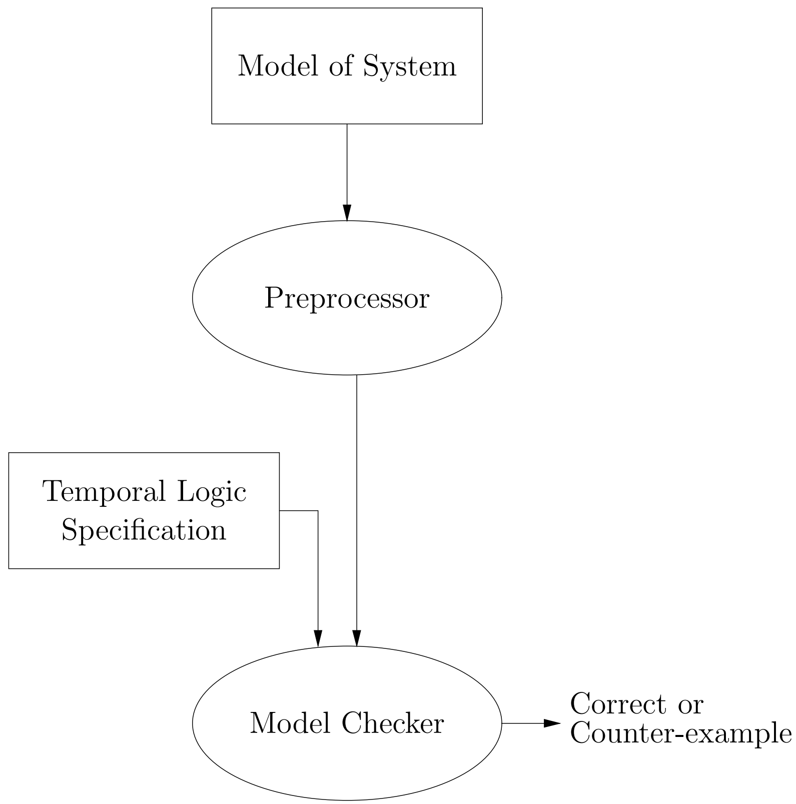
NuSMV: A Symbolic Model Checker

NuSMV: Overview



Fair Transition Systems: Recall

A fair transition system is $\Phi = (\mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$ (representing a reactive system) where

- $\mathcal{V}$ (vocabulary) is a finite set of typed variables
- Θ is an assertion characterizing the initial condition
- $\mathcal{T}$ is the set of transitions
- $\mathcal{J}$ describe the justice (or weak fairness) constraints
- $\mathcal{C}$ describe the compassion (or strong fairness) constraints

We will assume that each variable has a finite domain; thus the total number of "states" is finite.

NuSMV Description Language

The tool NuSMV comes with a description language that is used to describe the (finite) fair transition system modelling the program

- The description is broken down into modules that can be composed and reused.
- Modules describe initial values of variables and how they change in each step.
- Fairness conditions are also described in modules
- Has primitives to describe synchronous and asynchronous concurrent computations

An Example Program

```
MODULE main
  VAR
    request : boolean;
    state   : {ready, busy};
  ASSIGN
    init(state) := ready;
    next(state) := case
                     state = ready & request = 1 : busy;
                     1                           : {ready, busy};
                   esac;
```

- Variables can have `boolean` type, enumerated type, finite range of integers given by `<number> .. <number>`, or finite arrays
- `booleans` are `1` and `0`
- Keyword `init` is used to describe the initial value; unspecified variables can take any value in their type as the initial value
- Keyword `next` describes how the value of the variable changes in one step; again if the next value is unspecified, then the variable takes any value in its type at the next step
- `case` statement assigns the value associated with the first case condition that true right now; `1` is the default case

Reusing Modules: 3-bit counter

```
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;

MODULE main
  VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
```

- Modules can have parameters; they can be used to establish identities or connections. So `carry_in` variable of `bit1` is identified with `carry_out` of `bit0`
- `DEFINE` is used to define C-like "macros"; defined variables are not real variables in that they do not increase the state space.

Asynchronous Concurrent Components: 3-inverter gates

```
MODULE inverter(input)
  VAR
    output : boolean;
  ASSIGN
    init(output) := 0;
    next(output) := !input;

MODULE main
  VAR
    gate1 : process inverter(gate3.output);
    gate2 : process inverter(gate1.output);
    gate3 : process inverter(gate2.output);
```

- Unlike previous examples, where every variable was updated in each step synchronously, the keyword `process` can be used to describe asynchronous concurrently executing component
- At each time step one process is chosen nondeterministically, and all assignment statements in that process module is executed
- Variables of processes not assigned remain unchanged

Fairness (Justice) Conditions

For a system with asynchronous processes, it is not required that each component be eventually executed; this is ensured through fairness conditions.

Fairness/Justice: The keyword `FAIRNESS` (or `JUSTICE`) must be followed by a boolean conditions.

- A fair computation is one where the boolean condition is true infinitely often
- Every asynchronous process has a special variable `running` which is `1` exactly at the times when it is executing
- Thus, in order to ensure that an asynchronous process is eventually executed you add the following condition to the module

```
FAIRNESS
  running
```

Another Example: Mutual Exclusion

```
MODULE main
  VAR
    semaphore : boolean;
    proc1     : process user(semaphore);
    proc2     : process user(semaphore);
  ASSIGN
    init(semaphore) := 0;

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                 : state;
        1
      esac;
```

```
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting  : 0;
      1                : semaphore;
    esac;
FAIRNESS
  running
```

Asynchrony via Nondeterminism: 3-inverter gates

Another way to model asynchronous processes is to execute each process simultaneously, but allow a process to choose non-deterministically to either compute a new value or retain its old values

```
MODULE main
  VAR
    gate1 : inverter(gate3.output);
    gate2 : inverter(gate1.output);
    gate3 : inverter(gate2.output);

MODULE inverter(input)
  VAR
    output : boolean;
  ASSIGN
    init(output) := 0;
    next(output) := !input union output;
```

- The `union` operator forces its arguments to be singleton sets

Using Propositional Formulas

Instead of assigning values to variables, initial conditions and the transition relation can be described using propositional formulas

```
MODULE main
  VAR
    gate1 : inverter(gate3.output);
    gate2 : inverter(gate1.output);
    gate3 : inverter(gate2.output);

MODULE inverter(input)
  VAR
    output : boolean;
  INIT
    output = 0;
  TRANS
    next(output) = !input | next(output) = output
```

Keyword `INIT` describes the initial condition, and `TRANS` describes the transtion relation.

Specifications

- Each module can have requirements described in temporal logic
- Requirements either described in LTL or in Computation Tree Logic (CTL)
  - CTL specifications are described using the keyword `SPEC`
  - LTL specifications are described using the keyword `LTLSPEC`
  - In LTL the logical connectives are as follows: `X` (neXt), `G` (Globally or always), `F` (eventually in the Future), `U` (until), `V` (releases), plus past time operators

NuSMV in Action

NuSMV can be used either interactively or in batch mode to

- Automatically (model) check system with respect to requirements
- Simulate the system step-by-step interactively, randomly or deterministically.
- Bounded Model Check the specification

Model Checking in NuSMV

- Write the specification and system description in a file with `.smv` extension
- Type `NuSMV <file>.smv`
- NuSMV will check each specification automatically, informing whether it is satisfied or produce a trace (when possible) to demonstrate its violation.
- There are also commands to check properties interactively (see User Manual).

Simulating a Model

- Start NuSMV to execute interactively by typing `NuSMV -int <file>.smv`
- Type `go` to start simulation
- All traces simulated during one session are numbered and can be printed using `show_traces <traceid>`
- Each state in a trace is numbered sequentially; the first state is numbered `1`. The state is identified as `<traceid>.<stateid>`

Simulating a Model (contd)

- A new trace is begun when you pick the starting state
  - `pick_state -r` picks a state randomly from the initial states
  - `pick_state -i` picks a state interactively from the initial states. The user is prompted with a list of choices
  - `pick_state -c ''<const>'' -i` picks states interactively which satisfy the constraint `<const>`
  - `goto_state <traceid>.<stateid>` chooses the starting state to be some state in a previously simulated trace
- Subsequent states in the simulation can be picked using `simulate -<ops> <num>` which simulates for `<num>` steps according to `<ops>`
  - `r` does random simulation
  - `i` does interactive simulation
  - `c` picks only states that satisfy the constraint specified

Bounded Model Checking

Bounded Model Checking is used to check if LTL specifications are satisfied in traces whose length is bounded by some number

- Type `NuSMV -bmc <file>.smv`
- Will check progressively whether all traces of length $1, 2, \ldots 10$ satisfy the LTL requirements.
- Bound on the maximum trace length can be set by `-bmc_length` command line option; default setting is 10.

Compassion Constraints in NuSMV

Compassion constraints in NuSMV models is given by

```
COMPASSION
  (p, q)
```

A computation satisfies such a constraint when it satisfies the following: if `p` holds infinitely often then `q` holds infinitely often in the computation.

- A fair computation is one which satisfies both the compassion and justice constraints of the module.