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

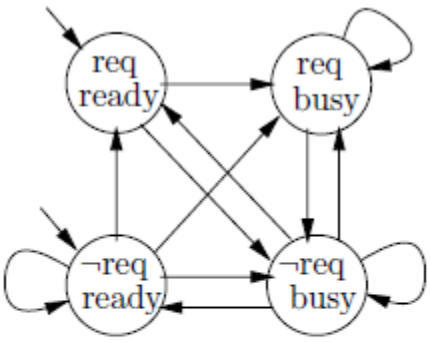

Figure 3.9. The model corresponding to the SMV program in the text.

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