Chapter 2

Examples

In this section we describe the input language of NuSMV by presenting some examples of NuSMV models. A complete description of the NuSMV language can be found in the NuSMV 2.6 User Manual. Also, all mentioned example files can be found in the distributed archive of NuSMV 2.6, or can be individually downloaded from the NuSMV web pages, at the URL <http://nusmv.fbk.eu/examples/examples.html>.

The input language of NuSMV is designed to allow for the description of Finite State Machines (FSMs from now on) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can specify a system as a synchronous Mealy machine, or as an asynchronous network of nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones – booleans, scalars and fixed arrays. Static data types can also be constructed.

The primary purpose of the NuSMV input is to describe the transition relation of the FSM; this relation describes the valid evolutions of the state of the FSM. In general, any propositional expression in the propositional calculus can be used to define the transition relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The NuSMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in NuSMV is similar to that of single assignment data flow language. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a description language, or a programming language.

2.1 Synchronous Systems

2.1.1 Single Process Example

Consider the following simple program in the NuSMV language:

```
MODULE main
VAR
  request : boolean;
  state   : {ready, busy};
```

```
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request = TRUE : busy;
                   TRUE                           : {ready, busy};
                 esac;
```

The space of states of the FSM is determined by the declarations of the state variables (in the above example `request` and `state`). The variable `request` is declared to be of (predefined) type `boolean`. This means that it can assume the (boolean) values `FALSE` and `TRUE`. The variable `state` is a scalar variable, which can take the symbolic values `ready` or `busy`. The following assignment sets the initial value of the variable `state` to `ready`. The initial value of `request` is completely unspecified, i.e. it can be either `FALSE` or `TRUE`. The transition relation of the FSM is expressed by defining the value of variables in the next state (i.e. after each transition), given the value of variables in the current states (i.e. before the transition). The `case` segment sets the next value of the variable `state` to the value `busy` (after the colon) if its current value is `ready` and `request` is `TRUE`. Otherwise (the `TRUE` before the colon) the next value for `state` can be any in the set {`ready, busy`}. The variable `request` is not assigned. This means that there are no constraints on its values, and thus it can assume any value. `request` is thus an unconstrained input to the system.

2.1.2 Binary Counter

The following program illustrates the definition of reusable modules and expressions. It is a model of a three bit binary counter circuit. The order of module definitions in the input file is not relevant.

```
MODULE counter_cell(carry_in)
 VAR
   value : boolean;
 ASSIGN
   init(value) := FALSE;
   next(value) := value xor carry_in;
 DEFINE
  carry_out := value & carry_in;
MODULE main
 VAR
   bit0 : counter_cell(TRUE);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
```

The FSM is defined by instantiating three times the module type `counter_cell` in the module `main`, with the names `bit0`, `bit1` and `bit2` respectively. The `counter_cell` module has one formal parameter `carry_in`. In the instance `bit0`, this parameter is given the actual value `TRUE`. In the instance `bit1`, `carry_in` is given the value of the expression `bit0.carry_out`. This expression is evaluated in the context of the `main` module. However, an expression of the form '$a.b$' denotes component 'b' of module 'a', just as if the module 'a' were a data structure in a standard programming language. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`.

The keyword 'DEFINE' is used to assign the expression `value & carry_in` to the symbol `carry_out`. A definition can be thought of as a variable with value (functionally) depending on the current values of other variables. The same effect could have been obtained as follows (notice that the *current* value of the variable is assigned, rather than the *next* value.):

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

Defined symbols do not require introducing a new variable, and hence do not increase the state space of the FSM. On the other hand, it is not possible to assign to a defined symbol a value non-deterministically. Another difference between defined symbols and variables is that while the type of variables is declared a priori, for definitions this is not the case.

2.2 Asynchronous Systems

Important!
Since NuSMV version 2.5.0 processes are *deprecated*. In future versions of NuSMV processes may be no longer supported, and only synchronous systems will be supported by the input language. Modeling of asynchronous systems will have to be resolved at higher level.

The previous examples describe synchronous systems, where the assignments statements are taken into account in parallel and simultaneously. NuSMV allows to model asynchronous systems. It is possible to define a collection of parallel processes, whose actions are interleaved, following an asynchronous model of concurrency. This is useful for describing communication protocols, or asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock region).

2.2.1 Inverter Ring

The following program represents a ring of three asynchronous inverting gates.

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
   init(output) := FALSE;
   next(output) := !input;
MODULE main
 VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);
```

Among all the modules instantiated with the `process` keyword, one is nondeterministically chosen, and the assignment statements declared in that process are executed in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates.

We remark that the system is not forced to eventually choose a given process to execute. As a consequence the output of a given gate may remain constant, regardless of its input. In order to force a given process to execute infinitely often, we can use a fairness constraint. A fairness constraint restricts the attention of the model checker to only those execution paths along which a given formula is true infinitely often. Each process has a special variable called `running` which is `TRUE` if and only if that process is currently executing.

By adding the declaration:

```
FAIRNESS
  running
```

to the module `inverter`, we can effectively force every instance of `inverter` to execute infinitely often.
An alternative to using processes to model an asynchronous circuit is to allow all gates to execute simultaneously, but to allow each gate to choose non-deterministically to re-evaluate its output or to keep the same output value. Such a model of the inverter ring would look like the following:

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
  init(output) := FALSE;
  next(output) := (!input) union output;
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate1.output);
   gate3 : inverter(gate2.output);
```

The `union` operator (set union) coerces its arguments to singleton sets as necessary. Thus, the next `output` of each gate can be either its current `output`, or the negation of its current `input` – each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as 2^n, where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation. We remark that in this case we cannot force the inverters to be effectively active infinitely often using a fairness declaration. In fact, a valid scenario for the synchronous model is the one where all the inverters are idle and assign to the next `output` the current value of `output`.

2.2.2 Mutual Exclusion

The following program is another example of asynchronous model. It uses a variable semaphore to implement mutual exclusion between two asynchronous processes. Each process has four states: `idle`, `entering`, `critical` and `exiting`. The `entering` state indicates that the process wants to enter its critical region. If the variable `semaphore` is `FALSE`, it goes to the `critical` state, and sets `semaphore` to `TRUE`. On exiting its critical region, the process sets `semaphore` to `FALSE` again.

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := FALSE;
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
```

2.3 Direct Specification

NuSMV allows to specify the FSM directly in terms of propositional formulas. The set of possible initial states is specified as a formula in the current state variables. A state is initial if it satisfies the formula. The transition relation is directly specified as a propositional formula in terms of the *current* and *next* values of the state variables. Any current state/next state pair is in the transition relation if and only if it satisfies the formula.

These two functions are accomplished by the 'INIT' and 'TRANS' keywords. As an example, here is a description of the three inverter ring using only TRANS and INIT:

```
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate1.output);
   gate3 : inverter(gate2.output);
MODULE inverter(input)
 VAR
   output : boolean;
 INIT
   output = FALSE
 TRANS
   next(output) = !input | next(output) = output
```

According to the TRANS declaration, for each inverter, the next value of the output is equal either to the negation of the input, or to the current value of the output. Thus, in effect, each gate can choose non-deterministically whether or not to delay.

Using TRANS and INIT it is possible to specify inadmissible FSMs, where the set of initial states is empty or the transition relation is not total. This may result in logical absurdities.