Chapter 4

CTL Model Checking

The main purpose of a model checker is to verify that a model satisfies a set of desired properties specified by the user. In NuSMV, the specifications to be checked can be expressed in two different temporal logics: the Computation Tree Logic CTL, and the Linear Temporal Logic LTL extended with Past Operators. CTL and LTL specifications are evaluated by NuSMV in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, NuSMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property. In this section we will describe model checking of specifications expressed in CTL, while the next section we consider the case of LTL specifications.

4.1 Computation Tree Logic

CTL is a *branching-time* logic: its formulas allow for specifying properties that take into account the non-deterministic, branching evolution of a FSM. More precisely, the evolution of a FSM from a given state can be described as an infinite tree, where the nodes are the states of the FSM and the branching in due to the non-determinism in the transition relation. The paths in the tree that start in a given state are the possible alternative evolutions of the FSM from that state. In CTL one can express properties that should hold for *all the paths* that start in a state, as well as for properties that should hold just for *some of the paths*.

Consider for instance CTL formula `AF p`. It expresses the condition that, for *all* the paths (`A`) stating from a state, *eventually in the future* (`F`) condition `p` must hold. That is, all the possible evolutions of the system will eventually reach a state satisfying condition `p`. CTL formula `EF p`, on the other hand, requires than there *exists* some path (`E`) that eventually in the future satisfies `p`.

Similarly, formula `AG p` requires that condition `p` is always, or *globally*, true in all the states of all the possible paths, while formula `EG p` requires that there is some path along which condition `p` is continuously true.

Other CTL operators are:

- `A [p U q]` and `E [p U q]`, requiring condition `p` to be true *until* a state is reached that satisfies condition `q`;
- `AX p` and `EX p`, requiring that condition `p` is true in all or in some of the next states reachable from the current state.

CTL operators can be nested in an arbitrary way and can be combined using logic operators (`!`, `&`, `|`, `->`, `<->`, ...). Typical examples of CTL formulas are `AG ! p` ("condition `p` is absent in all the evolutions"), `AG EF p` ("it is always possible to reach a state where `p` holds"), and `AG (p -> AF q)` ("each occurrence of condition `p` is followed by an occurrence of condition `q`").

In NuSMV a CTL specification is given as CTL formula introduced by the keyword "SPEC". Whenever a CTL specification is processed, NuSMV checks whether the CTL formula is true in all the initial states of the model. If this is not a case, then NuSMV generates a counter-example, that is, a (finite or infinite) trace that exhibits a valid behavior of the model that does not satisfy the specification. Traces are very useful for identifying the error in the specification that leads to the wrong behavior. We remark that the generation of a counter-example trace is not always possible for CTL specifications. Temporal operators corresponding to existential path quantifiers cannot be proved false by a showing of a single execution path. Similarly, sub-formulas preceded by universal path quantifier cannot be proved true by a showing of a single execution path.

4.2 Semaphore Example

Consider the case of the semaphore program described in Chapter 2 [Examples], page 3. A desired property for this program is that it should never be the case that the two processes `proc1` and `proc2` are at the same time in the `critical` state (this is an example of a "safety" property). This property can be expressed by the following CTL formula:

```
AG ! (proc1.state = critical & proc2.state = critical)
```

Another desired property is that, if `proc1` wants to enter its critical state, it eventually does (this is an example of a "liveness" property). This property can be expressed by the following CTL formula:

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

In order to verify the two formulas on the semaphore model, we add the two corresponding CTL specification to the program, as follows:

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := FALSE;
 SPEC AG ! (proc1.state = critical & proc2.state = critical)
 SPEC AG (proc1.state = entering -> AF proc1.state = critical)
MODULE user(semaphore)
 VAR
   state : {idle, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                  : {idle, entering};
       state = entering & !semaphore : critical;
       state = critical              : {critical, exiting};
       state = exiting               : idle;
       TRUE                          : state;
     esac;
   next(semaphore) :=
     case
       state = entering : TRUE;
       state = exiting  : FALSE;
```

```
      TRUE             : semaphore;
    esac;
FAIRNESS
  running
```

By running NuSMV with the command

```
system_prompt> NuSMV semaphore.smv
```

we obtain the following output:

```
-- specification AG (!(proc1.state = critical & proc2.state = critical))
-- is true
-- specification AG (proc1.state = entering -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following execution sequence
-> State: 1.1 <-
    semaphore = FALSE
    proc1.state = idle
    proc2.state = idle
-> Input: 1.2 <-
    _process_selector_ = proc1
-- Loop starts here
-> State: 1.2 <-
    proc1.state = entering
-> Input: 1.3 <-
    _process_selector_ = proc2
-> State: 1.3 <-
    proc2.state = entering
-> Input: 1.4 <-
    _process_selector_ = proc2
-> State: 1.4 <-
    semaphore = FALSE
    proc2.state = critical
-> Input: 1.5 <-
    _process_selector_ = proc1
-> State: 1.5 <-
-> Input: 1.6 <-
    _process_selector_ = proc2
-> State 1.6 <-
    proc2.state = exiting
-> Input: 1.7 <-
    _process_selector_ = proc2
-> State 1.7 <-
    semaphore = FALSE
    proc2.state = idle
```

Note that `_process_selector_` is a special variable which contains the name of the process that will execute to cause a transition to the next state. The '`Input`' section displays the values of variables that the model has no control over, that is it cannot change their value. Since processes are chosen nondeterministically in this model, it has no control over the value of `_process_selector_`.
NuSMV tells us that the first CTL specification is true: it is never the case that the two processes will be at the same time in the critical region. On the other hand, the second specification is false. NuSMV produces a counter-example path where initially `proc1` goes to state `entering` (state 1.2), and then a loop starts in which `proc2` repeatedly enters its

critical region (state 1.4) and then returns to its `idle` state (state 1.7); in the loop, `proc1` is activated only when `proc2` is in the critical region (input 1.5), and is therefore not able to enter its critical region (state 1.5). This path not only shows that the specification is false, it also points out why can it happen that `proc1` never enters its critical region.

Note that in the printout of a cyclic, infinite counter-example the starting point of the loop is marked by `-- loop starts here`. Moreover, in order to make it easier to follow the action in systems with a large number of variables, only the values of variables that have changed in the last step are printed in the states of the trace.