Formal Methods in Software Development
Resume of the 02/12/2020 and 09/12/2020 lessons

Igor Melatti

1 Symbolic Model Checkers: NuSMV

- SMV (Symbolic Model Verifier): McMillan implementation of the ideas in the famous paper "Symbolic model checking: 10^{20} states and beyond"
  - note that McMillan PhD dissertation, which describes SMV and how it works, is one of the most important dissertations in Computer Science
- SMV has been then re-written and standardized by the research group in Trento (also Genova and CMU collaborated), thus becoming NuSMV
  - the engine is still McMillan's work
  - code has been nearly entirely commented, and made more readable
  - some features has been added: interactive mode, bounded model checking
  - OBDDs are handled via the CUDD library (by F. Somenzi at Colorado University)
- First of all, we again start with the *input language*, which does not have a name
- To better illustrate such an input language, we begin with a small example

```
MODULE main
VAR
  request : {Tr, Fa}; -- same as saying boolean (stand for True and False)
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   1 : {ready,busy};
```

```
                esac;
SPEC
  AG((request = Tr) -> AF state = busy)
```

- taken from `examples/smv-dist/short.smv`
- one *module*, there may be more, but one of them must be named `main`
- module variables are those declared with `VAR`
- base types are like Murphi ones: enumerations and integer subranges, plus the `word` type (i.e., an array of bits)
- arrays are possible, but can be indexed only with constants
- structures are modeled through modules
  * that is, each module has its variables (fields of a structure) and may be instantiated many times
- `ASSIGN` section specifies (indirectly; it is also possible to it directly, as we will see) the set S_0 (via `init`) and the relation R (via `next`)
  * as in Murphi, there expressions which are essentially guard/action
  * differently from Murphi, each action deals with *one variable only*
    · the guard may be defined on any other variable (and it is typically the case)
  * if something is not specified, then it is understood to be non-deterministic
  * e.g., in `short.smv` initial states are those in which `state` is `ready` and `request` may be either `Tr` or `Fa`
  * thus, there are 2 initial states $I = \{\langle \texttt{ready}, \texttt{Tr} \rangle, \langle \texttt{ready}, \texttt{Fa} \rangle\}$, which may be represented with $\langle \texttt{ready}, \perp \rangle$
  * also `next(request)` is not specified; before analyzing what does this mean, let us see `next(state)`
  * the `case` expression works as follows: the first condition C which is evaluated to true is fired, other true guards possibly following C are ignored
  * this allows to put 1 (i.e., true) as the last guard, representing the "default" case
  * NuSMV also checks if a `case` expression is exhaustive in its conditions, as this allows it to assume that T is total
  * note that the last condition on `state` leads to a non-deterministic transition: if the first guard is false, then `state` may take any value between `ready` e `busy`, that is any value in its domain
  * in general, any subset of the variabe domain may be used
  * `request` is completely non-deterministic, as it does not occur in any `next`

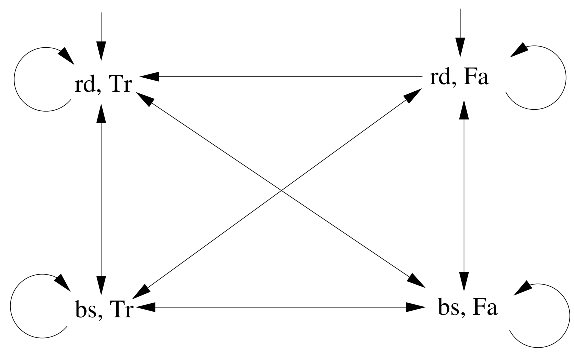

Figure 1: `short.smv`: R and S_0

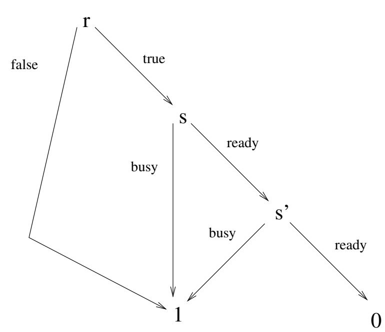

Figure 2: `short.smv`: OBDD for R; variables are only shown with their first letter

  * i.e., if other rules tells that the system may go from s to t and $(\texttt{request} = \texttt{Fa}) \in L(t)$, then there exists a transition from s to t' with $(\texttt{request} = \texttt{Tr}) \in L(t')$ and $L(t) \setminus \{(\texttt{request} = \texttt{Fa})\} = L(t') \setminus \{(\texttt{request} = \texttt{Tr})\}$
  * simply stated, if the system may go from s to t and `request` has a value v in t, then the system may also go from s to t' s.t. t and t' only differ in the value of `request`, which is different from v
  * by combining all non-determinism in this example, the Kripke structure defined here excludes just one transition: see Figure 1
  * OBDD for this example are in Figures 2, 3, 4 and 5

- Examples from `NuSMV.tutorial.pdf`: binary counter (see Figure 6)
  - 2 modules, `main` and `counter_cell`
  - main *instantiates* the module `counter_cell` for 3 times

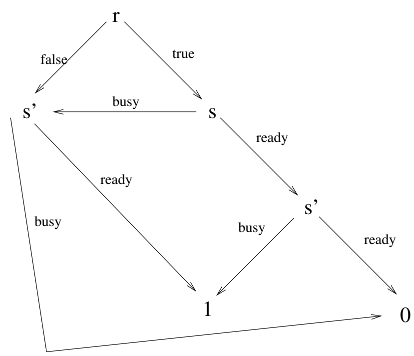

Figure 3: `short.soloready.smv`: OBDD for R

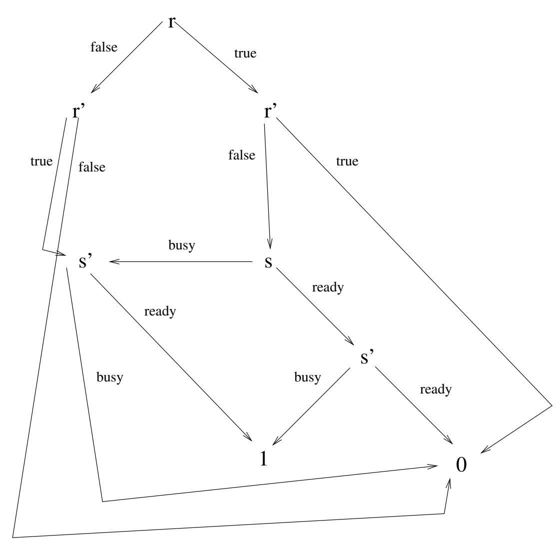

Figure 4: `short.soloready.req_const.smv`: OBDD for R

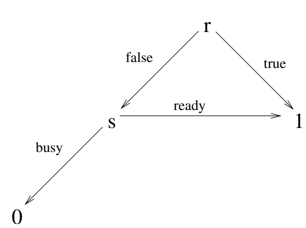

Figure 5: `short.soloready.req_const.smv`: OBDD for reachable states

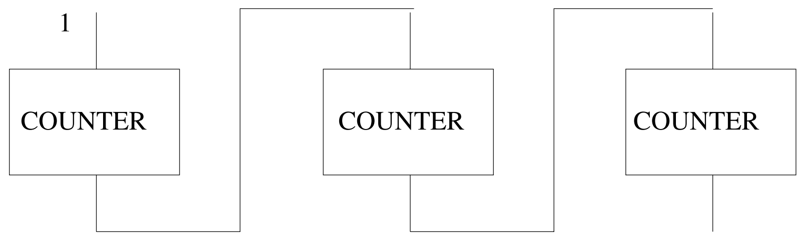

Figure 6: `counter.smv`

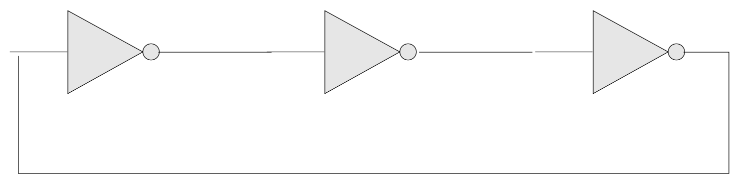

Figure 7: `inverter.smv`

- this is an hardware-like instantiation: the `main` module contains 3 equal copies of the `counter_cell` module, the only difference being the lines in input
- note that this means the module `main` will have 3 copies of variable `value`
- note that `carry_out` (being inside a `DEFINE` section) is *not* a variable, as it is only a shortcut for the expression it defines
  * i.e., there will not be a corresponding variable in the OBDD
  * and indeed, it is not declared as a variable...
- hence, `bit0` will always sum 1 to its internal variable, and `bit1` will sum 1 only if `bit0` will generate a carry
- the `main` module defines a counter from 0 to 7

• Examples in `NuSMV.tutorial.pdf`: inverter ring (see Figure 7)

- in the previous examples, all variables were evolving at the same time
- there is a global clock as in a synchronous digital circuit: given the current value for all variables in the current clock tick, in the next clock tick all variables may change their variables at the same time (synchronously: hardware parallel execution)
- in this example, instead, instantations are *processes*
- i.e., just one variable at a time may change; other variables are forced to stay fixed

  - no dynamic process spawning as in SPIN: the number of processes is known from the beginning
  - *synchronous* vs. *asynchronous* systems
  - in asynchronous systems, there is essentially one (implicit) additional module, which acts as a scheduler
  - this is indeed what the verification algorithm does
  - each process is automatically provided with an additional variable `running` which is true iff that process is currently running

- Examples in `NuSMV.tutorial.pdf`: mutual exclusion and direct specification

  - with direct specification it is possible to define non-total transition relations or empty initial states set

- NuSMV is provided with an interactive shell, as there are many tasks it may accomplish (simulation, many verification options); see user maual from chapter 3, especially Figure 3.1 at page 87

- NuSMV verification algorithm: it is exactly the fixed point computation

  - here we cover some interesting details about some preprocessing that NuSMV needs before starting the verification algorithm
  - executing a non-interactive verification in NuSMV is the same as giving the following list of interactive commands:

    read_model it reads and stores the syntactic structure of the input model

      * no OBDDs here: tree-like structure, but representing the syntactic structure of the input (abstract syntax tree)

    flatten_hierarchy (recursively) bring inside `main` all modules instantiated by `main`

      * very similar to the unfolding we mentioned for Murphi and SPIN: for such explicit model checkers, this was only needed for theoretical purposes, in order to define the Kriepke structure of an input model
      * here, it must be actually performed in the source code of NuSMV, in order to then be able to encode R and S_0 as OBDDs
      * to this aim, there must be only one module, the `main`, containing all variables coming from the modules it instantiates (to be applied recursively)
      * note that, again, this resembles digital circuits, where such a flattening is a natural operation

    * this could entail adding a scheduler module if processes are used

encode_variables for each variable x with domain D s.t. $|D| > 2$, NuSMV defines $x_1 \ldots, x_m$ *boolean* variables with $m = \lfloor \log_2 |D| \rfloor + 1$; it also defines the encoding for constants used in the input models

build_flat_model combines the result of the preceding operations to obtain the flattenized and boolenized syntactic structure which represents the Kriepke structure defined by the input model

build_model from the syntactic structure to OBDDs for R ed S_0 (plus other ones)

check_ctlspec (or `check_ltlspec`, or both, depending on what you have to verify); it starts the actual verification

    * generic algorithm for CTL model checking, based on the property structure
    * however, for $\mathbf{AG}p$ (i.e., safety properties) the fixed point algorithm computing the reachable states set is used, without using $\neg\mathbf{EG}\neg p$
    * that is, first the least fixpoint for $\mu Z \lambda x.I(x) \vee \exists y(Z(y)R(y,x))$ is computed (*reachable states set*)
    * then it is checked if $\exists x.Z(x) \wedge \neg p(x)$; if it is the case, a failing reachable state has been found
    * note that printing the counterexample is not trivial as it somewhat is in Murphi and SPIN (where the DFS stack already holds the counterexample): another OBDD-based computation is needed

  – differently from explicit model checkers, no need to give separate commands to generate a file to be compiled and executed: all is represented as OBDDs, you only have to use them properly

- From a NuSMV model $\mathcal{M}$ (defined with the `ASSIGN` section) to the corresponding Kriepke structure $M = (S, S_0, R, L)$

  – $V = \langle v_1, \ldots, v_n \rangle$ is the set of variables defined inside the `main` module of $\mathcal{M}$, with domains $\langle D_1, \ldots, D_n \rangle$

    * note that each D_i may be the instantiation of other modules
    * in which case, again, all variables must be considered as *unfolded*
    * that is, if a variable v is the instantiation of a module with k variables, then v counts as k variables instead of one
    * if one of such k variables is another instantiation, this procedure must be recursively repeated
    * NuSMV calls this operation *hierarchy flattening*
    * essentially, it is the same as for records in Murphi

    * simple types are the recursion base step
  - $S = D_1 \times \ldots \times D_n$ (as in Murphi)
  - S_0 is defined by looking at `init` predicates
    * $s \in S_0$ iff, for all variables $v \in V$, $s(v) \in \texttt{init}(v)$
      · note that, by NuSMV syntax, each $\texttt{init}(v)$ is actually a set (possibly a singleton)
    * if $\texttt{init}(v)$ is not specified in $\mathcal{M}$, then any value for v is ok: in this case, formally, if $s \in S_0$, then also $s' \in S_0$ being $s'(v') = s(v') \forall v' \neq v$
  - R is defined by looking at `next` predicates
    * we assume all `next` predicates to be defined by the `case` construct (if not, simply assume it is the `case` construct with just one `TRUE` condition)
    * for each (flattened) variable v, we name $g_1(v), \ldots g_{k_v}(v)$ the conditions (guards) of the `case` for $\texttt{next}(v)$, and $a_1(v), \ldots a_{k_v}(v)$ the resulting values (actions) of the `case` for $\texttt{next}(v)$
    * note that, by NuSMV syntax, each $a_i(v)$ is actually a set (possibly a singleton)
    * $(s, s') \in R$ iff, for all variables $v \in V$, if $g_i(s(v)) \wedge \forall j < i \neg g_j(s(v))$ then $s'(v) \in a_i(v)$
    * that is, s may go in s' iff, for all variables v, if the values of v in s satisfy the guard g_i (and none of the preceding guards for the same variable), then the value of v in s' is one of the values specified by the `case` for guard g_i
    * note that, in doing this, you also have to resolve inputs for modules
    * e.g., in the example with inverters (pag. 7 of the NuSMV tutorial), in defining `next(gate1.output)`, `gate1.input` must be relaced with `gate3.output`

- $AP = \{(v = d) \mid v = v_i \in V \wedge d \in D_i\}$
- $(v = d) \in L(s)$ iff variable v has value d in s
- If, instead, the NuSMV model $\mathcal{M}$ is defined with the `TRANS` section, then
  - $V = \langle v_1, \ldots, v_n \rangle$ is the set of variables as above and $S = D_1 \times \ldots \times D_n$
  - S_0 is defined by looking at `INIT` section
    * $s \in S_0$ iff, for all variables $v \in V$ and for all `INIT` sections I, $I(s(v))$ holds
  - R is defined by looking at `TRANS` section
    * $(s, s') \in R$ iff, for all variables $v \in V$ and `TRANS` sections T, $T(s(v), s'(v))$ holds