Formal Methods in Computer Scicene

UE 185.A93, WS 2019

Block 4: Model Checking

Florian Zuleger

Institute of Logic and Computation
Formal Methods in Systems Engineering

Slides by Laura Kovács

Introduction

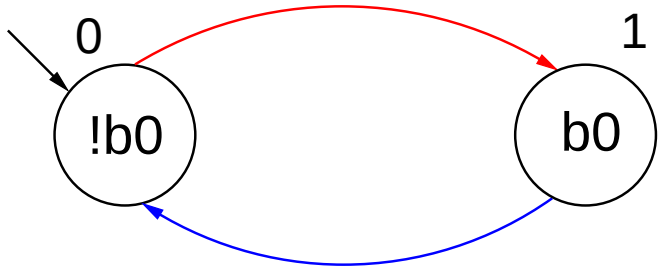
- We will use the NuSMV symbolic model checker:

  `http://nusmv.fbk.eu/`

The first NuSMV program – alternated.smv

```
MODULE main
 VAR
   b0 : boolean;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;
```



An NuSMV program consists of:

- Declarations of the state variables (`b0` in the example); the state variables determine the state space of the model.
- Assignments that define the valid initial states (`init(b0):=FALSE`).
- Assignments that define the transition relation (`next(b0):=!b0`).

Declaring state variables

The NuSMV language provides booleans, enumerative, bounded integers and words as data types:

boolean:

```
x : boolean;
```

enumerative:

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

bounded integers (intervals):

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

words:

```
w : word[8];
```

The NuSMV language provides also the possibility to define *arrays*.

Declaring the set of initial states

- For each variable, we constrain the values that it can assume in the *initial states*.

```
init(<variable>) := <simple_expression> ;
```

- `<simple_expression>` must evaluate to values in the domain of `<variable>`.
- If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.

Expressions

- Arithmetic operators:
  `+` `-` `*` `/` `mod` `-` (unary)
- Comparison operators:
  `=` `!=` `>` `<` `<=` `>=`
- Logic operators:
  `&` `|` `xor` `!` (not) `->` `<->`
- Conditional expression:

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

  if `c1` then `e1` else if `c2` then `e2` else if ... else `en`

- Set operators: `{v1,v2,...,vn}` (enumeration) `in` (set inclusion) `union` (set union)

Expressions

- Expressions in NuSMV 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} ;
```

- The meaning of `:=` in assignments is that the lhs can assume non-deterministically a value in the set of values represented by the rhs.
- A constant `c` is considered as a syntactic abbreviation for `{c}` (the singleton containing `c`).

Declaring the transition relation

- The transition relation is specified by constraining the values that variables can assume in the *next state*.

```
next(<variable>) := <next_expression> ;
```

- `<next_expression>` must evaluate to values in the domain of `<variable>`.
- `<next_expression>` depends on “current” and “next” variables:

```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

- If no `next()` assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.

Declaring the transition relation

Example: short.smv

```
MODULE main

 VAR
  request : boolean;
  state : {ready,busy};

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

Specifying normal assignments

- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

```
<variable> := <simple_expression> ;
```

- `<simple_expression>` must evaluate to values in the domain of the `<variable>`.

Specifying normal assignments

```
MODULE main
 VAR
   b0 : boolean;
   out : 0..3;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;

   out := 2*b0;
```

The DEFINE declaration

- DEFINE declarations can be used to define *abbreviations*.
- An alternative to normal assignments.
- Syntax:

  ```
  DEFINE <id> := <simple_expression> ;
  ```

  alternative to

  ```
  VAR <id>:type; ASSIGN <id> :=  <simple_expression>;
  ```

- They are similar to macro definitions.

- No new state variable is created for defined symbols (hence, no added complexity to model checking).

- Each occurrence of a defined symbol is replaced with the body of the definition.

The DEFINE declaration

```
MODULE main
 VAR
   b0 : boolean;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;

 DEFINE
   out := 2*b0;
```

Modules

A NuSMV program can consist of one or more *module declarations*.

In each NuSMV specification there must be a module `main`.

Specifications

In the NuSMV language:

- Specifications can be added in any module of the program
- Each property is verified separately
- The result of a property verification is either “true” or “false”. In the latter case, a counterexample is generated

Specifications

- We focus on the following properties:
  - properties on the computation tree (*branching time* temporal logics):
    - CTL (`SPEC`)
  - properties on the computation paths (*linear time* temporal logics):
    - LTL (`LTLSPEC`)

CTL specifications

- CTL properties are specified via the keyword `SPEC`:

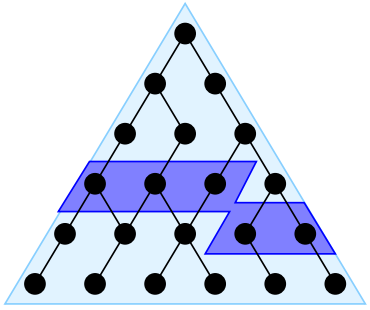
  ```
  SPEC <ctl_expression>
  ```

  where `<ctl_expression>` can contain the following temporal operators:

  ```
  AX _   AF _   AG _   A[_ U _]
  EX _   EF _   EG _   E[_ U _]
  ```

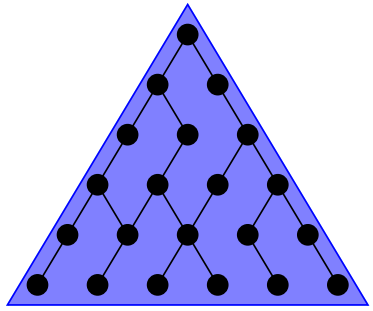
CTL specifications

finally P



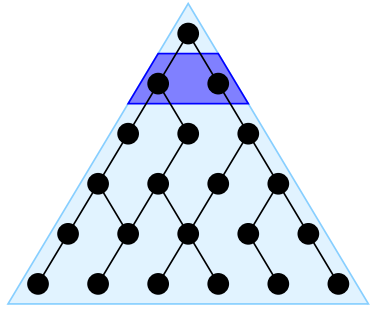
AF P

globally P



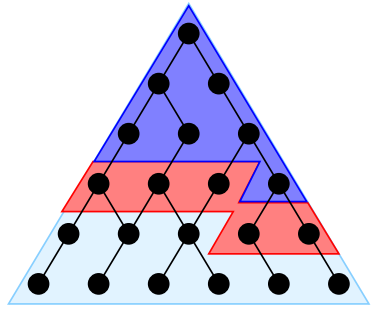
AG P

next P

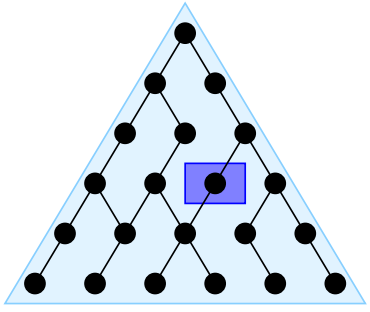


AX P

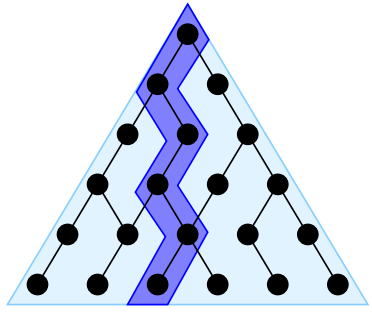
P until q



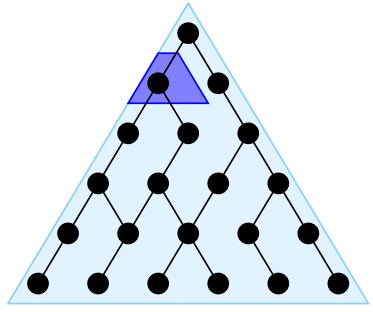
A[P U q]



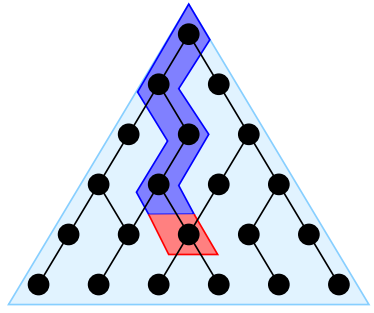
EF P



EG P



EX P



E[P U q]

CTL specifications

Examples of specifications aas:

- It is possible to reach a state in which `out = 3`
- A state in which `out = 3` is always reached
- It is always possible to reach a state in which `out = 3`
- Every time a state with `out = 2` is reached, a state with `out = 3` is reached afterward

CTL specifications

Examples of specifications:

- It is possible to reach a state in which `out = 3`

  ```
  SPEC EF out = 3
  ```

- A state in which `out = 3` is always reached

  ```
  SPEC AF out = 3
  ```

- It is always possible to reach a state in which `out = 3`

  ```
  SPEC AG EF out = 3
  ```

- Every time a state with `out = 2` is reached, a state with `out = 3` is reached afterward

  ```
  SPEC AG (out = 2 -> AF out = 3)
  ```

A simple mutex example – semaphore_ctl.smv

```
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;
      FALSE : semaphore;
    esac;
FAIRNESS
   running
```

A simple mutex example

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

SPEC
  AG (proc1.state = entering -> AF proc1.state = critical); -- liveness

> NuSMV semaphore.smv

-- specification AG (proc1.state = entering -> AF proc1.state = critical)
is false
-- as demonstrated by the following execution sequence [...]
```

LTL specifications

- LTL properties are specified via the keyword `LTLSPEC`:

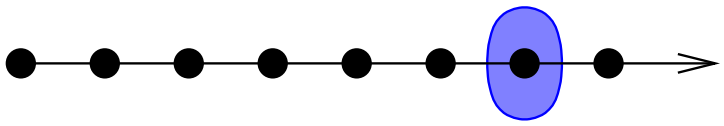
  ```
  LTLSPEC <ltl_expression>
  ```

  where `<ltl_expression>` can contain the following temporal operators:

  ```
  X _   F _   G _   _ U _
  ```

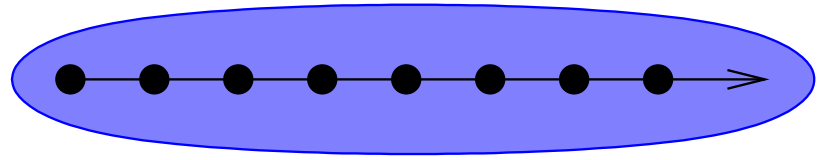
LTL specifications

finally P



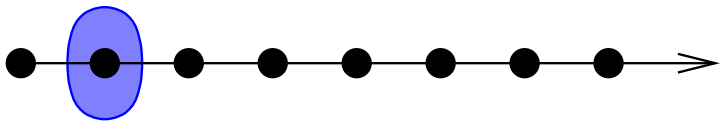
F P

globally P



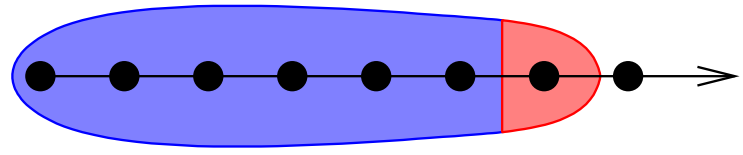
G P

next P



X P

P until q



P U q

LTL specifications

Examples of specifications:

- A state in which `out = 3` is eventually reached

```
LTLSPEC F out = 3
```

- Every time a state with `out = 2` is reached, a state with `out = 3` is reached afterward

```
LTLSPEC G (out = 2 -> F out = 3)
```

Example of annotated NuSMV program: `semaphore_ltl.smv`

Bounded Model Checking

Key ideas:

- looks for counter-example paths of increasing length k
  - oriented to finding bugs
- for each k, builds a boolean formula that is satisfiable iff there is a counter-example of length k
- satisfiability of the boolean formulas is checked using a *SAT procedure*

Bounded Model Checking

An example of a deterministic counter modulo 8 -- counter_bmc.smv:

```
MODULE main
VAR
  y   : 0 .. 15;
ASSIGN
  init(y) := 0;
TRANS
  case
     y=7  : next(y) = 0;
     TRUE : next(y) = ((y+1) mod 16);
  esac;
LTLSPEC G (y=4 -> X y=6)
```

```
> NuSMV -bmc counter_bmc.smv
```

The Ferryman Puzzle in NuSMV

The Ferryman Puzzle:

A ferry-man has to bring a goat, a cabbage, and a wolf safely from the right bank to the left bank of the river.

The ferry-man can cross the river alone or with exactly one of these three passengers.

At any time, either the ferry-man should be on the same bank as the goat, or the goat should be alone on a bank.
Otherwise, the goat could go ahead and eat the cabbage or the wolf may eat the goat.

Nice visualisation at:
`http://www.mathcats.com/explore/river/crossing.html`

The Ferryman Puzzle - Variables

```
MODULE main
VAR
-- the man and the three items
  cabbage : {right,left};
  goat    : {right,left};
  wolf    : {right,left};
  man     : {right,left};
-- possible moves
  move    : {c, g, w, e};

DEFINE
  carry_cabbage := move=c;
  carry_goat := move=g;
  carry_wolf := move=w;
  no_carry := move=e;
```

The Ferryman Puzzle

```
-- initially everything is on the right bank
ASSIGN
  init(cabbage) := right;
  init(goat)    := right;
  init(wolf)    := right;
  init(man)     := right;

TRANS
  carry_cabbage ->
                  cabbage=man &
                  next(cabbage)!=cabbage &
                  next(man)!=man &
                  next(goat)=goat &
                  next(wolf)=wolf
...
```

The Ferryman Puzzle – ferryman.smv

```
DEFINE

-- goat and wolf    must not be left unattended !
-- goat and cabbage must not be left unattended !
safe_state := (goat = wolf | goat = cabbage) -> goat = man;

goal := cabbage = left & goat = left & wolf = left;

-- spec to find a solution to the problem
SPEC
   ! E[safe_state U goal]
```