Information Systems Analysis

Temporal Logic and Timed Automata

(8)

System model verification in NuSMV

version 2.6

Contents of the lecture

System modelling

- Indirect modelling
- Direct modelling
- FAIRNESS constraints
- Nondeterminism

Contents of the lecture

Mistakes in system modelling

- Different definitions of a variable
- Recursive definition of a variable
- Mutual dependency of variables
- Contradictions in expressions INIT, INVAR and TRANS

Contents of the lecture

System verification

- Possibilities
- Property kinds to verify
- Counting a minimal and maximal path of states

Contents of the lecture

Examples

- Departure of a plane (clock and time of events)
- Function with a for loop (analysis of a program’s code)

Contents of the lecture

Interactive work

- Initial operations
- Model verification
- Model simulation
- Restart and end of work
- Executions of a script with operations
- Description of operations performed by NuSMV

System modelling

- Indirect modelling
- Direct modelling
- FAIRNESS constraints
- Nondeterminism

System modelling

Indirect modelling

```
MODULE main

VAR    a : boolean;
       b : 0..4;

ASSIGN   init(a):= TRUE;
         next(a):= !a;
         init(b):= {0,2,4};
         next(b):= case
            next(a) : {0,2,4};
            !next(a) : {1,3};
         esac;

CTLSPEC AG(a -> b in {0,2,4})

INVARSPEC (!a -> b in {1,3})
```

Direct modelling

```
MODULE main

VAR    a : boolean;
       b : 0..4;

INIT   a = TRUE &
       b in {0,2,4};
TRANS next(a) = !a;
TRANS next(b) in case
         next(a) : {0,2,4};
         !next(a) : {1,3};
      esac;

CTLSPEC AG(a -> b in {0,2,4})

INVARSPEC (!a -> b in {1,3})
```

System modelling

Indirect modelling

- Behaviour of an automaton is defined by specifying initial and next values of state variables.
- Example:

```
ASSIGN  init(a):= TRUE;
        next(a):= !a;
        init(b):= {0,2,4};
        next(b):= case
           next(a)  : {0,2,4};
           !next(a) : {1,3};
        esac;
```

- Operator `init` defines the initial value of a variable.
- Operator `next` defines the value of a variable in the next state.

System modelling

Indirect modelling

- If the initial value of a variable is not given, it will get any value from its range of values.

  (There exists at least 1 initial state.)

- If the next value of a variable is not given, it will get any value from its range of values.

  (There exists at least 1 next state for every state.)

Remark

- Every model defined indirectly can be defined directly.
- Not every model defined directly can be defined indirectly.

System modelling

Direct modelling

- Behaviour of an automaton is defined by logic expressions.
- Logic expressions express:
  - initial states,
  - reachable states,
  - transitions between states.
- Results of lack of expressions or of their mutual contradiction:
  - an empty set of initial states,
  - unreachable states,
  - lack of reachable states.

System modelling

Direct modelling

- Specification of initial values of variables:

```
INIT logic_expression
```

  - The expression given after `INIT` describes initial values of variables.
  - Example of specification of values of variables `a` and `b`:

```
INIT  a = TRUE &
      b in {0,2,4}
```

- If the initial value of a variable is not given,
  it will get any value from its range of values.
- If an untrue expression is given, then there are no initial states
  (model verification may be incorrect).
- Using the operator `next` is not allowed.

System modelling

Direct modelling

- Specification of reachable states by state invariants:

  `INVAR` *`logic_expression`*

  - The expression given after `INVAR` describes the values of variables, that characterise every state.
  - Example of specification of values of variables `a` and `b`:

    ```
    INVAR a=TRUE | a=FALSE
    INVAR !a -> b in {1,3}
    ```

- If an untrue expression is given, then there are no reachable states (model verification may be incorrect).
- Invariant definitions are not mandatory.
- Using the operator `next` is not allowed.

Direct modelling

- Specification of allowed transitions between states:

```
TRANS logic_expression
```

  - The expression given after `TRANS` describes allowed values of variables in the next state.
  - Example of specification of next values of variables `a` and `b`:

```
TRANS next(a) = !a;
TRANS next(b) in case
          next(a) : {0,2,4};
          !next(a) : {1,3};
      esac;
```

- If an untrue expression is given, then there may be no next state (model verification may be incorrect).

System modelling

Direct modelling

- `INVAR` or `INIT` combined with `TRANS`?
  - 1st way – invariantly a = 1:

    ```
    INVAR a=1
    ```

  - 2nd way – in the initial and every following state a = 1:

    ```
    INIT a=1
    TRANS next(a)=1
    ```

  - The effect seems to be the same, but the 1st way is more effective.
    - In this situation it is recommended to use an invariant.

FAIRNESS constraints

- Constraint `JUSTICE` *`expression`*
  - Alternatively: `FAIRNESS` *`expression`*
  - Model verification consists of these paths only, where the *expression* is true infinitely many times, e.g.:

    ```
    VAR   a : boolean;
    JUSTICE !a
    ```

  - It corresponds to the formula AG(AF(¬(a))).
  - Using the operator `next` in the *expression* is not allowed.

System modelling

FAIRNESS constraints

- Constraint `COMPASSION (`*`expression1,expression2`*`)`
  - Model verification consists of these paths only, where:
    - if the *expression1* is true infinitely many times,
    - then the *expression2* is also true infinitely many times on the same paths, e.g.:

```
VAR   a : boolean;
      b : boolean;
COMPASSION (!a,!b)
```

  - It corresponds to the formula AG(AG(AF(¬(*a*))) ⇒ AG(AF(¬(*b*)))).
  - Using the operator `next` in the *expressions* is not allowed.
  - NuSMV does not fully support the COMPASSION yet.

Nondeterminism

- Definition of a variable requires to give a set of its values, e.g.:

```
VAR
   a : 0..10;
   b : {s1, s2, s3};
```

- If no instruction assigns any value to a variable, then the variable gets a random value of the range of its values.
- If an instruction assigns a subset of a variable's set of values to the variable, then the variable gets a random value of this subset, e.g.:

```
b := {s1,s3}
```

Mistakes in system modelling

- Different definitions of a variable
- Recursive definition of a variable
- Mutual dependency of variables
- Contradictions in expressions INIT, INVAR and TRANS

Mistakes in system modelling

Different definitions of a variable

- Every variable should have one definition only, that defines its value for a given state:
  - wrong:
    ```
    init(a) := TRUE;
    init(a) := FALSE;
    ```
  - wrong:
    ```
    b := a;
    b := a+1;
    ```
  - wrong:
    ```
    init(c) := a;
    c := b;
    ```
  - good:
    ```
    init(a) := {TRUE,FALSE};
    ```

Mistakes in system modelling

Recursive definition of a variable

- Value of a variable cannot depend on its value from the same state:
  - wrong: `a := a+1;`
  - wrong: `next(a) := next(a)+1;`
- But it may depend on its value from the next state:
  - good: `next(a) := a+1;`

Mistakes in system modelling

Mutual dependency of variables

- Values of variables in the same state cannot be mutually dependent:
  - wrong:
    ```
    a := b+1;
    b := a-1;
    ```
  - wrong:
    ```
    next(a) := next(b);
    next(b) := next(a);
    ```
- But values of variables in different states may be mutually dependent:
  - good:
    ```
    next(a) := b;
    next(b) := a;
    ```
  - good:
    ```
    next(a) := next(b);
    next(b) := a;
    ```

Mistakes in system modelling

Contradictions in expressions INIT, INVAR and TRANS

- If an untrue expression `INIT` is given,
  then there are no initial states.
- If an untrue expression `INVAR` is given,
  then there are no reachable states.
- If an untrue expression `TRANS` is given,
  then there may not be a next state.

- These mistakes are reported by NuSMV.
- These mistakes may lead to an incorrect model verification.

System verification

- Possibilities
- Property kinds to verify
- Counting a minimal and maximal path of states

System verification

Possibilities

- Verification is automatic.
- Specification of a system is given by temporal logic formulas.
- Available logics: LTL, CTL, LTL^{-}, RTCTL (with upper and lower bounds for temporal operators) and PSL.
- All well–formed formulas are allowed.
- Every formulas is verified independently of the others.
- Verification of a formula returns *true* or *false*.
- The *false* result is returned with a counterexample (a path of states), if it can be generated.
- Length of minimal and maximal path between two determined states can by counted.

System verification

Property kinds to verify

- Properties described in LTL logic (dealing with linear time):

  `LTLSPEC` *`LTL_formula`*

- Properties described in CTL logic (dealing with branching time):

  `CTLSPEC` *`CTL_formula`*

- Properties described in logics LTL^{-}, PSL, RTCTL.
- Invariants (dealing with every state of the model):

  `INVARSPEC` *`logic_expression`*

System verification

Counting a minimal and maximal path of states

- Expression `COMPUTE` counts length of a path (number of states) between two specified states.
- Specification of a state is a logic expression expressing values of selected state variables in this state.
- Counting the minimal path:

  COMPUTE MIN[*state1*,*state2*]

- Counting the maximal path:

  COMPUTE MAX[*state1*,*state2*]

- The result is a number of states or INFINITY.

Examples

- Departure of a plane (clock and time of events)
- Function with a for loop (analysis of a program’s code)

Examples

Departure of a plane (clock and time of events)

- **Description of the situation**
  - A runway intersects a taxiway.
  - An aircraft begins moving before the intersection, accelerating.
  - The aircraft, accelerating, reaches the V1 velocity (in 6..8 seconds), and then takes off (in 1..3 seconds).
  - The take-off of the aircraft may happen before, on or after the intersection.
  - An intruder may appear on the intersection at any moment.
  - The intruder, when appears on the intersection, does not disappear from it.
  - If the aircraft accelerates before or on the intersection, where the intruder appears, it decelerate, if its velocity < V1.
  - Decelerating aircraft stops (in 3..4 seconds) before, on, or after the intersection.
  - If the aircraft and the intruder are on the intersection, a collision may happen.
  - Final states: the aircraft takes off, the aircraft stands, there is a collision.

Examples

Departure of a plane (clock and time of events)

```
MODULE main

VAR
   --location of the aircraft
   --in relation to the intersection
   location : {before, on, after};
   --kind of a movement of the aircraft
   movement: {accelerating, decelerating, standing, taking_off};
   --clock for time of the movement (reset to zero at the moment
   --of the beginning of a new movement kind)
   t : 0..9;
   --intruder on the intersection
   intruder : boolean;
   --collision with the intruder
   collision : boolean;
   --aircraft's velocity >= v1 (deceleration is forbidden)
   v1 : boolean;
```

Examples

Departure of a plane (clock and time of events)

```
--INITIAL STATE

INIT
   --the aircraft is before the intersection
   location = before &
   --the aircraft is accelerating
   movement = accelerating &
   --the time of acceleration begins
   t = 0 &
   --there is no intruder on the intersection
   intruder = FALSE &
   --there is no collision
   collision = FALSE &
   --the aircraft's velocity < v1
   v1 = FALSE
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE CLOCK t

TRANS next(t) in case
   --resetting the clock when taking-off begins
   movement = accelerating & next(movement) = taking_off : 0;
   --resetting the clock when decelerating begins
   movement = accelerating & next(movement) = decelerating : 0;
   --resetting the clock when standing begins
   movement = decelerating & next(movement) = standing : 0;
   --in other case, with any automaton state change,
   --one second passes
   TRUE : (t + 1) mod 10; esac;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE INTRUDER

TRANS next(intruder) in case
   --the intruder may appear at any moment
   !intruder : {FALSE,TRUE};
   --the intruder cannot disappear from the intersection,
   --if it already is there
   TRUE : intruder; esac;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE v1 VELOCITY

TRANS next(v1) in case
   --the v1 cannot be reached in the time t < 6
   !v1 & movement = accelerating & next(t)<6 : FALSE;
   --the v1 may be reached in the time t < 8
   !v1 & movement = accelerating & next(t)<8 : {TRUE,FALSE};
   --the v1 is reached at most in the time t = 8
   !v1 & movement = accelerating & next(t)=8 : TRUE;
   --once reached, the v1 velocity does not get smaller
   TRUE : v1; esac;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE COLLISION

--the collision is impossible, if there is no intruder
--or the aircraft is before the intersection
INVAR !intruder | location = before -> !collision;

TRANS next(collision) in case
   --if there is the collision, it will not pass away
   collision : TRUE;
   --if there is no collision, it is possible then,
   --if the intruder and the aircraft are on the intersection
   intruder & location = on : {FALSE, TRUE};
   --other states do not affect the collision
   TRUE : collision; esac;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE LOCATION OF THE AIRCRAFT

TRANS next(location) in case
   --the standing or taking-off aircraft does not change
   --its location (final state)
   movement = standing | movement = taking_off : location;
   --the aircraft being before the intersection may enter it
   location = before : {before, on};
   --the aircraft being on the intersection may leave it
   location = on : {on, after};
   --the aircraft being after the intersection does not change
   --its location
   location = after: after; esac;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE MOVEMENT OF THE AIRCRAFT (1)

TRANS next(movement) in case
   --the aircraft accelerating with the velocity >= v1
   --cannot take off if there is the collision
   --(no change of movement kind)
   movement = accelerating & v1 & collision : accelerating;
   --the aircraft accelerating with the velocity >= v1
   --cannot take off in time t < 1
   movement = accelerating & v1 & next(t)<1 : accelerating;
   --the aircraft accelerating with the velocity >= v1
   --may take off in time t < 3 (if there is no collision)
   movement = accelerating & v1 & next(t)<3 :
                                  {accelerating, taking_off};
   --the aircraft accelerating with the velocity >= v1 takes off
   --at last in the time t = 3 (if there is no collision)
   movement = accelerating & v1 & next(t)=3 : taking_off;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE MOVEMENT OF THE AIRCRAFT (2)

-- ...
   --the aircraft accelerating with the velocity < v1
   --still accelerates, if there is no intruder
   movement = accelerating & !v1 & !intruder : accelerating;
   --the aircraft accelerating with the velocity < v1
   --decelerates, if there is the intruder on the intersection
   movement = accelerating & !v1 & intruder : decelerating;
```

Examples

Departure of a plane (clock and time of events)

```
--BEHAVIOUR OF THE MOVEMENT OF THE AIRCRAFT (3)

-- ...
   --the decelerating aircraft cannot stop in the time t < 3
   movement = decelerating & next(t) < 3 : decelerating;
   --the decelerating aircraft may stop in the time t < 4
   movement = decelerating & next(t) < 4 :
                                       {decelerating, standing};
   --the decelerating aircraft will stop at last in the time t=4
   movement = decelerating & next(t) = 4 : standing;
   --the standing or taking off aircraft does not change
   --its kind of movement
   movement = standing | movement = taking_off : movement;
   --other states do not affect the movement
   TRUE : movement; esac;
```

Examples

Departure of a plane (clock and time of events)

- Verification of behaviour of the clock:

```
--Incrementation of the clock with every state change (mod 10)
   CTLSPEC AG(t=0 -> AX(t=1))
   CTLSPEC AG(t=9 -> AX(t=0))
   COMPUTE MIN[t=0,t=1]        --should be 1
   COMPUTE MAX[t=0,t=1]        --should be 1

--Change of a kind of movement of the aircraft resets the clock
--(e.g. change from decelerating to standing)
   CTLSPEC AG(movement=decelerating & AX(movement=standing)
                                                    -> AX(t=0))
   CTLSPEC AG(movement=decelerating & AX(movement=standing)&t!=0
                                                    -> AX(t=0))
```

Examples

Departure of a plane (clock and time of events)

- Verification of behaviour of the velocity V1:

```
--Accelerating aircraft reaches the v1 velocity
--in 6..8 seconds
   CTLSPEC EF(!v1 & movement=accelerating -> EX v1)
   CTLSPEC AG(!v1 & movement=accelerating & AX t=8 -> AX v1)
   CTLSPEC AG(!v1 & movement=accelerating & AX t<6 -> AX !v1)
   CTLSPEC AG(!v1 & movement=accelerating & AX t>=6 & AX t<8
                                          -> EX !v1)  --correct
   CTLSPEC AG(!v1 & movement=accelerating & AX t>=6 & AX t<8
                                          -> AX !v1)  --incorrect
```

Examples

Function with a for loop (analysis of a program's code)

- The function calculates the sum of numbers from 0 to x:

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)
  s = s + i;
 return s;}
```

- The goal – analyse the for loop, e.g.:
  - find the maximal number of the loop execution,
  - check reachability of s=21 for x=6.

Examples

Function with a for loop (analysis of a program's code)

- Labels of executed operations:

```
    int sum(int x){
     int s = 0; //sum of numbers       e0 (this can be omitted)
     for(int i = 0; i <= x; i++)       e1
      s = s + i;                       e2
     return s;}                        e3
```

- Model of the operations:

```
MODULE main
FROZENVAR x : 0..6;   --constant x of random value of 0 to 6
VAR e : {e1, e2, e3}; --labels of operations
    s : 0..21;        --sum of numbers
    i : 0..7;         --iterator of the loop
    c : 0..6;         --counter of loop executions
    a : 1..3;         --flag of reachability of deadlock
```

Examples

Function with a for loop (analysis of a program’s code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)      e1
  s = s + i;                      e2
 return s;}                       e3
```

- Model of the operations:

```
ASSIGN
--navigation of operations:
init(e) := e1;            --beginning in e1
next(e) := case
 e = e1 & i <= x : e2;  --transition to e2 from e1 if i<=x
     --(to perform the first operation from the for block)
 e = e1 & i > x : e3;   --transition to e3 from e1 if i>x
     --(not to perform operations from the for block)
 e = e2 : e1;           --return to e1 from e2
     --(the last operation from the for block is done)
 e = e3 : e3; esac;     --no entry from e3 (the end of program)
```

Examples

Function with a for loop (analysis of a program’s code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)       e1
  s = s + i;                       e2
 return s;}                        e3
```

- Model of the operations:

```
ASSIGN
--variability of s:
init(s) := 0;            --initially s=0
next(s) := case
 s + i > 21 : s;         --no change of s
                         --(not to exceed its range)
 next(e) = e2 : s + i;   --i is added to s (in operation e2)
 TRUE : s; esac;         --no change of s
                         --(in other cases)
```

Examples

Function with a for loop (analysis of a program's code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)      e1
  s = s + i;                      e2
 return s;}                       e3
```

- Model of the operations:

```
ASSIGN
--variability of i:
init(i) := 0;             --initially i=0
next(i) := case
 i + 1 > 7 : i;           --no change of i
                          --(not to exceed its range)
 next(e) = e1 : i + 1;    --incrementation of i
                          --(in operation e1, i.e. after e2)
 TRUE : i; esac;          --no change of i
                          --(in other cases)
```

Examples

Function with a for loop (analysis of a program's code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)      e1
  s = s + i;                      e2
 return s;}                       e3
```

- Model of the operations:

```
ASSIGN
--variability of c:
init(c) := 0;            --initially c=0
next(c) := case
 c + 1 > 6 : c;          --no change of c
                         --(not to exceed its range)
 next(e) = e2 : c + 1;   --incrementation of c
                         --(in operation e2, i.e. for block)
 TRUE : c; esac;         --no change of c
                         --(in other cases)
```

Examples

Function with a for loop (analysis of a program's code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)      e1
  s = s + i;                      e2
 return s;}                       e3
```

- Model of the operations:

```
ASSIGN
--variability of a:
init(a) := 1;  --initially a=1 (there will be a next state)
next(a) := case
 a = 1 & e = next(e) & s = next(s) & i = next(i) & c = next(c) : 2;
              --a is set to 2
              --(if there is no change of state for the 1st time)
 a = 2 : 3;   --a is set to 3
              --(so 2 be in one state only)
 TRUE : a; esac;  --no change of a
                  --(in other cases)
```

Examples

Function with a for loop (analysis of a program’s code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)      e1
  s = s + i;                      e2
 return s;}                       e3
```

- Verification of the model:
  - Does the flag *a* work correctly?

    `CTLSPEC AF(a=2 & AX(AG(a=3)))`   **true**

Examples

Function with a for loop (analysis of a program’s code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)         e1
  s = s + i;                         e2
 return s;}                          e3
```

- Code analysis:
  - What is the maximal length of the path of states to the final state?

    `COMPUTE MAX[a=1, a=2]` **16**

  - What is the maximal number of the *for* loop execution?
    - generate a maximal length path,
    - check the final value of the *c* counter on the path.

    `CTLSPEC !EBF 16..16 (a=2)` **false** **c = 6**

```
-> State: 5.12 <-
  e = e2
  s = 15
  c = 6
-> State: 5.13 <-
  e = e1
  i = 6
-> State: 5.14 <-
  e = e2
  s = 21
-> State: 5.15 <-
  e = e1
  i = 7
-> State: 5.16 <-
  e = e3
-> State: 5.17 <-
  a = 2
```

Examples

Function with a for loop (analysis of a program's code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)        e1
  s = s + i;                        e2
 return s;}                         e3
```

- Code analysis:
  - Can the *for* loop be executed 6 times?

    ```
    CTLSPEC EF(c=6)       false
    CTLSPEC !EF(c=6)      false       How is it possible?
    ```

  - Is the sum *s* = 21 reachable?

    ```
    CTLSPEC EF(s=21)      false
    CTLSPEC !EF(s=21)     false       How is it possible?
    ```

Examples

Function with a for loop (analysis of a program’s code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)       e1
  s = s + i;                       e2
 return s;}                        e3
```

- Code analysis:
  - If there is more than 1 initial state, then reachability checking for a given state by the formula `EF(state_to_check)` will return *false*, if this state is not reachable from <u>every</u> initial state, even so if it is reachable from some initial state.
    - solution 1: choose an initial state: `initial_state -> EF(state_to_check)`
    - solution 2: check non-reachability of the state: `!EF(state_to_check)`

Examples

Function with a for loop (analysis of a program's code)

```
int sum(int x){
 int s = 0; //sum of numbers
 for(int i = 0; i <= x; i++)       e1
  s = s + i;                       e2
 return s;}                        e3
```

- Code analysis:
  - Is the sum *s* = 21 reachable?

    ```
    CTLSPEC x=6 -> EF(s=21)    true
    CTLSPEC !EF(s=21)          false
    ```

  - Will the sum *s* = 21 be reached?

    ```
    CTLSPEC x=6 -> AF(s=21)    true
    ```

Interactive work

- Initial operations
- Model verification
- Model simulation
- Restart and end of work
- Executions of a script with operations
- Description of operations performed by NuSMV

Interactive work

Initial operations

- Start working with a .smv file in the interactive mode:

  ```
  NuSMV -int file
  ```

- Read the model of a system:

  ```
  read_model
  ```

- Create modules and processes:

  ```
  flatten_hierarchy
  ```

- Show a list of input variables and state variables: (optional)

  ```
  show_vars
  ```

Interactive work

Initial operations

- Show variables that are dependent on a given expression:
  (optional)

```
show_dependencies -e expession
```

- Create variables to compile the model into BDD (binary decision diagrams):

```
encode_variables
```

- Write the order of variables to a file:
  (optional)

```
write_order
```

- Compile the model into BDD:

```
build_model
```

Interactive work

Initial operations

- Initialise the system ready to be verified:

```
go
```

- Read and compile the model into BDD, verify the model and count a set of reachable states:

```
process_model
```

- Count a set of reachable states:

```
compute_reachable
```

- Show reachable states:
  (optional)

```
print_reachable_states -v
```

Interactive work

Model verification

- Show all properties:

  `show_property`

- Add a property of a given kind to the verification:

  `add_property -`*`kind`*` -p ”`*`formula`*`”`

- Add the property to verification in the context of a given module:

  `add_property -`*`kind`*` -p ”`*`formula`*` IN `*`module`*`”`

Kind: `c` (CTL formula), `l` (LTL formula), `s` (PSL formula), `i` (invariant), `q` (counting a path).

Interactive work

Model verification

- Verify a CTL formula of a given number:

  `check_ctlspec -n` *`number`*

- Verify a given CTL formula:

  `check_ctlspec -p "`*`formula`*`"`

- Verify a given CTL formula in the context of a given module:

  `check_ctlspec -p "`*`formula`*` IN `*`module`*`"`

Similarly for LTL specification: `check_ltlspec`

Interactive work

Model verification

- Check possibility of a deadlock of the system:

```
check_fsm
```

- Count length of a path between given states (for a given number of an expression):

```
check_compute -n number
```

- Count length of the minimal path between given states:

```
check_compute -p ”MIN[state1,state2]”
```

- Count length of the maximal path between given states in the context of a given module:

```
check_compute -p ”MAX[state1,state2] IN module”
```

Interactive work

Model verification

- Verify an invariant of a given number:

```
check_invar -n number
```

- Verify a given invariant:

```
check_invar -p "invariant"
```

- Verify a given invariant in the context of a given module:

```
check_invar -p "invariant IN module"
```

Model simulation

- Choose randomly an initial state:

```
pick_state -r
```

- Choose an initial state from the list of available states:

```
pick_state -i
```

- Choose an initial state from a restricted list of available states – the states that satisfy a given expression:

```
pick_state -i -c "expression"
```

Interactive work

Model simulation

- Make a simulation from a chosen state:

  `simulate [-p|-v] [-r|-i] [-k` *`number_of_states`*`]`

  - show changed state variables: `-p`
  - show all state variables: `-v`
  - randomly choose from available states: `-r`
  - manually choose from available states: `-i`
  - give length of path of states (e.g. 4): `-k 4`

    (The simulation consists of 10–state paths by default.)
  - examples:

```
simulate -p -r -k 5
simulate -v -i
```

Interactive work

Model simulation

Analysis of a chosen path of states:

- Paths of states are created in result of a negative verification of a formula, and in result of a simulation.
- Show generated paths:
  - all: `show_traces -v -a`
  - a chosen one: `show_traces -v` *`path_number`*
  - a chosen one with states (from – to):
    `show_traces -v` *`path_number`*`.from_state_number:to_state_number`
- Show a number of generated paths:

  `show_traces -t`

Interactive work

Model simulation

Analysis of a chosen path of states:

- Go to a chosen state of a chosen path:

  goto_state *path_number*.state_number

- Show description of the current state of the current path:

  print_current_state -v

Restart and end of work

- Restart of work (reset of adjustments):

```
reset
```

- End of work (reset of adjustments):

```
quit
```

Interactive work

Executions of a script with operations

- Automatically make a given sequence of operations from a file:

  `NuSMV -source` *`file`*

- If an error occurs, further operations cannot be executed.

Description of operations performed by NuSMV

- Set verbosity of performed operations:

  `NuSMV -v N -int` *`file`*

  (N – level of verbosity: from 0 (nothing) to 4)

The end

Literature:

- K.L. McMillan, „The SMV system”, 2001
- A. Cimatti et al. „NuSMV – a new symbolic model checker”
- R. Cavada et al. „NuSMV 2.5 User Manual”, 2010
- R. Cavada et al. „NuSMV 2.5 Tutorial”