COMP6210 Automated Software Verification

The NuSMV Model Checker

Pavel Naumov

Intended Learning Outcomes

By the end of this lecture, you will be able to

- write simple NuSMV models
- understand how to encode a program as a NuSMV model

NuSMV

- NuSMV is a symbolic model checker developed by ITC-IRST and Univ. Trento with the collaboration of CMU and Univ. Genova.
  - project url: http://nusmv.fbk.eu/
- it supports the modelling of both synchronous and asynchronous systems
  - allows for modular construction of models
- it supports the verification of properties expressed in LTL (Linear Temporal Logic) and CTL (Computation Tree Logic)
- it also supports bounded model checking

A First SMV Model

```
MODULE  main
VAR
   b0 : boolean
ASSIGN
   init(b0) :=  0;
   next(b0) :=  !b0;
```

An SMV model consists of:

- declarations of state variables (`b0` in the example); these determine the state space of the model.
- assignments that constrain the valid initial states

  `(init(b0) := 0)`
- assignments that constrain the transition relation

  `(next(b0) := !b0)`

SMV data types include:

- **boolean**:

```
VAR
  x :  boolean;
```

- **enumeration**:

```
VAR
  st :  {ready, busy, waiting, stopped};
```

- **bounded integer** (interval):

```
VAR
  n :  1..8;
```

- **array**:

索引 1-10

取值范围

```
VAR
  a :  array 1..10 of {red, green, blue};
```

Assignments

- initialisation:

```
ASSIGN
  init(x) := expression ;
```

  - If no `init()` assignment is specified for a variable, then it is initialised non-deterministically.

- progression:

```
ASSIGN
  next(x) := expression ;
```

  - If no `next()` assignment is specified for a variable, then it evolves nondeterministically, i.e. it is unconstrained.
  - Unconstrained variables can be used to model nondeterministic inputs to the system.

Assignments (Cont'd)

- immediate:

  ```
  ASSIGN
    y := expression ;
  ```

  or

  ```
  DEFINE
    y := expression ;
  ```

  - Immediate assignments constrain the current value of a variable in terms of the current values of other variables.
  - Immediate assignments can be used to model outputs of the system.

Expressions (1)

算术

- arithmetic operators:

  `+ − * / mod` (unary)

- comparison operators:

  `= != > < <= >=`

- logic operators:

  `&` `|` `xor` `!` (not) `->` `<->`

- set operators: `{v1,v2,...,vn}`
  - `in` : tests a value for membership in a set (set inclusion)
  - `union` : takes the union of 2 sets (set union)

- count operator: counts number of true boolean expressions

  `count(b1 + b2 + ... + bn)`

Expressions (2)

- case expression:

```
case
  c1 :  e1;
  c2 :  e2;
  ...
  TRUE : default;
esac
```

  - guards are evaluated sequentially,
  - first true guard determines the resulting value.
- if-then-else expression:

```
  cond_expr ?  basic_expr 1 :  basic_expr2
```

Set Expressions

从集合中
选一个

- Expressions in SMV do not necessarily evaluate to *one* value.
- In general, they can represent a set of possible values.

```
init(var) := {a,b,c} union {x,y,z};
```

  - destination (lhs) can take any value in the set represented by the set expression (rhs)
  - constant c is a syntactic abbreviation for singleton {c}

句法的 缩写

c本身是{c}的缩写

LTL specifications

- LTL properties are specified with the keyword `LTLSPEC`

  ```
  LTLSPEC <ltl expression>
  ```

  - `<ltl expression>` can contain the temporal operators:

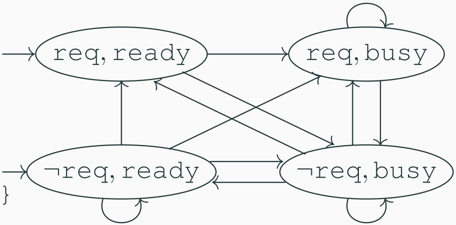
    ```
    X_   F_   G_   _U_
    ```

- e.g. condition `out = 0` holds until `reset` becomes false:

  ```
  LTLSPEC (out = 0) U (!reset)
  ```

Example

```
MODULE main
VAR
   request : boolean;
   state : {ready, busy};
ASSIGN
   init(state) := ready;
   next(state) := case
      request : busy;
      TRUE : {ready, busy}
   esac;
LTLSPEC G(request -> F state = busy);
```

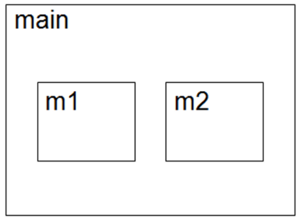


Note: `request` never receives an assignment – this models input.

Modules

An SMV program can consist of one or more module declarations.

```
MODULE add
VAR out: 0..9;
ASSIGN
    init(out) := 0;
    next(out):= (out+1) mod 10;
MODULE main
    VAR m1 : add;
        m2 : add;
        sum: 0..18;
    ASSIGN sum := m1.out + m2.out;
```



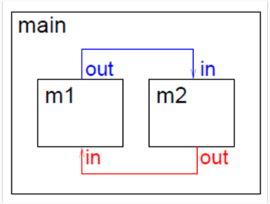
- Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- In each SMV program there must be a module `main` (top-most one).
- All variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g. `m1.out`).

Module Parameters

Module declarations may be parametric.

```
MODULE m(in)
   VAR out: 0..9;
   ...

MODULE main
   VAR m1 : m(m2.out);
       m2 : m(m1.out);
   ...
```



- Formal parameters (`in`) are substituted with the actual parameters (`m2.out`, `m1.out`) when the module is instantiated.
- Actual parameters can be any legal expression.
- Actual parameters are passed by reference.

Module Composition

- synchronous composition
  - all assignments are executed in parallel and synchronously
  - a single step of the resulting model corresponds to a step in each of the components
- asynchronous composition
  - a step of the composition is a step by exactly one process
  - variables not assigned in that process are left unchanged

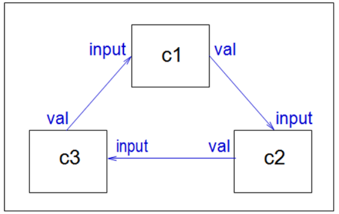
Synchronous Composition

By default, composition of modules is synchronous:

- all modules move at each step

relation but not excuetion

```
MODULE cell(input)
   VAR
      val : {red, green, blue};
   ASSIGN
      next(val) := {val, input};
MODULE main
   VAR
      c1 : cell(c3.val);
      c2 : cell(c1.val);
      c3 : cell(c2.val);
```



Synchronous Composition

A possible execution:

step	`c1.val`	`c2.val`	`c3.val`
0	**red**	**green**	**blue**
1	red	**red**	**green**
2	**green**	red	green
3	green	red	green
4	green	red	**red**
5	**red**	**green**	red
6	red	**red**	red
7	red	red	red
8	red	red	red
9	red	red	red
10	red	red	red

Asynchronous Composition

- Asynchronous composition can be specified using the keyword `process`
- In asynchronous composition one process moves at each step.

```
MODULE cell(input) VAR
   val : {red, green, blue};
   ASSIGN
      next(val) := {val, input};
MODULE main
   VAR
      c1 : process cell(c3.val);
      c2 : process cell(c1.val);
      c3 : process cell(c2.val);
```

Asynchronous Composition

A possible execution:

step	`c1.val`	`c2.val`	`c3.val`
0	**red**	**green**	**blue**
1	red	**red**	blue
2	**blue**	red	blue
3	blue	red	blue
4	blue	red	blue
5	blue	red	**red**
6	blue	**blue**	red
7	blue	blue	red
8	**red**	blue	red
9	red	blue	**blue**
10	red	blue	blue

Modelling Programs in NuSMV (part 1)

Given the following piece of code, computing the GCD of `a` and `b`, how do we model and verify it with NuSMV?

```
void main() {
...  // initialization of a and b
    while (a!=b) {
        if (a>b)
            a=a-b;
        else
            b=b-a;
    }
    ...  // GCD=a=b
}
```

Modelling Programs in NuSMV (part 2)

Step 1: label entry and exit point of every statement

```
void main() {
...  // initialization of a and b
l1:      while (a!=b) {
l2:          if (a>b)
l3:              a=a-b;
       else
l4:              b=b-a;
   }
l5:      ...  // GCD=a=b
}
```

Modelling Programs in NuSMV (part 3)

Step 2: encode the transition system using `ASSIGN`

```
MODULE main()
VAR a: 0..100; b: 0..100;
    pc: {l1,l2,l3,l4,l5};
ASSIGN
  init(pc):=l1;
  next(pc):= case
    pc=l1 & a!=b  : l2;
    pc=l1 & a=b   : l5;
    pc=l2 & a>b   : l3;
    pc=l2 & a<=b  : l4;
    pc=l3 | pc=l4 : l1;
    pc = l5       : l5;
  esac;
```

```
next(a):= case
  pc=l3 : a-b;
  TRUE  : a;
esac;

next(b):= case
  pc=l4 : b-a;
  TRUE  : b;
esac
```

reprsent the posiable relation

Example: Mutual Exclusion Protocol

Process 1:

```
repeat_forever{
out:  atomic {a := true; turn := true;}
wait:  wait (b = false or turn = false);
cs:  a := false;
}
```

=与:=的区别

||

Process 2:

```
repeat_forever{
out:  atomic {b := true; turn := false;}
wait:  wait (a = false or turn);
cs:  b := false;
}
```

Assume `a` and `b` are initially false.

Example: Mutual Exclusion Protocol in NuSMV (part 1)

```
MODULE process1(a,b,turn)
VAR
  pc: {out, wait, cs};
ASSIGN
  init(pc) := out;
  next(pc) := case
      pc=out : wait;
      pc=wait & (!b | !turn) : cs;
      pc=cs : out;
      TRUE : pc;
  esac;
  next(turn) := case
      pc=out : TRUE;
      TRUE : turn;
  esac;
  next(a) := case
      pc=out : TRUE;
      pc=cs : FALSE;
      TRUE : a;
  esac;
  next(b) := b;
```

Example: Mutual Exclusion Protocol in NuSMV (part 2)

```
MODULE process2(a,b,turn)
VAR
  pc: {out, wait, cs};
ASSIGN
  init(pc) := out;
  next(pc) := case
      pc=out : wait;
      pc=wait & (!a | turn) : cs;
      pc=cs : out;
      TRUE : pc;
  esac;
  next(turn) := case
      pc=out : FALSE;
      TRUE : turn;
  esac;
  next(b) := case
      pc=out : TRUE;
      pc=cs : FALSE;
      TRUE : b;
  esac;
  next(a) := a;
```

Example: Mutual Exclusion Protocol in NuSMV (part 3)

```
MODULE main
VAR
  a : boolean;
  b : boolean;
  turn : boolean;
  p1 : process process1(a,b,turn);
  p2 : process process2(a,b,turn);
ASSIGN
  init(a) := FALSE;
  init(b) := FALSE;
LTLSPEC
  G(!(p1.pc=cs & p2.pc=cs))
LTLSPEC
  G(a -> F(p1.pc=cs)) & G(b -> F(p2.pc=cs))
```

->和且的区别

Running NUSMV (interactive mode)

```
% NuSMV -int add.smv
NuSMV > go
NuSMV > check_ltlspec
NuSMV > quit
```

- `go` abbreviates the sequence of commands `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`
- for command options, use -h or look in the NuSMV User Manual

Example: Mutual Exclusion Protocol in NuSMV (part 3)

NuSMV output:

```
-- specification  G !(p1.pc = cs & p2.pc = cs)  is true
-- specification ( G (a ->  F p1.pc = cs) &  G (b ->  F p2.pc = cs))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    a = FALSE
    b = FALSE
    turn = FALSE
    p1.pc = out
    p2.pc = out
  -> Input: 1.2 <-
    _process_selector_ = p1
    running = FALSE
    p2.running = FALSE
    p1.running = TRUE
  -> State: 1.2 <-
    a = TRUE
    turn = TRUE
    p1.pc = wait
  -> Input: 1.3 <-
   _process_selector_ = p2
    p2.running = TRUE
    p1.running = FALSE
  -- Loop starts here
  -> State: 1.3 <-
    b = TRUE
    turn = FALSE
    p2.pc = wait
  -> Input: 1.4 <-
  -- Loop starts here
  -> State: 1.4 <-
  -> Input: 1.5 <-
  -- Loop starts here
  -> State: 1.5 <-
  -> Input: 1.6 <-
  -> State: 1.6 <-
```

Fairness Constraints

- allows to restrict attention to "fair" executions
- `FAIRNESS expr`
  - restricts attention to executions where `expr` is true infinitely often
- `COMPASSION (expr1, expr2)`
  - restricts attention to executions where if `expr1` is true infinitely often, then `expr2` is true infinitely often

Mutual Exclusion Revisited

- add the following declaration to `process1`:

```
COMPASSION
  (pc=wait & (!b | !turn), pc=cs);
```

- add the following declaration to `process2`:

```
COMPASSION
  (pc=wait & (!a | turn), pc=cs);
```

- NuSMV output:

```
-- specification  G !(p1.pc = cs & p2.pc = cs)  is true
-- specification ( G (a ->  F p1.pc = cs) &  G (b ->  F p2.pc = cs))  is true
```

Summary

- quick introduction to NuSMV
- from programs to NuSMV models

More on NuSMV:

- tutorial:
  `http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf`
- full manual:
  `http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`

Related Projects/Tools

The following all build on NuSMV:

- nuseen (IDE for NuSMV):
  `https://code.google.com/a/eclipselabs.org/p/nuseen/`
- planning tools, e.g. MBP
  (Model Based Planner `http://mbp.fbk.eu/`)
- Rebeca – actor-based language for modelling concurrent/reactive systems `http://rebeca-lang.org`
  - asynchronous message passing
  - event-driven computation
- AutoFOCUS3 – development tool for safety-critical embedded systems
  `https://www.fortiss.org/en/publications/software/autofocus-3`

(See `http://nusmv.fbk.eu` for full list.)