ \text{U}\ r]$ 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. $\mathcal{M}, s_0 \vDash \text{AG}\ (p \vee q \vee r \rightarrow \text{EF EG}\ r)$ 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 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: $\mathrm{EF}\,(\mathsf{started} \wedge \neg\mathsf{ready})$. To express impossibility, we simply negate the formula.
- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
  $\mathrm{AG}\,(\mathsf{requested} \to \mathrm{AF}\ \mathsf{acknowledged})$.
- 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 $\mathrm{AG}\,\mathrm{AF}\ \mathsf{enabled} \to \mathrm{AG}\,\mathrm{AF}\ \mathsf{running}$, 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:
  $\mathrm{AG}\,(\mathrm{AF}\ \mathsf{enabled})$.
- Whatever happens, a certain process will eventually be permanently deadlocked:
  $\mathrm{AF}\,(\mathrm{AG}\ \mathsf{deadlock})$.
- From any state it is possible to get to a restart state:
  $\mathrm{AG}\,(\mathrm{EF}\ \mathsf{restart})$.
- An upwards travelling lift at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  $\mathrm{AG}\,(\mathsf{floor2} \wedge \mathsf{directionup} \wedge \mathsf{ButtonPressed5} \to \mathrm{A}[\mathsf{directionup}\ \mathrm{U}\ \mathsf{floor5}])$
  Here, our atomic descriptions are boolean expressions built from system variables, e.g., floor2.
- The lift can remain idle on the third floor with its doors closed:
  $\mathrm{AG}\,(\mathsf{floor3} \wedge \mathsf{idle} \wedge \mathsf{doorclosed} \to \mathrm{EG}\,(\mathsf{floor3} \wedge \mathsf{idle} \wedge \mathsf{doorclosed}))$.
- 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 $\mathrm{AG}\,(n_1 \to \mathrm{EX}\,t_1)$ 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: $\mathrm{EF}\,(c_1 \wedge \mathrm{E}[c_1\ \mathrm{U}\ (\neg c_1 \wedge \mathrm{E}[\neg c_2\ \mathrm{U}\ c_1])])$.

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 universal and existential quantifiers, ranging over the states along a particular path. In view of these facts, it is not surprising to find that de Morgan rules exist:

$$\begin{aligned} \neg\mathrm{AF}\,\phi &\equiv \mathrm{EG}\,\neg\phi \\ \neg\mathrm{EF}\,\phi &\equiv \mathrm{AG}\,\neg\phi \\ \neg\mathrm{AX}\,\phi &\equiv \mathrm{EX}\,\neg\phi. \end{aligned} \tag{3.6}$$

We also have the equivalences

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

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 $\neg\mathrm{EX}\,\neg$; and AG, AF, EG and EF can be written in terms of AU and EU as follows: first, write $\mathrm{AG}\,\phi$ as $\neg\mathrm{EF}\,\neg\phi$ and $\mathrm{EG}\,\phi$ as $\neg\mathrm{AF}\,\neg\phi$, using (3.6), and then use $\mathrm{AF}\,\phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi]$ and $\mathrm{EF}\,\phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi]$. Therefore AU, EU and EX form an adequate set of temporal connectives.

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

$$\mathrm{A}[\phi\ \mathrm{U}\ \psi] \;\equiv\; \neg(\mathrm{E}[\neg\psi\ \mathrm{U}\ (\neg\phi \wedge \neg\psi)] \vee \mathrm{EG}\,\neg\psi) \tag{3.7}$$

which can be proved as follows:

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

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 $\{\mathrm{AX}, \mathrm{EX}\}$, at least one of $\{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$ 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:

- $\mathrm{A}[\phi \mathrm{~R~} \psi] = \neg\mathrm{E}[\neg\phi \mathrm{~U~} \neg\psi]$
- $\mathrm{E}[\phi \mathrm{~R~} \psi] = \neg\mathrm{A}[\neg\phi \mathrm{~U~} \neg\psi]$
- $\mathrm{A}[\phi \mathrm{~W~} \psi] = \mathrm{A}[\psi \mathrm{~R~} (\phi \vee \psi)]$, and then use the first equation above
- $\mathrm{E}[\phi \mathrm{~W~} \psi] = \mathrm{E}[\psi \mathrm{~R~} (\phi \vee \psi)]$, 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:

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

For example, the intuition for the third one is the following: in order to have $\mathrm{AF}\,\phi$ in a particular state, ϕ must be true at some point along each path from that state. To achieve this, we either have ϕ true now, in the current state; or we postpone it, in which case we must have $\mathrm{AF}\,\phi$ in each of the next states. Notice how this equivalence appears to define AF in terms of AX and AF itself, an apparently circular definition. In fact, these equivalences can be used to define the six connectives on the left in terms of AX and EX, in a *non-circular* way. This is called the fixed-point characterisation of CTL; it is the mathematical foundation for the model-checking algorithm developed in Section 3.6.1; 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 $\mathrm{F}\,p \to \mathrm{F}\,q$. It is not possible to write this in CTL because of the constraint that every F has an associated A or E. The formula $\mathrm{AF}\,p \to \mathrm{AF}\,q$ 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 $(p \to \mathrm{AF}\ q)$, 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 $\mathrm{F}\,p \to \mathrm{F}\,q$.

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

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

These formulas are *not* equivalent to, respectively, $\mathrm{A}[(p \vee q) \mathrm{~U~} r)]$, $\mathrm{AX}\,p \vee \mathrm{AX\,AX}\,p$ and $\mathrm{EG\,EF}\,p$. 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:

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

  where p is any atomic formula and α any path formula; and
- *path formulas*, which are evaluated along paths:

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \mathrm{~U~} \alpha) \mid (\mathrm{G}\,\alpha) \mid (\mathrm{F}\,\alpha) \mid (\mathrm{X}\,\alpha)$$

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 $\top$.

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 $\mathrm{A}[\alpha]$. 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

$$\alpha ::= (\phi \mathrm{~U~} \phi) \mid (\mathrm{G}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{X}\,\phi)$$

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

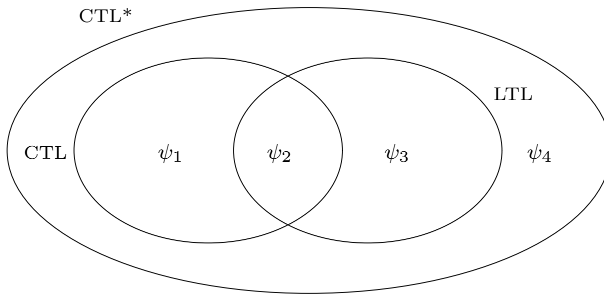
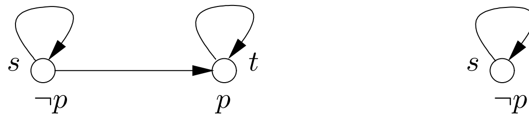


Figure 3.23. The expressive powers of CTL, LTL and CTL*.

shown:

In CTL but not in LTL: $\psi_1 \stackrel{\text{def}}{=} \mathrm{AG\,EF}\,p$. 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 $\mathrm{AG\,EF}\,p$ is not expressible in LTL is as follows. Let ϕ be an LTL formula such that $\mathrm{A}[\phi]$ is allegedly equivalent to $\mathrm{AG\,EF}\,p$. Since $\mathcal{M}, s \vDash \mathrm{AG\,EF}\,p$ in the left-hand diagram below, we have $\mathcal{M}, s \vDash \mathrm{A}[\phi]$. 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 $\mathcal{M}', s \vDash \mathrm{A}[\phi]$. Yet, it is *not* the case that $\mathcal{M}', s \vDash \mathrm{AG\,EF}\,p$; a contradiction.



In CTL*, but neither in CTL nor in LTL: $\psi_4 \stackrel{\text{def}}{=} \mathrm{E}[\mathrm{G\,F}\,p]$, 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?)

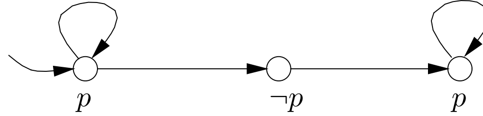
In LTL but not in CTL: $\psi_3 \stackrel{\text{def}}{=} \mathrm{A}[\mathrm{G\,F}\,p \to \mathrm{F}\,q]$, saying that if there are infinitely many p along the path, then there is an occurrence of q. This is an interesting thing to be able to say; for example, many fairness constraints are of the form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

In LTL and CTL: $\psi_2 \stackrel{\text{def}}{=} \mathrm{AG}\,(p \to \mathrm{AF}\,q)$ in CTL, or $\mathrm{G}\,(p \to \mathrm{F}\,q)$ 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 $\mathrm{G}\,(p \to \mathrm{F}\,q)$ is equivalent to the CTL formula AG $(p \to \mathrm{AF}\ q)$. We discuss two more negative examples:

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



  In fact, AF AG p is strictly stronger than $\mathrm{F}\,\mathrm{G}\,p$.
- While the LTL formulas $\mathrm{X}\,\mathrm{F}\,p$ and $\mathrm{F}\,\mathrm{X}\,p$ are equivalent, and they are equivalent to the CTL formula AX AF p, they are not equivalent to AF 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 $\mathrm{AF}\,\mathrm{AX}\,p$ 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 $\mathrm{E}[\mathrm{F}\,p \wedge \mathrm{F}\,q] \;\equiv\; \mathrm{EF}\,[p \wedge \mathrm{EF}\,q] \vee \mathrm{EF}\,[q \wedge \mathrm{EF}\,p]$ since, if we have $\mathrm{F}\,p \wedge \mathrm{F}\,q$ 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:

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

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

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 $\mathrm{G}\,(q \rightarrow O\, p)$.

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 $\neg p \mathrm{~W~} q$, or equivalently $\neg(\neg q \mathrm{~U~} (p \wedge \neg q))$ 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 $\mathcal{M}, s \vDash \phi$ 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

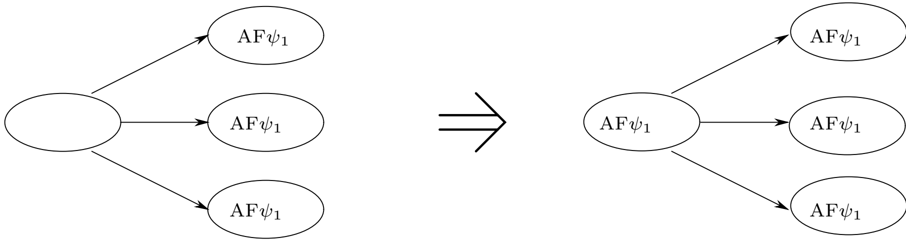
$$\mathcal{M}, s_0 \overset{?}{\vDash} \phi$$

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' ($\mathcal{M}, s_0 \vDash \phi$ holds), or 'no' ($\mathcal{M}, s_0 \vDash \phi$ 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.

The labelling algorithm We present an algorithm which, given a model and a CTL formula, outputs the set of states of the model that satisfy the formula. The algorithm does not need to be able to handle every CTL connective explicitly, since we have already seen that the connectives $\bot$, $\neg$ and $\wedge$ form an adequate set as far as the propositional connectives are concerned; and AF , EU and EX form an adequate set of temporal connectives. Given an arbitrary CTL formula ϕ, we would simply pre-process ϕ in order to write it in an equivalent form in terms of the adequate set of connectives, and then

Repeat…



… until no change.

Figure 3.24. The iteration step of the procedure for labelling states with subformulas of the form $\mathrm{AF}\,\psi_1$.

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 TRANSLATE (ϕ), i.e., we write ϕ in terms of the connectives AF, EU, EX, $\wedge$, $\neg$ and $\bot$ using the equivalences given earlier in the chapter. Next, label the states of $\mathcal{M}$ 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

- $\bot$: then no states are labelled with $\bot$.
- p: then label s with p if $p \in L(s)$.
- $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled both with ψ_1 and with ψ_2.
- $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1.
- $\mathrm{AF}\,\psi_1$:
  - If any state s is labelled with ψ_1, label it with $\mathrm{AF}\,\psi_1$.
  - Repeat: label any state with $\mathrm{AF}\,\psi_1$ if all successor states are labelled with $\mathrm{AF}\,\psi_1$, until there is no change. This step is illustrated in Figure 3.24.
- $\mathrm{E}[\psi_1 \mathrm{~U~} \psi_2]$:
  - If any state s is labelled with ψ_2, label it with $\mathrm{E}[\psi_1 \mathrm{~U~} \psi_2]$.
  - Repeat: label any state with $\mathrm{E}[\psi_1 \mathrm{~U~} \psi_2]$ if it is labelled with ψ_1 and at least one of its successors is labelled with $\mathrm{E}[\psi_1 \mathrm{~U~} \psi_2]$, until there is no change. This step is illustrated in Figure 3.25.
- $\mathrm{EX}\,\psi_1$: label any state with $\mathrm{EX}\,\psi_1$ if one of its successors is labelled with ψ_1.

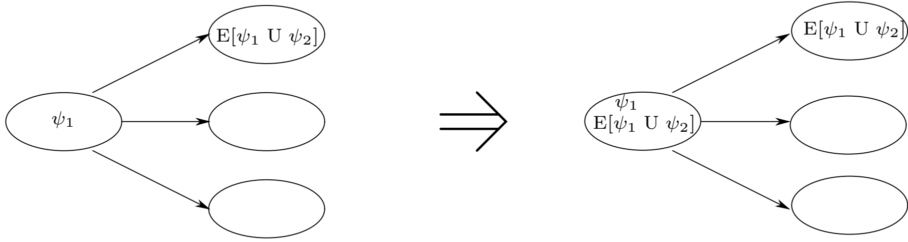


Figure 3.25. The iteration step of the procedure for labelling states with subformulas of the form $\mathrm{E}[\psi_1 \mathrm{~U~} \psi_2]$.

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

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

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 $\mathrm{EG}\,\psi_1$ *directly*:

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

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

A variant which is more efficient We can improve the efficiency of our labelling algorithm by using a cleverer way of handling EG. Instead of using EX, EU and AF as the adequate set, we use EX, EU and EG instead. For EX and EU we do as before (but take care to search the model by

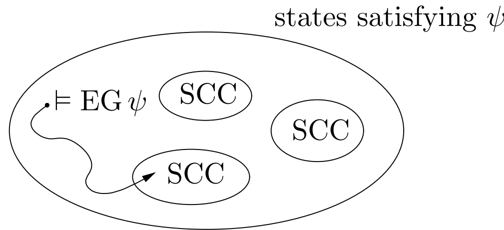


Figure 3.26. A better way of handling EG.

backwards breadth-first search, for this ensures that we won't have to pass over any node twice). For the $\mathrm{EG}\,\psi$ 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 $\mathrm{E}[\neg c_2 \mathrm{\ U\ } c_1]$; see Figure 3.27. The algorithm labels all states which satisfy c_1 during phase 1 with $\mathrm{E}[\neg c_2 \mathrm{\ U\ } c_1]$. This labels s_2 and s_4. During phase 2, it labels all states which do not satisfy c_2 and have a successor state that is already labelled. This labels states s_1 and s_3. During phase 3, we label s_0 because it does not satisfy c_2 and has a successor state (s_1) which is already labelled. Thereafter, the algorithm terminates because no additional states get labelled: all unlabelled states either satisfy c_2, or must pass through such a state to reach a labelled state.

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 $\mathrm{X}\,(\top \wedge \mathrm{EF}\, p_3)$, since this is not a CTL formula.

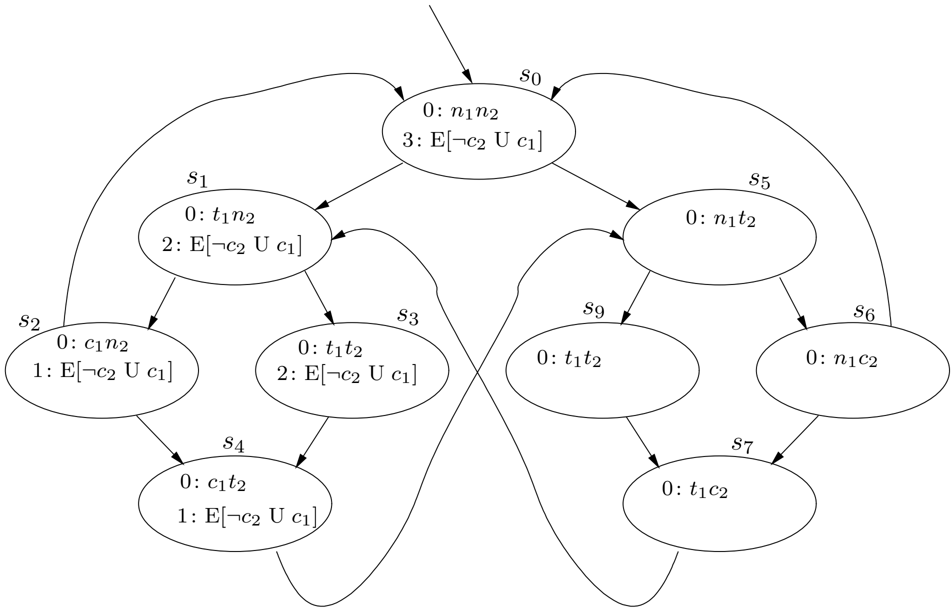


Figure 3.27. An example run of the labelling algorithm in our second model of mutual exclusion applied to the formula E[$\neg c_2$ U c_1].

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 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.

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

Figure 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 $\mathtt{SAT}_{\mathtt{EX}}$, $\mathtt{SAT}_{\mathtt{EU}}$ and $\mathtt{SAT}_{\mathtt{AF}}$, respectively, if EX, EU or AF is the root of the input's parse tree.

The algorithm is presented in Figure 3.28 and its subfunctions in Figures 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

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

'Pre' denotes travelling backwards along the transition relation. Both functions compute a pre-image of a set of states. The function $\mathrm{pre}_{\exists}$ (instrumental in $\mathtt{SAT}_{\mathtt{EX}}$ and $\mathtt{SAT}_{\mathtt{EU}}$) takes a subset Y of states and returns the set of states which *can* make a transition into Y. The function $\mathrm{pre}_{\forall}$, used in $\mathtt{SAT}_{\mathtt{AF}}$, takes

```
function SAT_EX (φ)
 /* determines the set of states satisfying EX φ */
local var X, Y
begin
    X := SAT (φ);
    Y := pre_∃(X);
    return Y
end
```

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

```
function SAT_AF (φ)
 /* determines the set of states satisfying AF φ */
local var X, Y
begin
    X := S;
    Y := SAT (φ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ pre_∀(Y)
    end
    return Y
end
```

Figure 3.30. The function $\mathtt{SAT}_{\mathtt{AF}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{AF}\,\phi$ in the manner described in the labelling algorithm.

a set Y and returns the set of states which make transitions *only* into Y. Observe that $\mathrm{pre}_{\forall}$ can be expressed in terms of complementation and $\mathrm{pre}_{\exists}$, as follows:

$$\mathrm{pre}_{\forall}(Y) = S - \mathrm{pre}_{\exists}(S - Y) \tag{3.8}$$

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.

```
function SAT_EU (φ, ψ)
 /* determines the set of states satisfying E[φ U ψ] */
local var W, X, Y
begin
    W := SAT (φ);
    X := S;
    Y := SAT (ψ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ (W ∩ pre_∃(Y))
    end
    return Y
end
```

Figure 3.31. The function $\mathtt{SAT}_{\mathtt{EU}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{E}[\phi \mathrm{~U~} \psi]$ in the manner described in the labelling algorithm.

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 $\mathcal{M}, s_0 \vDash \phi$ 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 $\mathrm{AG}\,(t_1 \rightarrow \mathrm{AF}\,c_1)$, 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 $\mathrm{FG}\neg c_2 \rightarrow \phi$, 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 $\mathrm{FG}\neg c_2 \rightarrow \phi$ 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 $C \stackrel{\text{def}}{=} \{\psi_1, \psi_2, \ldots, \psi_n\}$ be a set of n fairness constraints. A computation path $s_0 \to s_1 \to \ldots$ is fair with respect to these fairness constraints iff for each i there are infinitely many j such that $s_j \vDash \psi_i$, 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, $\mathcal{M}, s_0 \vDash \mathrm{A}_C\mathrm{G}\,\phi$ iff ϕ is true in every state along all fair paths; and similarly for $\mathrm{A}_C\mathrm{F}$, $\mathrm{A}_C\mathrm{U}$, etc. Notice that these operators explicitly depend on the chosen set C of fairness constraints. We already know that $\mathrm{E}_C\mathrm{U}$, $\mathrm{E}_C\mathrm{G}$ and $\mathrm{E}_C\mathrm{X}$ 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

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

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 $\mathrm{E}_C\mathrm{G}\,\phi$. 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.

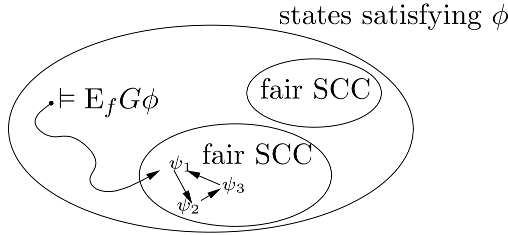


Figure 3.32. Computing the states satisfying $\mathrm{E}_C\mathrm{G}\,\phi$. A state satisfies $\mathrm{E}_C\mathrm{G}\,\phi$ iff, in the graph resulting from the restriction to states satisfying ϕ, the state has a fair path from it. A fair path is one which leads to an SCC with a cycle passing through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

- 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, 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 $\mathrm{G\,F}\,\phi \rightarrow \psi$. 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 $\mathrm{G\,F}\,\phi \rightarrow \psi$ will result in a formula with a different meaning from the intended one. For example, $\mathrm{AG\,AF}\,\phi \rightarrow \psi$ 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.

The basic strategy Let $\mathcal{M} = (S, \to, L)$ be a model, $s \in S$, and ϕ an LTL formula. We determine whether $\mathcal{M}, s \vDash \phi$, i.e., whether ϕ is satisfied along all paths of $\mathcal{M}$ starting at s. Almost all LTL model checking algorithms proceed along the following three steps.

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: $\pi \vDash \psi$ 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.
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.
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, $\mathcal{M}, s \vDash \phi$.' Otherwise, if there *is such a path*, output 'No, $\mathcal{M}, s \nvDash \phi$.' 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 $\neg(a \text{ U } b)$. Since it is not the case that all paths of $\mathcal{M}$ satisfy the formula (for example, the path $q_3, q_2, q_2 \ldots$ does not satisfy it) we expect the model check to fail.

In accordance with Step 1, we construct an automaton $A_{a\text{U}b}$ which characterises precisely the traces which satisfy $a \text{ U } b$. (We use the fact that $\neg\neg(a \text{ U } b)$ is equivalent to $a \text{ U } b$.) 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.

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;

```
init(a) := 1;
init(b) := 0;
next(a) := case
             !a : 0;
             b  : 1;
             1  : {0,1};
           esac;
next(b) := case
             a & next(a) : !b;
             !a : 1;
             1  : {0,1};
           esac;
```

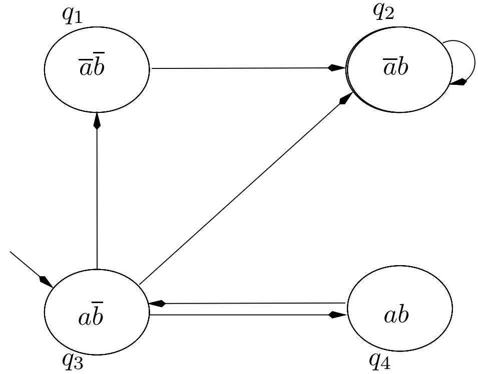


Figure 3.33. An SMV program and its model $\mathcal{M}$.

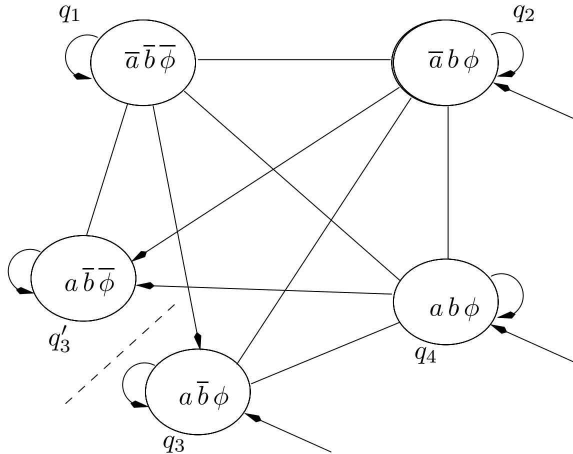


Figure 3.34. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} a \text{ U } b$. The transitions with no arrows can be taken in either direction. The acceptance condition is that the path of the automaton cannot loop indefinitely through q_3.

- the path respects a certain 'accepting condition.' For the automaton of Figure 3.34, the accepting condition is that the path should not end $q_3, q_3, q_3 \ldots$, indefinitely.

For example, suppose t is $a\,\overline{b}, a\,\overline{b}, a\,\overline{b}, a\,b, a\,b, \overline{a}\,\overline{b}, a\,\overline{b}, a\,\overline{b}, \ldots$, eventually repeating forevermore the state $a\,\overline{b}$. Then we choose the path $q_3, q_3, q_3, q_4, q_4, q_1, q_3', q_3' \ldots$. We start in q_3 because the first state is $a\,\overline{b}$ 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 $a\,\overline{b}$ 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 $q_3, q_3', \ldots$ doesn't meet the condition that we have to respect the transition relation.
- The path $q_3, q_3, q_3, q_4, q_4, q_1, q_3, q_3 \ldots$ 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 $\{\overline{a}\,\overline{b}, \overline{a}\,b, a\,\overline{b}, a\,b\}$, and in fact two states for the valuation $a\,\overline{b}$. One state for each of $\{\overline{a}\,\overline{b}, \overline{a}\,b, a\,b\}$ is intuitively enough, because those valuations determine whether $a \mathrm{~U~} b$ holds. But $a \mathrm{~U~} b$ could be false or true in $a\,\overline{b}$, so we have to consider the two cases. The presence of $\phi \stackrel{\text{def}}{=} a \mathrm{~U~} b$ in a state indicates that either we are still expecting ϕ to become true, or we have just obtained it. Whereas $\overline{\phi}$ 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 $a \mathrm{~U~} b$ to deliver b is eventually fulfilled.

Using this automaton $A_{a\mathrm{U}b}$, we proceed to Step 2. To combine the automaton $A_{a\mathrm{U}b}$ 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 which ever one we choose leads to the same successors. But it allows us to superimpose it on $A_{a\mathrm{U}b}$ 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: $q_3, (q_4, q_3,)^* q_2, q_2 \ldots,$ and $q_3, q_4, (q_3, q_4,)^* q_3', q_1, q_2, q_2, \ldots$ 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

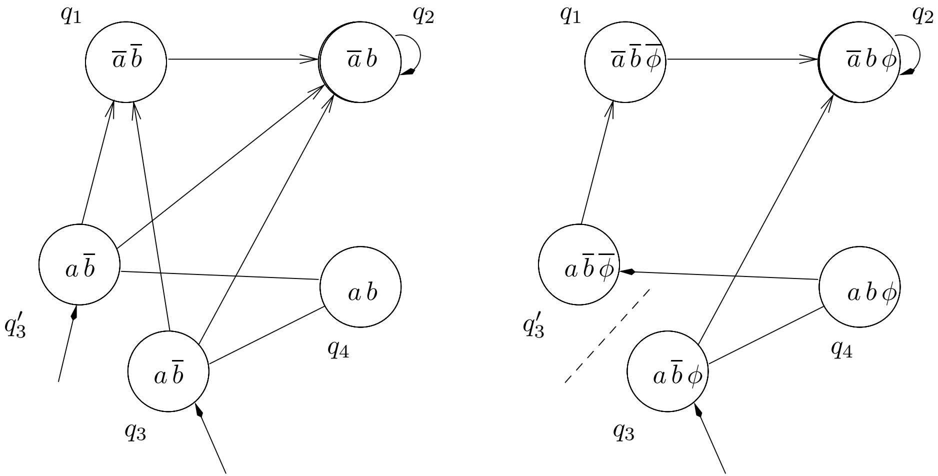


Figure 3.35. Left: the system $\mathcal{M}$ of Figure 3.33, redrawn with an expanded state space; right: the expanded $\mathcal{M}$ and $A_{a\mathrm{U}b}$ combined.

to Step 3, and as we expected, $\neg(a \mathrm{~U~} b)$ is not satisfied in all paths of the original system $\mathcal{M}$.

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 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, $\mathcal{C}(a \mathrm{~U~} b) = \{a, b, \neg a, \neg b, a \mathrm{~U~} b, \neg(a \mathrm{~U~} b)\}$. 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 $\psi_1 \mathrm{~U~} \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q$, 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 $\mathrm{X}\,\psi \in q$ then $\psi \in q'$;
- if $\neg\mathrm{X}\,\psi \in q$ then $\neg\psi \in q'$;
- If $\psi_1 \mathrm{~U~} \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \mathrm{~U~} \psi_2 \in q'$;
- If $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q'$.

These last two conditions are justified by the recursion laws

$$\psi_1 \mathrm{~U~} \psi_2 = \psi_2 \vee (\psi_1 \wedge \mathrm{X}\,(\psi_1 \mathrm{~U~} \psi_2))$$
$$\neg(\psi_1 \mathrm{~U~} \psi_2) = \neg\psi_2 \wedge (\neg\psi_1 \vee \mathrm{X}\,\neg(\psi_1 \mathrm{~U~} \psi_2))\ .$$

In particular, they ensure that whenever some state contains $\psi_1 \mathrm{~U~} \psi_2$, 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 $\psi_1 \mathrm{~U~} \psi_2$, namely that A_ϕ cannot stay for ever in states satisfying ψ_1 without ever obtaining ψ_2. Recall that, for the automaton of Figure 3.34 for $a \mathrm{~U~} b$, we stipulated the acceptance condition that the path through the automaton should not end $q_3, q_3, \ldots$.

The acceptance conditions of A_ϕ are defined so that they ensure that every state containing some formula $\chi \mathrm{~U~} \psi$ will eventually be followed by some state containing ψ. Let $\chi_1 \mathrm{~U~} \psi_1, \ \ldots, \ \chi_k \mathrm{~U~} \psi_k$ 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 $1 \le i \le k$, the run has infinitely many states satisfying $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$. 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 $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$. Let us advance through all those finitely many states, taking the suffix of the run none of whose states satisfies $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$, i.e., all of whose states satisfy $(\chi_i \mathrm{~U~} \psi_i) \wedge \neg\psi_i$. That is precisely the sort of run we want to eliminate.

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

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 $\mathcal{M}, s \vDash \phi$ holds by constructing the automaton $A_{\neg\phi}$, combining it with $\mathcal{M}$,

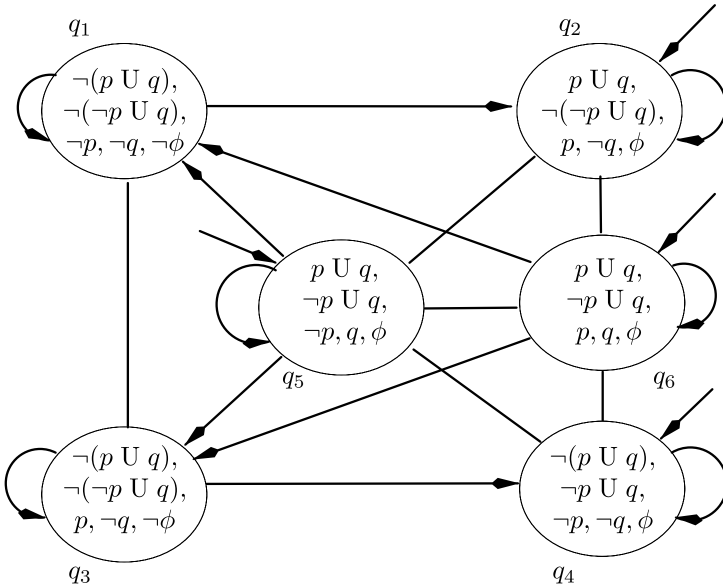


Figure 3.36. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} (p \text{ U } q) \vee (\neg p \text{ U } q)$. The transitions with no arrows can be taken in either direction. The acceptance condition asserts that every run must pass infinitely often through the set $\{q_1,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

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

3.7 The fixed-point characterisation of CTL

On page 227, we presented an algorithm which, given a CTL formula ϕ and a model $\mathcal{M} = (S, \to, L)$, computes the set of states $s \in S$ satisfying ϕ. We write this set as $[\![\phi]\!]$. The algorithm works recursively on the structure of ϕ. For formulas ϕ of height 1 ($\bot$, $\top$ or p), $[\![\phi]\!]$ 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 $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ and combines them in a certain way (in this case, by taking the union) in order to obtain $[\![\psi_1 \vee \psi_2]\!]$.

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

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 $\phi_1 \to \phi_2$.

Our aim in this section is to prove the termination and correctness of $\mathtt{SAT}_{\mathtt{AF}}$ and $\mathtt{SAT}_{\mathtt{EU}}$. In fact, we will also write a procedure $\mathtt{SAT}_{\mathtt{EG}}$ and prove its termination and correctness[1]. The procedure $\mathtt{SAT}_{\mathtt{EG}}$ is given in Figure 3.37 and is based on the intuitions given in Section 3.6.1: 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 $\mathrm{EG}\,\phi$ says that $s_0 \vDash \mathrm{EG}\,\phi$ holds iff there exists a computation path $s_0 \to s_1 \to s_2 \to \ldots$ such that $s_i \vDash \phi$ holds *for all* $i \geq 0$. We could instead express it as follows: $\mathrm{EG}\,\phi$ holds if ϕ holds and $\mathrm{EG}\,\phi$ holds in one of the successor states to the current state. This suggests the equivalence $\mathrm{EG}\,\phi \equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi$ which can easily be proved from the semantic definitions of the connectives.

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

[1] Section 3.6.1 handles $\mathrm{EG}\,\phi$ by translating it into $\neg\mathrm{AF}\,\neg\phi$, but we already noted in Section 3.6.1 that EG could be handled directly.

```
function SAT_EG (φ)
/* determines the set of states satisfying EG φ */
local var X, Y
begin
    Y := SAT (φ);
    X := ∅;
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∩ pre_∃(Y)
    end
    return Y
end
```

Figure 3.37. The pseudo-code for $\mathtt{SAT}_{\mathrm{EG}}$.

3.7.1 Monotone functions

Definition 3.22 Let S be a set of states and $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ 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 $S \stackrel{\mathrm{def}}{=} \{s_0, s_1\}$ and $F(Y) \stackrel{\mathrm{def}}{=} Y \cup \{s_0\}$ 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 $G\colon \mathcal{P}(S) \to \mathcal{P}(S)$, which is *not* monotone, is given by

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

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 $\mathtt{SAT_{EU}}$ and $\mathtt{SAT_{AF}}$ code up such fixed-point computations, and are correct by item 2.

Notation 3.23 $F^i(X)$ means

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

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 $(F^1, F^2, F^3, \ldots)$ stabilises in such a way. For example, this won't happen for the function G defined above (see Exercise 1(d) on page 253). 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 $\{s_0, s_1, \ldots, s_n\}$ with $n+1$ elements. If $F \colon \mathcal{P}(S) \to \mathcal{P}(S)$ 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

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

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 $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ 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 $\mathtt{SAT}_{\mathtt{EG}}$

We saw at the end of the last section that $[\![\mathrm{EG}\,\phi]\!] = [\![\phi]\!] \cap \mathrm{pre}_\exists([\![\mathrm{EG}\,\phi]\!])$. This implies that $\mathrm{EG}\,\phi$ is a fixed point of the function $F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$. In fact, F is monotone, $\mathrm{EG}\,\phi$ is its greatest fixed point and therefore $\mathrm{EG}\,\phi$ 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, $[\![\mathrm{EG}\,\phi]\!]$ is the greatest fixed point of F, and $[\![\mathrm{EG}\,\phi]\!] = F^{n+1}(S)$.

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 $s_0 \to s_1$, we certainly have $s_0 \to s_1$, where $s_1 \in Y$, for X is a subset of Y. Thus, we showed $\mathrm{pre}_\exists(X) \subseteq \mathrm{pre}_\exists(Y)$ from which we readily conclude that $F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X) \subseteq [\![\phi]\!] \cap \mathrm{pre}_\exists(Y) = F(Y)$.
2. We have already seen that $[\![\mathrm{EG}\,\phi]\!]$ 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 $[\![\mathrm{EG}\,\phi]\!]$. So let s_0 be an element of such a fixed point X. We need to show that s_0 is in $[\![\mathrm{EG}\,\phi]\!]$ as well. For that we use the fact that

$$s_0 \in X = F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$$

   to infer that $s_0 \in [\![\phi]\!]$ and $s_0 \to s_1$ for some $s_1 \in X$; but, since s_1 is in X, we may apply that same argument to $s_1 \in X = F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$ and we get $s_1 \in [\![\phi]\!]$ and $s_1 \to s_2$ for some $s_2 \in X$. By mathematical induction, we can therefore construct an infinite path $s_0 \to s_1 \to \cdots \to s_n \to s_{n+1} \to \ldots$ such that $s_i \in [\![\phi]\!]$ for all $i \geq 0$. By the definition of $[\![\mathrm{EG}\,\phi]\!]$, this entails $s_0 \in [\![\mathrm{EG}\,\phi]\!]$.
3. The last item is now immediately accessible from the previous one and Theorem 3.24. □

Now we can see that the procedure $\mathtt{SAT_{EG}}$ is correctly coded and terminates. First, note that the line $Y := Y \cap \mathrm{pre}_\exists(Y)$ in the procedure $\mathtt{SAT_{EG}}$ (Figure 3.37) could be changed to $Y := \mathtt{SAT}(\phi) \cap \mathrm{pre}_\exists(Y)$ without changing the effect of the procedure. To see this, note that the first time round the loop, Y *is* $\mathtt{SAT}(\phi)$; and in subsequent loops, $Y \subseteq \mathtt{SAT}(\phi)$, so it doesn't matter whether we intersect with Y or $\mathtt{SAT}(\phi)$[2]. With the change, it is clear that $\mathtt{SAT_{EG}}$ is calculating the greatest fixed point of F; therefore its correctness follows from Theorem 3.25.

3.7.3 The correctness of $\mathtt{SAT_{EU}}$

Proving the correctness of $\mathtt{SAT_{EU}}$ is similar. We start by noting the equivalence $\mathrm{E}[\phi \;\mathrm{U}\; \psi] \equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi \;\mathrm{U}\; \psi])$ and we write it as $[\![\mathrm{E}[\phi \;\mathrm{U}\; \psi]]\!] = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists [\![\mathrm{E}[\phi \;\mathrm{U}\; \psi]]\!])$. That tells us that $[\![\mathrm{E}[\phi \;\mathrm{U}\; \psi]]\!]$ is a fixed point of the function $G(X) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists(X))$. As before, we can prove that this function is monotone. It turns out that $[\![\mathrm{E}[\phi \;\mathrm{U}\; \psi]]\!]$ is its *least* fixed point and that the function $\mathtt{SAT_{EU}}$ 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, $[\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$ is the least fixed point of G, and we have $[\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!] = G^{n+1}(\emptyset)$.

[2] 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:

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

The last of these equalities follows from the monotonicity of $\mathrm{pre}_\exists$.

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

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

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

A formal proof would follow by induction on i.

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 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 $[\![\phi]\!]$ and $[\![\psi]\!]$.
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 $[\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$. Simply observe what kind of states we obtain by iterating G on the empty set $\emptyset$: $G^1(\emptyset) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \text{pre}_\exists([\![\emptyset]\!])) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \emptyset) = [\![\psi]\!] \cup \emptyset = [\![\psi]\!]$, which are all states $s_0 \in [\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$, where we chose $i = 0$ according to the definition of Until. Now,

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

tells us that the elements of $G^2(\emptyset)$ are all those $s_0 \in [\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$ 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 $s_0 \in [\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$. Since this holds for all k, we see that $[\![\mathrm{E}(\phi \;\mathrm{U}\; \psi)]\!]$ 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 $\texttt{SAT}_{\texttt{EU}}$ follows similarly to that of $\texttt{SAT}_{\texttt{EG}}$. We change the line $Y := Y \cup (W \cap \text{pre}_\exists(Y))$ into $Y := \texttt{SAT}(\psi) \cup (W \cap \text{pre}_\exists(Y))$ and observe that this does not change the result of the procedure, because the first time round the loop, Y is $\texttt{SAT}(\psi)$; and, since Y is always increasing, it makes no difference whether we perform a union with Y or with $\texttt{SAT}(\psi)$. Having made that change, it is then clear that $\texttt{SAT}_{\texttt{EU}}$ 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 $[\![\mathrm{EF}\,p]\!]$. By the definition of EF this is just $[\![\mathrm{E}(\top \;\mathrm{U}\; p)]\!]$. So we have $\phi_1 \stackrel{\text{def}}{=} \top$ and $\phi_2 \stackrel{\text{def}}{=} p$. From Figure 3.38, we obtain $[\![p]\!] = \{s_3\}$ and of course $[\![\top]\!] = S$. Thus, the function G above equals $G(X) = \{s_3\} \cup \text{pre}_\exists(X)$. Since $[\![\mathrm{E}(\top \;\mathrm{U}\; p)]\!]$ 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 $[\![\mathrm{E}(\top \;\mathrm{U}\; p)]\!]$ by Theorem 3.20. But then $[\![\mathrm{E}(\top \;\mathrm{U}\; p)]\!] = [\![\mathrm{EF}\,p]\!]$.

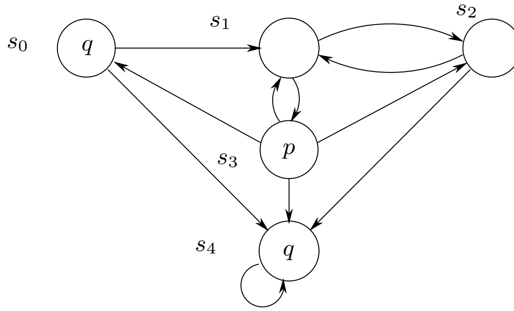


Figure 3.38. A system for which we compute invariants.

The other example we study is the computation of the set $[\![\mathrm{EG}\, q]\!]$. By Theorem 3.25, that set is the greatest fixed point of the function F above, where $\phi \stackrel{\text{def}}{=} q$. From Figure 3.38 we see that $[\![q]\!] = \{s_0, s_4\}$ and so $F(X) = [\![q]\!] \cap \mathrm{pre}_\exists(X) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(X)$. Since $[\![\mathrm{EG}\, q]\!]$ equals the greatest fixed point of F, we need to iterate F on S until this process stabilises. First, $F^1(S) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(S) = \{s_0, s_4\} \cap S$ since every s has some s' with $s \to s'$. Thus, $F^1(S) = \{s_0, s_4\}$.

Second, $F^2(S) = F(F^1(S)) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(\{s_0, s_4\}) = \{s_0, s_4\}$. Therefore, $\{s_0, s_4\}$ is the greatest fixed point of F, which equals $[\![\mathrm{EG}\, q]\!]$ 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 analyser according to the classification criteria for formal methods proposed on page 172.
2. Visit and browse the websites[3] and[4] to find formal methods that interest you for whatever reason. Then classify them according to the criteria from page 172.

Exercises 3.2

1. Draw parse trees for the LTL formulas:
   (a) $\mathrm{F}\, p \wedge \mathrm{G}\, q \to p \mathrel{\mathrm{W}} r$
   (b) $\mathrm{F}\,(p \to \mathrm{G}\, r) \vee \neg q \mathrel{\mathrm{U}} p$
   (c) $p \mathrel{\mathrm{W}} (q \mathrel{\mathrm{W}} r)$
   (d) $\mathrm{G}\,\mathrm{F}\, p \to \mathrm{F}\,(q \vee s)$

[3] `www.afm.sbu.ac.uk`
[4] `www.cs.indiana.edu/formal-methods-education/`

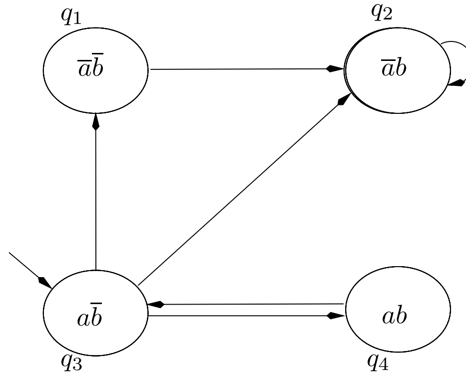


Figure 3.39. A model $\mathcal{M}$.

2. Consider the system of Figure 3.39. For each of the formulas ϕ:
   (a) $\mathrm{G}\, a$
   (b) $a \mathrm{~U~} b$
   (c) $a \mathrm{~U~X} (a \wedge \neg b)$
   (d) $\mathrm{X} \neg b \wedge \mathrm{G} (\neg a \vee \neg b)$
   (e) $\mathrm{X} (a \wedge b) \wedge \mathrm{F} (\neg a \wedge \neg b)$
      (i) Find a path from the initial state q_3 which satisfies ϕ.
      (ii) Determine whether $\mathcal{M}, q_3 \vDash \phi$.
3. Working from the clauses of Definition 3.1 (page 175), prove the equivalences:

$$\phi \mathrm{~U~} \psi \equiv \phi \mathrm{~W~} \psi \wedge \mathrm{F}\, \psi$$
$$\phi \mathrm{~W~} \psi \equiv \phi \mathrm{~U~} \psi \vee \mathrm{G}\, \phi$$
$$\phi \mathrm{~W~} \psi \equiv \psi \mathrm{~R~} (\phi \vee \psi)$$
$$\phi \mathrm{~R~} \psi \equiv \psi \mathrm{~W~} (\phi \wedge \psi)\,.$$

4. Prove that $\phi \mathrm{~U~} \psi \equiv \psi \mathrm{~R~} (\phi \vee \psi) \wedge \mathrm{F}\, \psi$.
5. List all subformulas of the LTL formula $\neg p \mathrm{~U~} (\mathrm{F}\, r \vee G \neg q \to q \mathrm{~W~} \neg r)$.
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, $\pi \vDash \phi \mathrm{~W~} \psi \wedge \mathrm{F}\, \psi$ implies $\pi \vDash \phi \mathrm{~U~} \psi$. That is, prove the remaining half of equivalence (3.2) on page 185.
8. Recall the algorithm `NNF` on page 62 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 193).
   * (a) Verify that `G(req -> F busy)` holds in all initial states.
   (b) Does $\neg$(`req` U $\neg$`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(req -> F busy)`. Then run it.
2. Verify Remark 3.11 from page 190.

* 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) $\mathrm{EG}\, r$
   * (b) $\mathrm{AG}\, (q \rightarrow \mathrm{EG}\, r)$
   * (c) $\mathrm{A}[p \mathrm{\ U\ EF}\, r]$
   * (d) $\mathrm{EF\, EG}\, p \rightarrow \mathrm{AF}\, r$, recall Convention 3.13
   (e) $\mathrm{A}[p \mathrm{\ U\ A}[q \mathrm{\ U\ } r]]$
   (f) $\mathrm{E}[\mathrm{A}[p \mathrm{\ U\ } q] \mathrm{\ U\ } r]$
   (g) $\mathrm{AG}\, (p \rightarrow \mathrm{A}[p \mathrm{\ U\ } (\neg p \wedge \mathrm{A}[\neg p \mathrm{\ U\ } q])])$.
2. Explain why the following are not well-formed CTL formulas:
   * (a) $\mathrm{F\, G}\, r$
   (b) $\mathrm{X\, X}\, r$
   (c) $\mathrm{A}\neg\mathrm{G}\,\neg p$
   (d) $\mathrm{F}\, [r \mathrm{\ U\ } q]$
   (e) $\mathrm{EX\, X}\, r$
   * (f) $\mathrm{AEF}\, r$
   * (g) $\mathrm{AF}\, [(r \mathrm{\ U\ } q) \wedge (p \mathrm{\ U\ } r)]$.
3. State which of the strings below are well-formed CTL formulas. For those which are well-formed, draw the parse tree. For those which are not well-formed, explain why not.

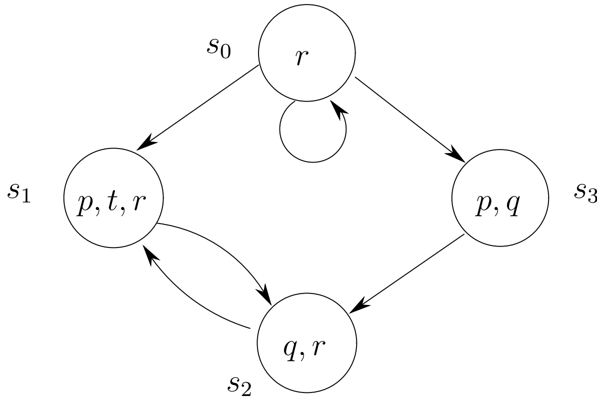


Figure 3.40. A model with four states.

(a) $\neg(\neg p) \vee (r \wedge s)$
(b) $\mathrm{X}\, q$
* (c) $\neg \mathrm{AX}\, q$
(d) $p\ \mathrm{U}\ (\mathrm{AX}\,\bot)$
* (e) $\mathrm{E}[(\mathrm{AX}\, q)\ \mathrm{U}\ (\neg(\neg p) \vee (\top \wedge s))]$
* (f) $(\mathrm{F}\, r) \wedge (\mathrm{AG}\, q)$
(g) $\neg(\mathrm{AG}\, q) \vee (\mathrm{EG}\, q)$.

* 4. List all subformulas of the formula $\mathrm{AG}\,(p \to \mathrm{A}[p\ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p\ \mathrm{U}\ q])])$.

5. Does $\mathrm{E}[\texttt{req}\ \mathrm{U}\ \neg\texttt{busy}]$ hold in all initial states of the model in Figure 3.9 on page 193?

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 $\mathcal{M}, s_0 \vDash \phi$ and $\mathcal{M}, s_2 \vDash \phi$ hold and justify your answer, where ϕ is the LTL or CTL formula:
   * (i) $\neg p \to r$
   (ii) $\mathrm{F}\, t$
   *(iii) $\neg \mathrm{EG}\ r$
   (iv) $\mathrm{E}\,(t\ \mathrm{U}\ q)$
   (v) $\mathrm{F}\, q$
   (vi) $\mathrm{EF}\ q$
   (vii) $\mathrm{EG}\ r$
   (viii) $\mathrm{G}\,(r \vee q)$.

7. Let $\mathcal{M} = (S, \to, L)$ be any model for CTL and let $[\![\phi]\!]$ denote the set of all $s \in S$ such that $\mathcal{M}, s \vDash \phi$. Prove the following set identities by inspecting the clauses of Definition 3.15 from page 211.
   * (a) $[\![\top]\!] = S$,
   (b) $[\![\bot]\!] = \emptyset$

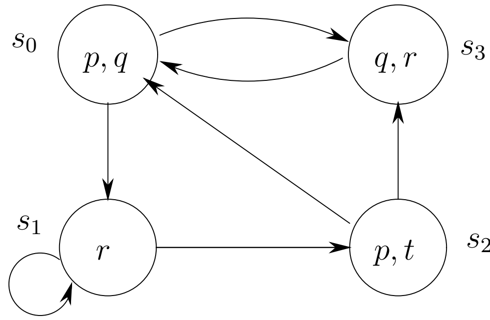


Figure 3.41. Another model with four states.

(c) $[\![\neg\phi]\!] = S - [\![\phi]\!]$,
(d) $[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$
(e) $[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!]$
* (f) $[\![\phi_1 \rightarrow \phi_2]\!] = (S - [\![\phi_1]\!]) \cup [\![\phi_2]\!]$
* (g) $[\![\mathrm{AX}\,\phi]\!] = S - [\![\mathrm{EX}\,\neg\phi]\!]$
(h) $[\![\mathrm{A}(\phi_2 \mathrm{~U~} \phi_2)]\!] = [\![\neg(\mathrm{E}(\neg\phi_1 \mathrm{~U~} (\neg\phi_1 \wedge \neg\phi_2)) \vee \mathrm{EG}\,\neg\phi_2)]\!]$.

8. Consider the model $\mathcal{M}$ in Figure 3.41. Check whether $\mathcal{M}, s_0 \vDash \phi$ and $\mathcal{M}, s_2 \vDash \phi$ hold for the CTL formulas ϕ:
   (a) $\mathrm{AF}\,q$
   (b) $\mathrm{AG}\,(\mathrm{EF}\,(p \vee r))$
   (c) $\mathrm{EX}\,(\mathrm{EX}\,r)$
   (d) $\mathrm{AG}\,(\mathrm{AF}\,q)$.

* 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, $\mathrm{EF}\,p$ 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 208 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) $\mathrm{EF}\,\phi$ and $\mathrm{EG}\,\phi$
   * (b) $\mathrm{EF}\,\phi \vee \mathrm{EF}\,\psi$ and $\mathrm{EF}\,(\phi \vee \psi)$
   * (c) $\mathrm{AF}\,\phi \vee \mathrm{AF}\,\psi$ and $\mathrm{AF}\,(\phi \vee \psi)$
   (d) $\mathrm{AF}\,\neg\phi$ and $\neg\mathrm{EG}\,\phi$
   * (e) $\mathrm{EF}\,\neg\phi$ and $\neg\mathrm{AF}\,\phi$
   (f) $\mathrm{A}[\phi_1 \mathrm{~U~} \mathrm{A}[\phi_2 \mathrm{~U~} \phi_3]]$ and $\mathrm{A}[\mathrm{A}[\phi_1 \mathrm{~U~} \phi_2] \mathrm{~U~} \phi_3]$, hint: it might make it simpler if you think first about models that have just one path
   (g) $\top$ and $\mathrm{AG}\,\phi \rightarrow \mathrm{EG}\,\phi$
   * (h) $\top$ and $\mathrm{EG}\,\phi \rightarrow \mathrm{AG}\,\phi$.

11. Find operators to replace the ?, to make the following equivalences:

* (a) $\mathrm{AG}\,(\phi \wedge \psi) \;\equiv\; \mathrm{AG}\,\phi \;?\; \mathrm{AG}\,\psi$

(b) $\mathrm{EF}\,\neg\phi \;\equiv\; \neg ?? \phi$

12. State explicitly the meaning of the temporal connectives AR etc., as defined on page 217.
13. Prove the equivalences (3.6) on page 216.

* 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 $\{\bot, \neg, \wedge, \mathrm{AF}, \mathrm{EU}, \mathrm{EX}\}$.

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 183 and 215 capture the stated ‘informal’ properties expressed in plain English.
3. Consider the set of LTL/CTL formulas $\mathcal{F} = \{\mathrm{F}\,p \to \mathrm{F}\,q, \mathrm{AF}\;p \to \mathrm{AF}\;q, \mathrm{AG}\;(p \to \mathrm{AF}\;q)\}$.
   (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 $\mathrm{AG}\,(p \to \mathrm{AF}\,(s \wedge \mathrm{AX}\,(\mathrm{AF}\,t)))$. Explain what exactly it expresses in terms of the order of occurrence of events p, s and t.
5. Extend the algorithm `NNF` from page 62 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 218.
6. Find a transition system which distinguishes the following pairs of CTL* formulas, i.e., show that they are not equivalent:
   (a) $\mathrm{AF}\,\mathrm{G}\,p$ and $\mathrm{AF}\,\mathrm{AG}\,p$
   * (b) $\mathrm{AG}\,\mathrm{F}\,p$ and $\mathrm{AG}\,\mathrm{EF}\,p$
   (c) $\mathrm{A}[(p\;\mathrm{U}\;r) \vee (q\;\mathrm{U}\;r)]$ and $\mathrm{A}[(p \vee q)\;\mathrm{U}\;r)]$

* (d) $\mathrm{A}[X\, p \vee \mathrm{X}\,\mathrm{X}\, p]$ and $\mathrm{AX}\, p \vee \mathrm{AX}\,\mathrm{AX}\, p$
(e) $\mathrm{E}[\mathrm{G}\,\mathrm{F}\, p]$ and $\mathrm{EG}\,\mathrm{EF}\, p$.

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) $\mathrm{E}[\mathrm{F}\, p \wedge (q \mathrm{~U~} r)]$
* (b) $\mathrm{E}[\mathrm{F}\, p \wedge \mathrm{G}\, q]$.
In this way, we have dealt with all formulas of the form $\mathrm{E}[\phi \wedge \psi]$. Formulas of the form $\mathrm{E}[\phi \vee \psi]$ can be rewritten as $\mathrm{E}[\phi] \vee \mathrm{E}[\psi]$ and $\mathrm{A}[\phi]$ can be written $\neg\mathrm{E}[\neg\phi]$. Use this translation to write the following in CTL:
(c) $\mathrm{E}[(p \mathrm{~U~} q) \wedge \mathrm{F}\, p]$
* (d) $\mathrm{A}[(p \mathrm{~U~} q) \wedge \mathrm{G}\, p]$
* (e) $\mathrm{A}[\mathrm{F}\, p \rightarrow \mathrm{F}\, q]$.
8. The aim of this exercise is to demonstrate the expansion given for AW at the end of the last section, i.e., $\mathrm{A}[p \mathrm{~W~} q] \equiv \neg\mathrm{E}[\neg q \mathrm{~U~} \neg(p \vee q)]$.
(a) Show that the following LTL formulas are valid (i.e., true in any state of any model):
(i) $\neg q \mathrm{~U~} (\neg p \wedge \neg q) \rightarrow \neg\mathrm{G}\, p$
(ii) $\mathrm{G}\,\neg q \wedge \mathrm{F}\,\neg p \rightarrow \neg q \mathrm{~U~} (\neg p \wedge \neg q)$.
(b) Expand $\neg((p \mathrm{~U~} q) \vee \mathrm{G}\, p)$ using de Morgan rules and the LTL equivalence $\neg(\phi \mathrm{~U~} \psi) \equiv (\neg\psi \mathrm{~U~} (\neg\phi \wedge \neg\psi)) \vee \neg\mathrm{F}\,\psi$.
(c) Using your expansion and the facts (i) and (ii) above, show $\neg((p \mathrm{~U~} q) \vee \mathrm{G}\, p) \equiv \neg q \mathrm{~U~} \neg(p \wedge q)$ 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 198. 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 196).
* 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 188).
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 191).
5. Prove that (3.8) on page 228 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 227 by writing a special procedure for $\mathrm{AG}\,\psi_1$, without rewriting it in terms of other formulas[5].

[5] Question: will your routine be more like the routine for AF, or more like that for EG on page 224? Why?

* 8. Write the pseudo-code for $\texttt{SAT}_{\mathrm{EG}}$, based on the description in terms of deleting labels given in Section 3.6.1.

* 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 $\vDash$ between states and CTL formulas to explain why $s \vDash \mathrm{AG\,AF}\,\phi$ 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 $s_0 \to s_1 \to s_2 \to \dots$ iff for all $n \geq 0$ there is some $m \geq n$ such that $s_m \vDash \phi$.

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 $\mathrm{E}_C\mathrm{G}\,\top$ labels for Figure 3.11, given the fairness constraints of the code in Figure 3.10 on page 196.

Exercises 3.7

1. Consider the functions

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

defined by

$$H_1(Y) \stackrel{\mathrm{def}}{=} Y - \{1,4,7\}$$
$$H_2(Y) \stackrel{\mathrm{def}}{=} \{2,5,9\} - Y$$
$$H_3(Y) \stackrel{\mathrm{def}}{=} \{1,2,3,4,5\} \cap (\{2,4,8\} \cup Y)$$

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.

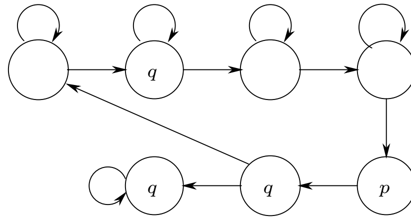


Figure 3.42. Another system for which we compute invariants.

(c) Does H_2 have any fixed points?

(d) Recall $G\colon \mathcal{P}(\{s_0, s_1\}) \to \mathcal{P}(\{s_0, s_1\})$ with

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

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 $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ be a monotone function. Show that:
   (a) $F_1 : \mathcal{P}(S) \to \mathcal{P}(S)$ with $F_1(Y) \stackrel{\text{def}}{=} A \cap F(Y)$ is monotone;
   (b) $F_2 : \mathcal{P}(S) \to \mathcal{P}(S)$ 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) $[\![\mathrm{EF}\, p]\!]$
   (b) $[\![\mathrm{EG}\, q]\!]$.

4. Using the function $F(X) = [\![\phi]\!] \cup \mathrm{pre}_\forall(X)$ prove that $[\![\mathrm{AF}\,\phi]\!]$ is the least fixed point of F. Hence argue that the procedure $\texttt{SAT}_{\texttt{AF}}$ is correct and terminates.

* 5. One may also compute $\mathrm{AG}\,\phi$ directly as a fixed point. Consider the function $H\colon \mathcal{P}(S) \to \mathcal{P}(S)$ with $H(X) = [\![\phi]\!] \cap \mathrm{pre}_\forall(X)$. Show that H is monotone and that $[\![\mathrm{AG}\,\phi]\!]$ is the greatest fixed point of H. Use that insight to write a procedure $\texttt{SAT}_{\texttt{AG}}$.

6. Similarly, one may compute $\mathrm{A}[\phi_1 \mathrm{~U~} \phi_2]$ directly as a fixed point, using $K\colon \mathcal{P}(S) \to \mathcal{P}(S)$, where $K(X) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathrm{pre}_\forall(X))$. Show that K is monotone and that $[\![\mathrm{A}[\phi_1 \mathrm{~U~} \phi_2]]\!]$ is the least fixed point of K. Use that insight to write a procedure $\texttt{SAT}_{\texttt{AU}}$. Can you use that routine to handle all calls of the form $\mathrm{AF}\,\phi$ as well?

7. Prove that $[\![\mathrm{A}[\phi_1 \mathrm{~U~} \phi_2]]\!] = [\![\phi_2 \vee (\phi_1 \wedge \mathrm{AX}\,(\mathrm{A}[\phi_1 \mathrm{~U~} \phi_2]))]\!]$.

8. Prove that $[\![\mathrm{AG}\,\phi]\!] = [\![\phi \wedge \mathrm{AX}\,(\mathrm{AG}\,\phi)]\!]$.

9. Show that the repeat-statements in the code for $\texttt{SAT}_{\texttt{EU}}$ and $\texttt{SAT}_{\texttt{EG}}$ 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 University[6]. NuSMV[7] is a reimplementation, developed in Trento by A. Cimatti, and M. Roveri and is aimed at being customisable and extensible. Extensive documentation 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 SMV[8] 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 website 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. Dillon[9].

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 website[10]. A model checker called FDR2 based on the process algebra CSP is available[11].

[6] `www.cs.cmu.edu/~modelcheck/`
[7] `nusmv.irst.itc.it`
[8] `www-cad.eecs.berkeley.edu/~kenmcmil/`
[9] `patterns.projects.cis.ksu.edu/`
[10] `netlib.bell-labs.com/netlib/spin/whatispin.html`
[11] `www.fsel.com.fdr2_download.html`

The Edinburgh Concurrency Workbench[12] and the Concurrency Workbench of North Carolina[13] 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 Bogor[14].

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/`.

[12] `www.dcs.ed.ac.uk/home/cwb`
[13] `www.cs.sunysb.edu/~cwb`
[14] `http://bogor.projects.cis.ksu.edu/`

4

Program verification

The methods of the previous chapter are suitable for verifying systems of communicating processes, where control is the main issue, but there are no complex *data*. We relied on the fact that those (abstracted) systems are in a *finite state*. These assumptions are not valid for sequential programs running on a single processor, the topic of this chapter. In those cases, the programs may manipulate non-trivial data and – once we admit variables of type integer, list, or tree – we are in the domain of machines with *infinite* state space.

In terms of the classification of verification methods given at the beginning of the last chapter, the methods of this chapter are

Proof-based. We do not exhaustively check every state that the system can get in to, as one does with model checking; this would be impossible, given that program variables can have infinitely many interacting values. Instead, we construct a proof that the system satisfies the property at hand, using a proof calculus. This is analogous to the situation in Chapter 2, where using a suitable proof calculus avoided the problem of having to check infinitely many models of a set of predicate logic formulas in order to establish the validity of a sequent.

Semi-automatic. Although many of the steps involved in proving that a program satisfies its specification are mechanical, there are some steps that involve some intelligence and that cannot be carried out algorithmically by a computer. As we will see, there are often good heuristics to help the programmer complete these tasks. This contrasts with the situation of the last chapter, which was fully automatic.

Property-oriented. Just like in the previous chapter, we verify properties of a program rather than a full specification of its behaviour.

Application domain. The domain of application in this chapter is sequential transformational programs. 'Sequential' means that we assume the program runs on a single processor and that there are no concurrency issues. 'Transformational' means that the program takes an input and, after some computation, is expected to terminate with an output. For example, methods of objects in Java are often programmed in this style. This contrasts with the previous chapter which focuses on reactive systems that are not intended to terminate and that react continually with their environment.

Pre/post-development. The techniques of this chapter should be used during the coding process for small fragments of program that perform an identifiable (and hence, specifiable) task and hence should be used during the development process in order to avoid functional bugs.

4.1 Why should we specify and verify code?

The task of specifying and verifying code is often perceived as an unwelcome addition to the programmer's job and a dispensable one. Arguments in favour of verification include the following:

- **Documentation:** The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds.
- **Time-to-market:** Debugging big systems during the testing phase is costly and time-consuming and local 'fixes' often introduce new bugs at other places. Experience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components.
- **Refactoring:** Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do.
- **Certification audits:** Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: a guarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty.

The degree to which the software industry accepts the benefits of proper verification of code depends on the perceived extra cost of producing it and the perceived benefits of having it. As verification technology improves, the costs are declining; and as the complexity of software and the extent to which society depends on it increase, the benefits are becoming more important. Thus, we can expect that the importance of verification to industry will continue to increase over the next decades. Microsoft's emergent technology A# combines program verification, testing, and model-checking techniques in an integrated in-house development environment.

Currently, many companies struggle with a legacy of ancient code without proper documentation which has to be adapted to new hardware and network environments, as well as ever-changing requirements. Often, the original programmers who might still remember what certain pieces of code are for have moved, or died. Software systems now often have a longer life-expectancy than humans, which necessitates a durable, transparent and portable design and implementation process; the year-2000 problem was just one such example. Software verification provides some of this.

4.2 A framework for software verification

Suppose you are working for a software company and your task is to write programs which are meant to solve sophisticated problems, or computations. Typically, such a project involves an outside customer – a utility company, for example – who has written up an informal description, in plain English, of the real-world task that is at hand. In this case, it could be the development and maintenance of a database of electricity accounts with all the possible applications of that – automated billing, customer service etc. Since the informality of such descriptions may cause ambiguities which eventually could result in serious and expensive design flaws, it is desirable to condense all the requirements of such a project into formal specifications. These formal specifications are usually symbolic encodings of real-world constraints into some sort of logic. Thus, a framework for producing the software could be:

- Convert the informal description R of requirements for an application domain into an 'equivalent' formula ϕ_R of some symbolic logic;
- Write a program P which is meant to realise ϕ_R in the programming environment supplied by your company, or wanted by the particular customer;
- *Prove* that the program P satisfies the formula ϕ_R.

This scheme is quite crude – for example, constraints may be actual design decisions for interfaces and data types, or the specification may 'evolve'

and may partly be 'unknown' in big projects – but it serves well as a first approximation to trying to define good programming methodology. Several variations of such a sequence of activities are conceivable. For example, you, as a programmer, might have been given only the formula ϕ_R, so you might have little if any insight into the real-world problem which you are supposed to solve. Technically, this poses no problem, but often it is handy to have both informal and formal descriptions available. Moreover, crafting the informal requirements R is often a mutual process between the client and the programmer, whereby the attempt at formalising R can uncover ambiguities or undesired consequences and hence lead to revisions of R.

This 'going back and forth' between the realms of informal and formal specifications is necessary since it is impossible to 'verify' whether an *informal* requirement R is equivalent to a *formal* description ϕ_R. The meaning of R as a piece of natural language is grounded in common sense and general knowledge about the real-world domain and often based on heuristics or quantitative reasoning. The meaning of a logic formula ϕ_R, on the other hand, is defined in a precise mathematical, qualitative and compositional way by structural induction on the parse tree of ϕ_R – the first three chapters contain examples of this.

Thus, the process of finding a suitable formalisation ϕ_R of R requires the utmost care; otherwise it is always possible that ϕ_R specifies behaviour which is different from the one described in R. To make matters worse, the requirements R are often inconsistent; customers usually have a fairly vague conception of what exactly a program should do for them. Thus, producing a clear and coherent description R of the requirements for an application domain is already a crucial step in successful programming; this phase ideally is undertaken by customers and project managers around a table, or in a video conference, talking to each other. We address this first item only implicitly in this text, but you should certainly be aware of its importance in practice.

The next phase of the software development framework involves constructing the program P and after that the last task is to verify that P satisfies ϕ_R. Here again, our framework is oversimplifying what goes on in practice, since often proving that P satisfies its specification ϕ_R goes hand-in-hand with inventing a suitable P. This correspondence between proving and programming can be stated quite precisely, but that is beyond the scope of this book.

4.2.1 A core programming language

The programming language which we set out to study here is the typical core language of most imperative programming languages. Modulo trivial

syntactic variations, it is a subset of Pascal, C, C++ and Java. Our language consists of assignments to integer- and boolean-valued variables, if-statements, while-statements and sequential compositions. Everything that can be computed by large languages like C and Java can also be computed by our language, though perhaps not as conveniently, because it does not have any objects, procedures, threads or recursive data structures. While this makes it seem unrealistic compared with fully blown commercial languages, it allows us to focus our discussion on the process of formal program verification. The features missing from our language could be implemented on top of it; that is the justification for saying that they do not add to the power of the language, but only to the convenience of using it. Verifying programs using those features would require non-trivial extensions of the proof calculus we present here. In particular, dynamic scoping of variables presents hard problems for program-verification methods, but this is beyond the scope of this book.

Our core language has three syntactic domains: integer expressions, boolean expressions and commands – the latter we consider to be our programs. Integer expressions are built in the familiar way from variables $x, y, z, \ldots$, numerals $0, 1, 2, \ldots, -1, -2, \ldots$ and basic operations like addition $(+)$ and multiplication $(*)$. For example,

$$
\begin{array}{l}
5 \\
x \\
4 + (x - 3) \\
x + (x * (y - (5 + z)))
\end{array}
$$

are all valid integer expressions. Our grammar for generating integer expressions is

$$E \ ::= \ n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E) \tag{4.1}$$

where n is any numeral in $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and x is any variable. Note that we write multiplication in 'mathematics' as $2 \cdot 3$, whereas our core language writes $2 * 3$ instead.

Convention 4.1 In the grammar above, negation $-$ binds more tightly than multiplication $*$, which binds more tightly than subtraction $-$ and addition $+$.

Since if-statements and while-statements contain conditions in them, we also need a syntactic domain B of boolean expressions. The grammar in

Backus Naur form

$$B \ ::= \ \texttt{true} \ | \ \texttt{false} \ | \ (!B) \ | \ (B \,\&\, B) \ \ | \ (B \,||\, B) \ | \ (E < E) \qquad (4.2)$$

uses `!` for the negation, `&` for conjunction and `||` for disjunction of boolean expressions. This grammar may be freely expanded by operators which are definable in terms of the above. For example, the test for equality[1] $E_1 == E_2$ may be expressed via $!(E_1 < E_2) \,\&\, !(E_2 < E_1)$. We generally make use of shorthand notation whenever this is convenient. We also write $(E_1 \,!\!= E_2)$ to abbreviate $!(E_1 == E_2)$. We will also assume the usual binding priorities for logical operators stated in Convention 1.3 on page 5. Boolean expressions are built on top of integer expressions since the last clause of (4.2) mentions integer expressions.

Having integer and boolean expressions at hand, we can now define the syntactic domain of commands. Since commands are built from simpler commands using assignments and the control structures, you may think of commands as the actual programs. We choose as grammar for commands

$$C \ \ ::= \ \ \mathrm{x} = E \ | \ C; C \ | \ \texttt{if } B \ \{C\} \texttt{ else } \{C\} \ | \texttt{while } B \ \{C\} \qquad (4.3)$$

where the braces { and } are to mark the extent of the blocks of code in the if-statement and the while-statement, as in languages such as C and Java. They can be omitted if the blocks consist of a single statement. The intuitive meaning of the programming constructs is the following:

1. The atomic command `x` $= E$ is the usual assignment statement; it evaluates the integer expression E in the current state of the store and then overwrites the current value stored in x with the result of that evaluation.
2. The compound command $C_1; C_2$ is the sequential composition of the commands C_1 and C_2. It begins by executing C_1 in the current state of the store. If that execution terminates, then it executes C_2 in the storage state resulting from the execution of C_1. Otherwise – if the execution of C_1 does not terminate – the run of $C_1; C_2$ also does not terminate. Sequential composition is an example of a *control structure* since it implements a certain policy of flow of control in a computation.

[1] In common with languages like C and Java, we use a single equals sign = to mean assignment and a double sign == to mean equality. Earlier languages like Pascal used := for assignment and simple = for equality; it is a great pity that C and its successors did not keep this convention. The reason that = is a bad symbol for assignment is that assignment is not symmetric: if we interpret $x = y$ as the assignment, then x becomes y which is not the same thing as y becoming x; yet, $x = y$ and $y = x$ are the same thing if we mean equality. The two dots in := helped remind the reader that this is an asymmetric assignment operation rather than a symmetric assertion of equality. However, the notation = for assignment is now commonplace, so we will use it.

3. Another control structure is `if` B $\{C_1\}$ `else` $\{C_2\}$. It first evaluates the boolean expression B in the current state of the store; if that result is true, then C_1 is executed; if B evaluated to false, then C_2 is executed.
4. The third control construct `while` B $\{C\}$ allows us to write statements which are executed repeatedly. Its meaning is that:

   a the boolean expression B is evaluated in the current state of the store;
   b if B evaluates to false, then the command terminates,
   c otherwise, the command C will be executed. If that execution terminates, then we resume at step (a) with a re-evaluation of B as the updated state of the store may have changed its value.

   The point of the while-statement is that it repeatedly executes the command C as long as B evaluates to true. If B never becomes false, or if one of the executions of C does not terminate, then the while-statement will not terminate. While-statements are the only real source of non-termination in our core programming language.

Example 4.2 The factorial $n!$ of a natural number n is defined inductively by

$$\begin{aligned} 0! &\stackrel{\text{def}}{=} 1 \\ (n+1)! &\stackrel{\text{def}}{=} (n+1) \cdot n! \end{aligned} \tag{4.4}$$

For example, unwinding this definition for n being 4, we get $4! \stackrel{\text{def}}{=} 4 \cdot 3! = \cdots = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 24$. The following program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

is intended to compute the factorial[2] of x and to store the result in y. We will prove that `Fac1` really does this later in the chapter.

4.2.2 Hoare triples

Program fragments generated by (4.3) commence running in a 'state' of the machine. After doing some computation, they might terminate. If they do, then the result is another, usually different, state. Since our programming

[2] Please note the difference between the formula $x! = y$, saying that the factorial of x is equal to y, and the piece of code `x != y` which says that `x` is not equal to `y`.

language does not have any procedures or local variables, the 'state' of the machine can be represented simply as a vector of values of all the variables used in the program.

What syntax should we use for ϕ_R, the formal specifications of requirements for such programs? Because we are interested in the output of the program, the language should allow us to talk about the variables in the state after the program has executed, using operators like $=$ to express equality and $<$ for less than. You should be aware of the overloading of $=$. In code, it represents an assignment instruction; in logical formulas, it stands for equality, which we write `==` within program code.

For example, if the informal requirement R says that we should

> Compute a number y whose square is less than the input x.

then an appropriate specification may be $y \cdot y < x$. But what if the input x is -4? There is no number whose square is less than a negative number, so it is not possible to write the program in a way that it will work with all possible inputs. If we go back to the client and say this, he or she is quite likely to respond by saying that the requirement is only that the program work for positive numbers; i.e., he or she *revises* the informal requirement so that it now says

> If the input x is a positive number, compute a number whose square is less than x.

This means we need to be able to talk not just about the state *after* the program executes, but also about the state *before* it executes. The assertions we make will therefore be triples, typically looking like

$$(\!|\phi|\!)\ P\ (\!|\psi|\!) \tag{4.5}$$

which (roughly) means:

> If the program P is run in a state that satisfies ϕ, then the state resulting from P's execution will satisfy ψ.

The specification of the program P, to calculate a number whose square is less than x, now looks like this:

$$(\!|x > 0|\!)\ P\ (\!|y \cdot y < x|\!). \tag{4.6}$$

It means that, if we run P in a state such that $x > 0$, then the resulting state will be such that $y \cdot y < x$. It does not tell us what happens if we run P in a state in which $x \leq 0$, the client required nothing for non-positive values of x. Thus, the programmer is free to do what he or she wants in that case. A program which produces 'garbage' in the case that $x \leq 0$ satisfies the specification, as long as it works correctly for $x > 0$.

Let us make these notions more precise.

Definition 4.3 1. The form $(\!|\phi|\!)\, P\, (\!|\psi|\!)$ of our specification is called a Hoare triple, after the computer scientist C. A. R. Hoare.
2. In (4.5), the formula ϕ is called the precondition of P and ψ is called the postcondition.
3. A store or state of core programs is a function l that assigns to each variable x an integer $l(x)$.
4. For a formula ϕ of predicate logic with function symbols $-$ (unary), $+$, $-$, and $*$ (binary); and a binary predicate symbols $<$ and $=$, we say that a state l satisfies ϕ or l is a ϕ-state – written $l \vDash \phi$ – iff $\mathcal{M} \vDash_l \phi$ from page 128 holds, where l is viewed as a look-up table and the model $\mathcal{M}$ has as set A all integers and interprets the function and predicate symbols in their standard manner.
5. For Hoare triples in (4.5), we demand that quantifiers in ϕ and ψ only bind variables that do not occur in the program P.

Example 4.4 For any state l for which $l(x) = -2$, $l(y) = 5$, and $l(z) = -1$, the relation

1. $l \vDash \neg(x + y < z)$ holds since $x + y$ evaluates to $-2 + 5 = 3$, z evaluates to $l(z) = -1$, and 3 is not strictly less than -1;
2. $l \vDash y - x * z < z$ does not hold, since the lefthand expression evaluates to $5 - (-2) \cdot (-1) = 3$ which is not strictly less than $l(z) = -1$;
3. $l \vDash \forall u\, (y < u \to y * z < u * z)$ does not hold; for u being 7, $l \vDash y < u$ holds, but $l \vDash y * z < u * z$ does not.

Often, we do not want to put any constraints on the initial state; we simply wish to say that, no matter what state we start the program in, the resulting state should satisfy ψ. In that case the precondition can be set to $\top$, which is – as in previous chapters – a formula which is true in any state.

Note that the triple in (4.6) does not specify a unique program P, or a unique behaviour. For example, the program which simply does `y = 0;` satisfies the specification – since $0 \cdot 0$ is less than any positive number – as does the program

```
y = 0;
while (y * y < x) {
   y = y + 1;
   }
y = y - 1;
```

This program finds the greatest y whose square is less than x; the while-statement overshoots a bit, but then we fix it after the while-statement.[3]

[3] We could avoid this inelegance by using the `repeat` construct of exercise 3 on page 299.

Note that these two programs have different behaviour. For example, if x is 22, the first one will compute $y = 0$ and the second will render $y = 4$; but both of them satisfy the specification.

Our agenda, then, is to develop a notion of proof which allows us to prove that a program P satisfies the specification given by a precondition ϕ and a postcondition ψ in (4.5). Recall that we developed proof calculi for propositional and predicate logic where such proofs could be accomplished by investigating the structure of the formula one wanted to prove. For example, for proving an implication $\phi \to \psi$ one had to assume ϕ and manage to show ψ; then the proof could be finished with the proof rule for implies-introduction. The proof calculi which we are about to develop follow similar lines. Yet, they are different from the logics we previously studied since they prove triples which are built from two different sorts of things: logical formulas ϕ and ψ versus a piece of code P. Our proof calculi have to address each of these appropriately. Nonetheless, we retain proof strategies which are *compositional*, but now in the structure of P. Note that this is an important advantage in the verification of big projects, where code is built from a multitude of modules such that the correctness of certain parts will depend on the correctness of certain others. Thus, your code might call subroutines which other members of your project are about to code, but you can already check the correctness of your code by assuming that the subroutines meet their own specifications. We will explore this topic in Section 4.5.

4.2.3 Partial and total correctness

Our explanation of when the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds was rather informal. In particular, it did not say what we should conclude if P does not terminate. In fact there are two ways of handling this situation. *Partial correctness* means that we do not require the program to terminate, whereas in *total correctness* we insist upon its termination.

Definition 4.5 (Partial correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under partial correctness if, for all states which satisfy ϕ, the state resulting from P's execution satisfies the postcondition ψ, provided that P actually terminates. In this case, the relation $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds. We call $\vDash_{\mathsf{par}}$ the satisfaction relation for partial correctness.

Thus, we insist on ψ being true of the resulting state only if the program P has terminated on an input satisfying ϕ. Partial correctness is rather a weak requirement, since any program which does not terminate at all satisfies its

specification. In particular, the program

```
while true { x = 0; }
```

– which endlessly 'loops' and never terminates – satisfies all specifications, since partial correctness only says what must happen *if* the program terminates.

Total correctness, on the other hand, requires that the program terminates in order for it to satisfy a specification.

Definition 4.6 (Total correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under total correctness if, for all states in which P is executed which satisfy the precondition ϕ, P is guaranteed to terminate and the resulting state satisfies the postcondition ψ. In this case, we say that $\vDash_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds and call $\vDash_{\mathsf{tot}}$ the satisfaction relation of total correctness.

A program which 'loops' forever on all input does not satisfy any specification under total correctness. Clearly, total correctness is more useful than partial correctness, so the reader may wonder why partial correctness is introduced at all. Proving total correctness usually benefits from proving partial correctness first and then proving termination. So, although our primary interest is in proving total correctness, it often happens that we have to or may wish to split this into separate proofs of partial correctness and of termination. Most of this chapter is devoted to the proof of partial correctness, though we return to the issue of termination in Section 4.4.

Before we delve into the issue of crafting sound and complete proof calculi for partial and total correctness, let us briefly give examples of typical sorts of specifications which we would like to be able to prove.

Examples 4.7

1. Let `Succ` be the program

```
a = x + 1;
if (a - 1 == 0) {
    y = 1;
} else {
    y = a;
}
```

The program `Succ` satisfies the specification $(\!|\top|\!)$ `Succ` $(\!|y = (x + 1)|\!)$ under partial and total correctness, so if we think of x as input and y as output, then `Succ` computes the successor function. Note that this code is far from optimal.

In fact, it is a rather roundabout way of implementing the successor function. Despite this non-optimality, our proof rules need to be able to prove this program behaviour.

2. The program `Fac1` from Example 4.2 terminates only if x is initially non-negative – why? Let us look at what properties of `Fac1` we expect to be able to prove.

   We should be able to prove that $\vDash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Fac1` $(\!| y = x! |\!)$ holds. It states that, provided $x \geq 0$, `Fac1` terminates with the result $y = x!$. However, the stronger statement that $\vDash_{\mathsf{tot}} (\!| \top |\!)$ `Fac1` $(\!| y = x! |\!)$ holds should not be provable, because `Fac1` does not terminate for negative values of x.

   For partial correctness, both statements $\vDash_{\mathsf{par}} (\!| x \geq 0 |\!)$ `Fac1` $(\!| y = x! |\!)$ and $\vDash_{\mathsf{par}} (\!| \top |\!)$ `Fac1` $(\!| y = x! |\!)$ should be provable since they hold.

Definition 4.8 1. If the partial correctness of triples $(\!| \phi |\!)\ P\ (\!| \psi |\!)$ can be proved in the partial-correctness calculus we develop in this chapter, we say that the sequent $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ is valid.

2. Similarly, if it can be proved in the total-correctness calculus to be developed in this chapter, we say that the sequent $\vdash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ is valid.

Thus, $\vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ holds if P is partially correct, while the validity of $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ means that P can be proved to be partially-correct by our calculus. The first one means it is actually correct, while the second one means it is provably correct according to our calculus.

If our calculus is any good, then the relation $\vdash_{\mathsf{par}}$ should be contained in $\vDash_{\mathsf{par}}$! More precisely, we will say that our calculus is *sound* if, whenever it tells us something can be proved, that thing is indeed true. Thus, it is sound if it doesn't tell us that false things can be proved. Formally, we write that $\vdash_{\mathsf{par}}$ is sound if

$$\vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds whenever } \vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid}$$

for all ϕ, ψ and P; and, similarly, $\vdash_{\mathsf{tot}}$ is sound if

$$\vDash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds whenever } \vdash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid}$$

for all ϕ, ψ and P. We say that a calculus is *complete* if it is able to prove everything that is true. Formally, $\vdash_{\mathsf{par}}$ is complete if

$$\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid whenever } \vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds}$$

for all ϕ, ψ and P; and similarly for $\vdash_{\mathsf{tot}}$ being complete.

In Chapters 1 and 2, we said that soundness is relatively easy to show, since typically the soundness of individual proof rules can be established independently of the others. Completeness, on the other hand, is harder to

show since it depends on the entire set of proof rules cooperating together. The same situation holds for the program logic we introduce in this chapter. Establishing its soundness is simply a matter of considering each rule in turn – done in exercise 3 on page 303 – whereas establishing its (relative) completeness is harder and beyond the scope of this book.

4.2.4 Program variables and logical variables

The variables which we have seen so far in the programs that we verify are called *program variables.* They can also appear in the preconditions and postconditions of specifications. Sometimes, in order to formulate specifications, we need to use other variables which do not appear in programs.

Examples 4.9

1. Another version of the factorial program might have been `Fac2`:

```
y = 1;
while (x != 0) {
   y = y * x;
   x = x - 1;
   }
```

   Unlike the previous version, it 'consumes' the input x. Nevertheless, it correctly calculates the factorial of x and stores the value in y; and we would like to express that as a Hoare triple. However, it is not a good idea to write $(\!|x \geq 0|\!)$ `Fac2` $(\!|y = x!|\!)$ because, if the program terminates, then x will be 0 and y will be the factorial of the initial value of x.

   We need a way of remembering the initial value of x, to cope with the fact that it is modified by the program. Logical variables achieve just that: in the specification $(\!|x = x_0 \wedge x \geq 0|\!)$ `Fac2` $(\!|y = x_0!|\!)$ the x_0 is a logical variable and we read it as being universally quantified in the precondition. Therefore, this specification reads: for all integers x_0, if x equals x_0, $x \geq 0$ and we run the program such that it terminates, then the resulting state will satisfy y equals $x_0!$. This works since x_0 cannot be modified by `Fac2` as x_0 does not occur in `Fac2`.
2. Consider the program `Sum`:

```
z = 0;
while (x > 0) {
    z = z + x;
    x = x - 1;
    }
```

   This program adds up the first x integers and stores the result in z. Thus, $(\!|x = 3|\!)$ `Sum` $(\!|z = 6|\!)$, $(\!|x = 8|\!)$ `Sum` $(\!|z = 36|\!)$ etc. We know from Theorem 1.31 on page 41 that $1 + 2 + \cdots + x = x(x + 1)/2$ for all $x \geq 0$, so

we would like to express, as a Hoare triple, that the value of z upon termination is $x_0(x_0+1)/2$ where x_0 is the initial value of x. Thus, we write $(\!|x = x_0 \wedge x \geq 0|\!)$ `Sum` $(\!|z = x_0(x_0+1)/2|\!)$.

Variables like x_0 in these examples are called *logical variables*, because they occur only in the logical formulas that constitute the precondition and postcondition; they do not occur in the code to be verified. The state of the system gives a value to each program variable, but not for the logical variables. Logical variables take a similar role to the dummy variables of the rules for $\forall$i and $\exists$e in Chapter 2.

Definition 4.10 For a Hoare triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$, its set of logical variables are those variables that are free in ϕ or ψ; and don't occur in P.

4.3 Proof calculus for partial correctness

The proof calculus which we now present goes back to R. Floyd and C. A. R. Hoare. In the next subsection, we specify proof rules for each of the grammar clauses for commands. We could go on to use these proof rules directly, but it turns out to be more convenient to present them in a different form, suitable for the construction of proofs known as *proof tableaux*. This is what we do in the subsection following the next one.

4.3.1 Proof rules

The proof rules for our calculus are given in Figure 4.1. They should be interpreted as rules that allow us to pass from simple assertions of the form $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ to more complex ones. The rule for assignment is an axiom as it has no premises. This allows us to construct some triples out of nothing, to get the proof going. Complete proofs are trees, see page 274 for an example.

Composition. Given specifications for the program fragments C_1 and C_2, say

$$(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \quad \text{and} \quad (\!|\eta|\!)\ C_2\ (\!|\psi|\!),$$

where the postcondition of C_1 is also the precondition of C_2, the proof rule for sequential composition shown in Figure 4.1 allows us to derive a specification for $C_1; C_2$, namely

$$(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!).$$

$$\frac{(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \qquad (\!|\eta|\!)\ C_2\ (\!|\psi|\!)}{(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!)}\ \text{Composition}$$

$$\frac{}{(\!|\psi[E/x]|\!)\ x = E\ (\!|\psi|\!)}\ \text{Assignment}$$

$$\frac{(\!|\phi \wedge B|\!)\ C_1\ (\!|\psi|\!) \qquad (\!|\phi \wedge \neg B|\!)\ C_2\ (\!|\psi|\!)}{(\!|\phi|\!)\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)}\ \text{If-statement}$$

$$\frac{(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)}{(\!|\psi|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\psi \wedge \neg B|\!)}\ \text{Partial-while}$$

$$\frac{\vdash_{\mathrm{AR}} \phi' \to \phi \qquad (\!|\phi|\!)\ C\ (\!|\psi|\!) \qquad \vdash_{\mathrm{AR}} \psi \to \psi'}{(\!|\phi'|\!)\ C\ (\!|\psi'|\!)}\ \text{Implied}$$

Figure 4.1. Proof rules for partial correctness of Hoare triples.

Thus, if we know that C_1 takes ϕ-states to η-states and C_2 takes η-states to ψ-states, then running C_1 and C_2 in that sequence will take ϕ-states to ψ-states.

Using the proof rules of Figure 4.1 in program verification, we have to read them bottom-up: e.g. in order to prove $(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!)$, we need to find an appropriate η and prove $(\!|\phi|\!)\ C_1\ (\!|\eta|\!)$ and $(\!|\eta|\!)\ C_2\ (\!|\psi|\!)$. If $C_1; C_2$ runs on input satisfying ϕ and we need to show that the store satisfies ψ after its execution, then we hope to show this by splitting the problem into two. After the execution of C_1, we have a store satisfying η which, considered as input for C_2, should result in an output satisfying ψ. We call η a *midcondition*.

Assignment. The rule for assignment has no premises and is therefore an axiom of our logic. It tells us that, if we wish to show that ψ holds in the state after the assignment `x = E`, we must show that $\psi[E/x]$ holds before the assignment; $\psi[E/x]$ denotes the formula obtained by taking ψ and replacing all free occurrences of x with E as defined on page 105. We read the stroke as 'in place of;' thus, $\psi[E/x]$ is ψ with E in place of x. Several explanations may be required to understand this rule.

- At first sight, it looks as if the rule has been stated in reverse; one might expect that, if ψ holds in a state in which we perform the assignment `x = E`, then surely

$\psi[E/x]$ holds in the resulting state, i.e. we just replace x by E. This is wrong. It is true that the assignment `x = E` replaces the value of x in the starting state by E, but that does not mean that we replace occurrences of x in a *condition on* the starting state by E.

For example, let ψ be $x = 6$ and E be 5. Then $(\!|\psi|\!)$ `x = 5` $(\!|\psi[x/E]|\!)$ does *not* hold: given a state in which x equals 6, the execution of `x = 5` results in a state in which x equals 5. But $\psi[x/E]$ is the formula $5 = 6$ which holds in no state.

The right way to understand the Assignment rule is to think about what you would have to prove about the initial state in order to prove that ψ holds in the resulting state. Since ψ will – in general – be saying something about the value of x, whatever it says about that value must have been true of E, since in the resulting state the value of x is E. Thus, ψ with E in place of x – which says whatever ψ says about x but applied to E – must be true in the initial state.

- The axiom $(\!|\psi[E/x]|\!)\ x = E\ (\!|\psi|\!)$ is best applied backwards than forwards in the verification process. That is to say, if we know ψ and we wish to find ϕ such that $(\!|\phi|\!)\ x = E\ (\!|\psi|\!)$, it is easy: we simply set ϕ to be $\psi[E/x]$; but, if we know ϕ and we want to find ψ such that $(\!|\phi|\!)\ x = E\ (\!|\psi|\!)$, there is no easy way of getting a suitable ψ. This backwards characteristic of the assignment and the composition rule will be important when we look at how to construct proofs; we will work from the end of a program to its beginning.
- If we apply this axiom in this backwards fashion, then it is completely mechanical to apply. It just involves doing a substitution. That means we could get a computer to do it for us. Unfortunately, that is not true for all the rules; application of the rule for while-statements, for example, requires ingenuity. Therefore a computer can at best assist us in performing a proof by carrying out the mechanical steps, such as application of the assignment axiom, while leaving the steps that involve ingenuity to the programmer.
- Observe that, in computing $\psi[E/x]$ from ψ, we replace all the free occurrences of x in ψ. Note that there cannot be problems caused by *bound* occurrences, as seen in Example 2.9 on page 106, *provided that preconditions and postconditions quantify over logical variables only.* For obvious reasons, this is recommended practice.

Examples 4.11

1. Suppose P is the program `x = 2`. The following are instances of axiom Assignment:

   a $(\!|2 = 2|\!)P(\!|x = 2|\!)$
   b $(\!|2 = 4|\!)P(\!|x = 4|\!)$
   c $(\!|2 = y|\!)P(\!|x = y|\!)$
   d $(\!|2 > 0|\!)P(\!|x > 0|\!)$.

These are all correct statements. Reading them backwards, we see that they say:

a If you want to prove $x = 2$ after the assignment `x = 2`, then we must be able to prove that $2 = 2$ before it. Of course, 2 is equal to 2, so proving it shouldn't present a problem.

b If you wanted to prove that $x = 4$ after the assignment, the only way in which it would work is if $2 = 4$; however, unfortunately it is not. More generally, $(\!|\bot|\!)\ x = E\ (\!|\psi|\!)$ holds for any E and ψ – why?

c If you want to prove $x = y$ after the assignment, you will need to prove that $2 = y$ before it.

d To prove $x > 0$, we'd better have $2 > 0$ prior to the execution of P.

2. Suppose P is $\mathtt{x} = \mathtt{x} + \mathtt{1}$. By choosing various postconditions, we obtain the following instances of the assignment axiom:

   a $(\!|x + 1 = 2|\!)\,P\,(\!|x = 2|\!)$
   b $(\!|x + 1 = y|\!)\,P\,(\!|x = y|\!)$
   c $(\!|x + 1 + 5 = y|\!)\,P\,(\!|x + 5 = y|\!)$
   d $(\!|x + 1 > 0 \wedge y > 0|\!)\,P\,(\!|x > 0 \wedge y > 0|\!)$.

   Note that the precondition obtained by performing the substitution can often be simplified. The proof rule for implications below will allow such simplifications which are needed to make preconditions appreciable by human consumers.

If-statements. The proof rule for if-statements allows us to prove a triple of the form

$$(\!|\phi|\!)\ \mathtt{if}\ B\ \{C_1\}\ \mathtt{else}\ \{C_2\}\ (\!|\psi|\!)$$

by decomposing it into two triples, subgoals corresponding to the cases of B evaluating to true and to false. Typically, the precondition ϕ will not tell us anything about the value of the boolean expression B, so we have to consider both cases. If B is true in the state we start in, then C_1 is executed and hence C_1 will have to translate ϕ states to ψ states; alternatively, if B is false, then C_2 will be executed and will have to do that job. Thus, we have to prove that $(\!|\phi \wedge B|\!)\ C_1\ (\!|\psi|\!)$ and $(\!|\phi \wedge \neg B|\!)\ C_2\ (\!|\psi|\!)$. Note that the preconditions are augmented by the knowledge that B is true and false, respectively. This additional information is often crucial for completing the respective subproofs.

While-statements. The rule for while-statements given in Figure 4.1 is arguably the most complicated one. The reason is that the while-statement is the most complicated construct in our language. It is the only command that 'loops,' i.e. executes the same piece of code several times. Also, unlike as the for-statement in languages like Java we cannot generally predict how

many times while-statements will 'loop' around, or even whether they will terminate at all.

The key ingredient in the proof rule for Partial-while is the 'invariant' ψ. In general, the body C of the command `while (`B`)` $\{C\}$ changes the values of the variables it manipulates; but the invariant expresses a relationship between those values which is preserved by any execution of C. In the proof rule, ψ expresses this invariant; the rule's premise, $(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)$, states that, if ψ and B are true before we execute C, and C terminates, then ψ will be true after it. The conclusion of Partial-while states that, no matter how many times the body C is executed, if ψ is true initially and the while-statement terminates, then ψ will be true at the end. Moreover, since the while-statement has terminated, B will be false.

Implied. One final rule is required in our calculus: the rule Implied of Figure 4.1. It tells us that, if we have proved $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ and we have a formula ϕ' which implies ϕ and another one ψ' which is implied by ψ, then we should also be allowed to prove that $(\!|\phi'|\!)\ P\ (\!|\psi'|\!)$. A sequent $\vdash_{\mathrm{AR}} \phi \to \phi'$ is valid iff there is a proof of ϕ' in the natural deduction calculus for predicate logic, where ϕ and standard laws of arithmetic – e.g. $\forall x\,(x = x + 0)$ – are premises. Note that the rule Implied allows the precondition to be strengthened (thus, we *assume* more than we need to), while the postcondition is weakened (i.e. we *conclude* less than we are entitled to). If we tried to do it the other way around, weakening the precondition or strengthening the postcondition, then we would conclude things which are incorrect – see exercise 9(a) on page 300.

The rule Implied acts as a link between program logic and a suitable extension of predicate logic. It allows us to import proofs in predicate logic enlarged with the basic facts of arithmetic, which are required for reasoning about integer expressions, into the proofs in program logic.

4.3.2 Proof tableaux

The proof rules presented in Figure 4.1 are not in a form which is easy to use in examples. To illustrate this point, we present an example of a proof in Figure 4.2; it is a proof of the triple $(\!|\top|\!)$ `Fac1` $(\!|y = x!|\!)$ where `Fac1` is the factorial program given in Example 4.2. This proof abbreviates rule names; and drops the bars and names for Assignment as well as sequents for $\vdash_{\mathrm{AR}}$ in all applications of the Implied rule. We have not yet presented enough information for the reader to complete such a proof on her own, but she can at least use the proof rules in Figure 4.1 to check whether all rule instances of that proof are permissible, i.e. match the required pattern.

$$(\!|1 = 1|\!)\ \texttt{y = 1}\ (\!|y = 1|\!) \quad i$$

$$(\!|\top|\!)\ \texttt{y = 1}\ (\!|y = 1|\!)$$

$$(\!|y = 1 \wedge 0 = 0|\!)\ \texttt{z = 0}\ (\!|y = 1 \wedge z = 0|\!) \quad i$$

$$(\!|y = 1|\!)\ \texttt{z = 0}\ (\!|y = 1 \wedge z = 0|\!)$$

$$(\!|\top|\!)\ \texttt{y = 1; z = 0}\ (\!|y = 1 \wedge z = 0|\!) \quad c$$

$$(\!|y \cdot (z+1) = (z+1)!|\!)\ \texttt{z = z+1}\ (\!|y \cdot z = z!|\!) \quad i$$

$$(\!|y = z! \wedge z \neq x|\!)\ \texttt{z = z+1}\ (\!|y \cdot z = z!|\!)$$

$$(\!|y \cdot z = z!|\!)\ \texttt{y = y*z}\ (\!|y = z!|\!)$$

$$(\!|y = z! \wedge z \neq x|\!)\ \texttt{z = z+1; y = y*z}\ (\!|y = z!|\!) \quad c$$

$$(\!|y = z!|\!)\ \texttt{while (z != x) \{z = z+1; y = y*z\}}\ (\!|y = z! \wedge z = x|\!) \quad w$$

$$(\!|y = 1 \wedge z = 0|\!)\ \texttt{while (z != x) \{z = z+1; y = y*z\}}\ (\!|y = x!|\!) \quad i$$

$$(\!|\top|\!)\ \texttt{y = 1; z = 0; while (z != x) \{z = z+1; y = y*z\}}\ (\!|y = x!|\!) \quad c$$

Figure 4.2. A partial-correctness proof for `Fac1` in tree form.

It should be clear that proofs in this form are unwieldy to work with. They will tend to be very wide and a lot of information is copied from one line to the next. Proving properties of programs which are longer than `Fac1` would be very difficult in this style. In Chapters 1, 2 and 5 we abandon representation of proofs as trees for similar reasons. The rule for sequential composition suggests a more convenient way of presenting proofs in program logic, called *proof tableaux*. We can think of any program of our core programming language as a sequence

$$\begin{array}{l} C_1; \\ C_2; \\ \cdot \\ \cdot \\ \cdot \\ C_n \end{array}$$

where none of the commands C_i is a composition of smaller programs, i.e. all of the C_i above are either assignments, if-statements or while-statements. Of course, we allow the if-statements and while-statements to have embedded compositions.

Let P stand for the program $C_1; C_2; \ldots; C_{n-1}; C_n$. Suppose that we want to show the validity of $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\phi_n|\!)$ for a precondition ϕ_0 and a postcondition ϕ_n. Then, we may split this problem into smaller ones by trying to find formulas ϕ_j $(0 < j < n)$ and prove the validity of $\vdash_{\mathsf{par}} (\!|\phi_i|\!)\ C_{i+1}\ (\!|\phi_{i+1}|\!)$ for $i = 0, 1, \ldots, n-1$. This suggests that we should design a proof calculus which presents a proof of $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\psi_n|\!)$ by interleaving formulas with code as in

$$\begin{array}{lll}
 & (\!|\phi_0|\!) & \\
C_1\,; & & \\
 & (\!|\phi_1|\!) & \quad \texttt{justification} \\
C_2\,; & & \\
\cdot & & \\
\cdot & & \\
\cdot & & \\
 & (\!|\phi_{n-1}|\!) & \quad \texttt{justification} \\
C_n\,; & & \\
 & (\!|\phi_n|\!) & \quad \texttt{justification}
\end{array}$$

Against each formula, we write a justification, whose nature will be clarified shortly. Proof tableaux thus consist of the program code interleaved with formulas, which we call *midconditions*, that should hold at the point they are written.

Each of the transitions

$$\begin{array}{l} \quad (\!|\phi_i|\!) \\ C_{i+1} \\ \quad (\!|\phi_{i+1}|\!) \end{array}$$

will appeal to one of the rules of Figure 4.1, depending on whether C_{i+1} is an assignment, an if-statement or a while-statement. Note that this notation for proofs makes the proof rule for composition in Figure 4.1 implicit.

How should the intermediate formulas ϕ_i be found? In principle, it seems as though one could start from ϕ_0 and, using C_1, obtain ϕ_1 and continue working downwards. However, because the assignment rule works backwards, it turns out that it is more convenient to start with ϕ_n and work upwards, using C_n to obtain ϕ_{n-1} etc.

Definition 4.12 The process of obtaining ϕ_i from C_{i+1} and ϕ_{i+1} is called computing the weakest precondition of C_{i+1}, given the postcondition ϕ_{i+1}. That is to say, we are looking for the logically weakest formula whose truth at the beginning of the execution of C_{i+1} is enough to guarantee ϕ_{i+1}[4].

The construction of a proof tableau for $(\!|\phi|\!)\ C_1; \dots ; C_n\ (\!|\psi|\!)$ typically consists of starting with the postcondition ψ and pushing it upwards through C_n, then $C_{n-1}, \dots$, until a formula ϕ' emerges at the top. Ideally, the formula ϕ' represents the weakest precondition which guarantees that the ψ will hold if the composed program $C_1; C_2; \dots ; C_{n-1}; C_n$ is executed and terminates. The weakest precondition ϕ' is then checked to see whether it follows from the given precondition ϕ. Thus, we appeal to the Implied rule of Figure 4.1.

Before a discussion of how to find invariants for while-statement, we now look at the assignment and the if-statement to see how the weakest precondition is calculated for each one.

Assignment. The assignment axiom is easily adapted to work for proof tableaux. We write it thus:

[4] ϕ is weaker than ψ means that ϕ is implied by ψ in predicate logic enlarged with the basic facts about arithmetic: the sequent $\vdash_{\mathrm{AR}} \psi \to \phi$ is valid. We want the weakest formula, because we want to impose as few constraints as possible on the preceding code. In some cases, especially those involving while-statements, it might not be possible to extract the logically weakest formula. We just need one which is sufficiently weak to allow us to complete the proof at hand.

$$
\begin{array}{ll}
\quad (\!|\psi[E/x]|\!) & \\
\texttt{x = } E & \\
\quad (\!|\psi|\!) & \qquad\qquad \textsf{Assignment}
\end{array}
$$

The justification is written against the ψ, since, once the proof has been constructed, we want to read it in a forwards direction. The construction itself proceeds in a backwards direction, because that is the way the assignment axiom facilitates.

Implied. In tableau form, the Implied rule allows us to write one formula ϕ_2 directly underneath another one ϕ_1 with no code in between, provided that ϕ_1 implies ϕ_2 in that the sequent $\vdash_{\mathrm{AR}} \phi_1 \to \phi_2$ is valid. Thus, the Implied rule acts as an interface between predicate logic with arithmetic and program logic. This is a surprising and crucial insight. Our proof calculus for partial correctness is a hybrid system which interfaces with another proof calculus via the Implied proof rule *only*.

When we appeal to the Implied rule, we will usually not explicitly write out the proof of the implication in predicate logic, for this chapter focuses on the program logic. Mostly, the implications we typically encounter will be easy to verify.

The Implied rule is often used to simplify formulas that are generated by applications of the other rules. It is also used when the weakest precondition ϕ' emerges by pushing the postcondition upwards through the whole program. We use the Implied rule to show that the given precondition implies the weakest precondition. Let's look at some examples of this.

Examples 4.13

1. We show that $\vdash_{\textsf{par}} (\!|y = 5|\!)$ `x = y + 1` $(\!|x = 6|\!)$ is valid:

$$
\begin{array}{ll}
\quad (\!|y = 5|\!) & \\
\quad (\!|y + 1 = 6|\!) & \qquad \textsf{Implied} \\
\texttt{x = y + 1} & \\
\quad (\!|x = 6|\!) & \qquad \textsf{Assignment}
\end{array}
$$

   The proof is constructed from the bottom upwards. We start with $(\!|x = 6|\!)$ and, using the assignment axiom, we push it upwards through `x = y + 1`. This means substituting $y + 1$ for all occurrences of x, resulting in $(\!|y + 1 = 6|\!)$. Now, we compare this with the given precondition $(\!|y = 5|\!)$. The given precondition and the arithmetic fact $5 + 1 = 6$ imply it, so we have finished the proof.

Although the proof is constructed bottom-up, its justifications make sense when read top-down: the second line is implied by the first and the fourth follows from the second by the intervening assignment.

2. We prove the validity of $\vdash_{\mathsf{par}} (\!|y < 3|\!)$ `y = y + 1` $(\!|y < 4|\!)$:

$$
\begin{array}{ll}
(\!|y < 3|\!) & \\
(\!|y + 1 < 4|\!) & \text{Implied} \\
\texttt{y = y + 1;} & \\
(\!|y < 4|\!) & \text{Assignment}
\end{array}
$$

Notice that Implied always refers to the immediately preceding line. As already remarked, proofs in program logic generally combine two logical levels: the first level is directly concerned with proof rules for programming constructs such as the assignment statement; the second level is ordinary entailment familiar to us from Chapters 1 and 2 plus facts from arithmetic – here that $y < 3$ implies $y + 1 < 3 + 1 = 4$.

We may use ordinary logical and arithmetic implications to change a certain condition ϕ to any condition ϕ' which is implied by ϕ for reasons which have nothing to do with the given code. In the example above, ϕ was $y < 3$ and the implied formula ϕ' was then $y + 1 < 4$. The validity of $\vdash_{\mathrm{AR}} (y < 3) \to (y + 1 < 4)$ is rooted in general facts about integers and the relation $<$ defined on them. Completely formal proofs would require separate proofs attached to all instances of the rule Implied. As already said, we won't do that here as this chapter focuses on aspects of proofs which deal directly with code.

3. For the sequential composition of assignment statements

```
z = x;
z = z + y;
u = z;
```

our goal is to show that u stores the sum of x and y after this sequence of assignments terminates. Let us write P for the code above. Thus, we mean to prove $\vdash_{\mathsf{par}} (\!|\top|\!)\ P\ (\!|u = x + y|\!)$.

We construct the proof by starting with the postcondition $u = x + y$ and pushing it up through the assignments, in reverse order, using the assignment rule.

– Pushing it up through `u = z` involves replacing all occurrences of u by z, resulting in $z = x + y$. We thus have the proof fragment

$$
\begin{array}{ll}
(\!|z = x + y|\!) & \\
\texttt{u = z;} & \\
(\!|u = x + y|\!) & \text{Assignment}
\end{array}
$$

– Pushing $z = x + y$ upwards through `z = z + y` involves replacing z by $z + y$, resulting in $z + y = x + y$.

– Pushing that upwards through `z = x` involves replacing z by x, resulting in $x + y = x + y$. The proof fragment now looks like this:

```
    (|x + y = x + y|)
z = x;
    (|z + y = x + y|)        Assignment
z = z + y;
    (|z = x + y|)            Assignment
u = z;
    (|u = x + y|)            Assignment
```

The weakest precondition that thus emerges is $x + y = x + y$; we have to check that this follows from the given precondition $\top$. This means checking that any state that satisfies $\top$ also satisfies $x + y = x + y$. Well, $\top$ is satisfied in all states, but so is $x + y = x + y$, so the sequent $\vdash_{\mathrm{AR}} \top \to (x + y = x + y)$ is valid.
The final completed proof therefore looks like this:

```
    (|⊤|)
    (|x + y = x + y|)        Implied
z = x;
    (|z + y = x + y|)        Assignment
z = z + y;
    (|z = x + y|)            Assignment
u = z;
    (|u = x + y|)            Assignment
```

and we can now read it from the top down.

The application of the axiom Assignment requires some care. We describe two pitfalls which the unwary may fall into, if the rule is not applied correctly.

- Consider the example 'proof'

```
    (|x + 1 = x + 1|)
x = x + 1;
    (|x = x + 1|)            Assignment
```

which uses the rule for assignment incorrectly. Pattern matching with the assignment axiom means that ψ has to be $x = x + 1$, the expression E is $x + 1$ and $\psi[E/x]$ is $x + 1 = x + 1$. However, $\psi[E/x]$ is obtained by replacing *all* occurrences of x in ψ by E, thus, $\psi[E/x]$ would have to be equal to $x + 1 = x + 1 + 1$. Therefore, the corrected proof

$$\begin{array}{ll} & (\!|x+1 = x+1+1|\!) \\ \texttt{x = x + 1;} & \\ & (\!|x = x+1|\!) \qquad \text{Assignment} \end{array}$$

shows that $\vdash_{\text{par}} (\!|x+1 = x+1+1|\!)$ `x = x + 1` $(\!|x = x+1|\!)$ is valid.

As an aside, this corrected proof is not very useful. The triple says that, if $x+1 = (x+1)+1$ holds in a state and the assignment `x = x + 1` is executed and terminates, then the resulting state satisfies $x = x+1$; but, since the precondition $x+1 = x+1+1$ can never be true, this triple tells us nothing informative about the assignment.

- Another way of using the proof rule for assignment incorrectly is by allowing additional assignments to happen in between $\psi[E/x]$ and `x =` E, as in the 'proof'

$$\begin{array}{ll} & (\!|x+2 = y+1|\!) \\ \texttt{y = y + 1000001;} & \\ \texttt{x = x + 2;} & \\ & (\!|x = y+1|\!) \qquad \text{Assignment} \end{array}$$

This is not a correct application of the assignment rule, since an additional assignment happens in line 2 right before the actual assignment to which the inference in line 4 applies. This additional assignment makes this reasoning unsound: line 2 overwrites the current value in y to which the equation in line 1 is referring. Clearly, $x+2 = y+1$ won't be true any longer. Therefore, we are allowed to use the proof rule for assignment only if there is no additional code between the precondition $\psi[E/x]$ and the assignment `x =` E.

If-statements. We now consider how to push a postcondition upwards through an if-statement. Suppose we are given a condition ψ and a program fragment `if (`B`)` $\{C_1\}$ `else` $\{C_2\}$. We wish to calculate the weakest ϕ such that

$$(\!|\phi|\!)\ \texttt{if (}B\texttt{)}\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!).$$

This ϕ may be calculated as follows.

1. Push ψ upwards through C_1; let's call the result ϕ_1. (Note that, since C_1 may be a sequence of other commands, this will involve appealing to other rules. If C_1 contains another if-statement, then this step will involve a 'recursive call' to the rule for if-statements.)
2. Similarly, push ψ upwards through C_2; call the result ϕ_2.
3. Set ϕ to be $(B \to \phi_1) \wedge (\neg B \to \phi_2)$.

Example 4.14 Let us see this proof rule at work on the non-optimal code for `Succ` given earlier in the chapter. Here is the code again:

```
a = x + 1;
if (a - 1 == 0)  {
  y = 1;
} else {
  y = a;
}
```

We want to show that $\vdash_{\mathsf{par}} (\!|\top|\!)\, \texttt{Succ}\, (\!|y = x+1|\!)$ is valid. Note that this program is the sequential composition of an assignment and an if-statement. Thus, we need to obtain a suitable midcondition to put between the if-statement and the assignment.

We push the postcondition $y = x + 1$ upwards through the two branches of the if-statement, obtaining

- ϕ_1 is $1 = x + 1$;
- ϕ_2 is $a = x + 1$;

and obtain the midcondition $(a - 1 = 0 \to 1 = x + 1) \wedge (\neg(a - 1 = 0) \to a = x + 1)$ by appealing to a slightly different version of the rule If-statement:

$$\frac{(\!|\phi_1|\!)\; C_1\; (\!|\psi|\!) \qquad (\!|\phi_2|\!)\; C_2\; (\!|\psi|\!)}{(\!|(B \to \phi_1) \wedge (\neg B \to \phi_2)|\!)\; \texttt{if}\; B\; \{C_1\}\; \texttt{else}\; \{C_2\}\; (\!|\psi|\!)}\ \text{If-Statement} \tag{4.7}$$

However, this rule can be derived using the proof rules discussed so far; see exercise 9(c) on page 301. The partial proof now looks like this:

$(\!|\top|\!)$	
$(\!|?|\!)$	?
`a = x + 1;`	
$(\!|(a - 1 = 0 \to 1 = x + 1) \wedge (\neg(a - 1 = 0) \to a = x + 1)|\!)$	?
`if (a - 1 == 0) {`	
$(\!|1 = x + 1|\!)$	If-Statement
`y = 1;`	
$(\!|y = x + 1|\!)$	Assignment
`} else {`	
$(\!|a = x + 1|\!)$	If-Statement
`y = a;`	
$(\!|y = x + 1|\!)$	Assignment
`}`	
$(\!|y = x + 1|\!)$	If-Statement

Continuing this example, we push the long formula above the if-statement through the assignment, to obtain

$$(x + 1 - 1 = 0 \to 1 = x + 1) \wedge (\neg(x + 1 - 1 = 0) \to x + 1 = x + 1) \tag{4.8}$$

We need to show that this is implied by the given precondition $\top$, i.e. that it is true in any state. Indeed, simplifying (4.8) gives

$$(x = 0 \to 1 = x + 1) \land (\neg(x = 0) \to x + 1 = x + 1)$$

and both these conjuncts, and therefore their conjunction, are clearly valid implications. The above proof now is completed as:

```
   (|⊤|)
   (|(x + 1 − 1 = 0 → 1 = x + 1) ∧ (¬(x + 1 − 1 = 0) → x + 1 = x + 1)|)   Implied
a = x + 1;
   (|(a − 1 = 0 → 1 = x + 1) ∧ (¬(a − 1 = 0) → a = x + 1)|)               Assignment
if (a - 1 == 0) {
       (|1 = x + 1|)                                                      If-Statement
     y = 1;
       (|y = x + 1|)                                                      Assignment
} else {
       (|a = x + 1|)                                                      If-Statement
     y = a;
       (|y = x + 1|)                                                      Assignment
}
   (|y = x + 1|)                                                          If-Statement
```

While-statements. Recall that the proof rule for partial correctness of while-statements was presented in the following form in Figure 4.1 – here we have written η instead of ψ:

$$\frac{(\!|\eta \land B|\!)\ C\ (\!|\eta|\!)}{(\!|\eta|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\eta \land \neg B|\!)}\ \text{Partial-while}. \tag{4.9}$$

Before we look at how Partial-while will be represented in proof tableaux, let us look in more detail at the ideas behind this proof rule. The formula η is chosen to be an invariant of the body C of the while-statement: provided the boolean guard B is true, if η is true before we start C, and C terminates, then it is also true at the end. This is what the premise $(\!|\eta \land B|\!)\ C\ (\!|\eta|\!)$ expresses.

Now suppose the while-statement executes a terminating run from a state that satisfies η; and that the premise of (4.9) holds.

- If B is false as soon as we embark on the while-statement, then we do not execute C at all. Nothing has happened to change the truth value of η, so we end the while-statement with $\eta \land \neg B$.

- If B is true when we embark on the while-statement, we execute C. By the premise of the rule in (4.9), we know η is true at the end of C.
  - if B is now false, we stop with $\eta \wedge \neg B$.
  - if B is true, we execute C again; η is again re-established. No matter how many times we execute C in this way, η is re-established at the end of each execution of C. The while-statement terminates if, and only if, B is false after some finite (zero including) number of executions of C, in which case we have $\eta \wedge \neg B$.

This argument shows that Partial-while is sound with respect to the satisfaction relation for partial correctness, in the sense that anything we prove using it is indeed true. However, as it stands it allows us to prove only things of the form $(\!|\eta|\!)$ `while (`B`)` $\{C\}$ $(\!|\eta \wedge \neg B|\!)$, i.e. triples in which the postcondition is the same as the precondition conjoined with $\neg B$. Suppose that we are required to prove

$$(\!|\phi|\!)\ \texttt{while (}B\texttt{)}\ \{\texttt{C}\}\ (\!|\psi|\!) \tag{4.10}$$

for some ϕ and ψ which are not related in that way. How can we use Partial-while in a situation like this?

The answer is that we must *discover* a suitable η, such that

1. $\vdash_{\mathrm{AR}} \phi \to \eta$,
2. $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$ and
3. $\vdash_{\mathsf{par}} (\!|\eta|\!)$ `while (`B`)` $\{C\}$ $(\!|\eta \wedge \neg B|\!)$

are all valid, where the latter is shown by means of Partial-while. Then, Implied infers that (4.10) is a valid partial-correctness triple.

The crucial thing, then, is the discovery of a suitable invariant η. It is a necessary step in order to use the proof rule Partial-while and in general it requires intelligence and ingenuity. This contrasts markedly with the case of the proof rules for if-statements and assignments, which are purely mechanical in nature: their usage is just a matter of symbol-pushing and does not require any deeper insight.

Discovery of a suitable invariant requires careful thought about what the while-statement is really doing. Indeed the eminent computer scientist, the late E. Dijkstra, said that to understand a while-statement is tantamount to knowing what its invariant is with respect to given preconditions and postconditions for that while-statement.

This is because a suitable invariant can be interpreted as saying that the intended computation performed by the while-statement is correct up to the current step of the execution. It then follows that, when the execution

terminates, the entire computation is correct. Let us formalize invariants and then study how to discover them.

Definition 4.15 An invariant of the while-statement `while (`B`)` $\{C\}$ is a formula η such that $\models_{\mathsf{par}} (\!|\eta \wedge B|\!)\ C\ (\!|\eta|\!)$ holds; i.e. for all states l, if η and B are true in l and C is executed from state l and terminates, then η is again true in the resulting state.

Note that η does not have to be true continuously during the execution of C; in general, it will not be. All we require is that, if it is true before C is executed, then it is true (if and) when C terminates.

For any given while-statement there are several invariants. For example, $\top$ is an invariant for *any* while-statement; so is $\bot$, since the premise of the implication 'if $\bot \wedge B$ is true, then ...' is false, so that implication is true. The formula $\neg B$ is also an invariant of `while (`B`) do` $\{C\}$; but most of these invariants are useless to us, because we are looking for an invariant η for which the sequents $\vdash_{\mathrm{AR}} \phi \rightarrow \eta$ and $\vdash_{\mathrm{AR}} \eta \wedge \neg B \rightarrow \psi$, are valid, where ϕ and ψ are the preconditions and postconditions of the while-statement. Usually, this will single out just one of all the possible invariants – up to logical equivalence.

A useful invariant expresses a relationship between the variables manipulated by the body of the while-statement which is preserved by the execution of the body, even though the values of the variables themselves may change. The invariant can often be found by constructing a trace of the while-statement in action.

Example 4.16 Consider the program `Fac1` from page 262, annotated with location labels for our discussion:

```
      y = 1;
      z = 0;
l1:   while (z != x) {
          z = z + 1;
          y = y * z;
l2:   }
```

Suppose program execution begins in a store in which x equals 6. When the program flow first encounters the while-statement at location `l1`, z equals 0 and y equals 1, so the condition $z \neq x$ is true and the body is executed. Thereafter at location `l2`, z equals 1 and y equals 1 and the boolean guard is still true, so the body is executed again. Continuing in this way, we obtain