Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

September 12^{th}, 2016

769857918

[COPY WITH SOLUTIONS]

1 Spin

Write a Promela model for the "Prisoners' Dilemma". Two **prisoners** processes independently send a `CONFESS` or `DENY` message through a pair of **synchronous channels** to a **policeman** process. Each prisoner chooses the content of his message in a **random** fashion and independently from the other. The policeman receives the messages, decides an adequate penalty and sends back to each prisoner a `SENTENCE` message with the number of years he is supposed to spend in detention, using the same channel from which he received the initial message. The penalty is decided as follows:

- if both confess, then both spend 5 years each in prison.
- if one confesses and the other denies, then the former is free while the latter spends 20 years in prison.
- if both deny, then both spend 1 year each in prison.

Simulate the model and (visually) verify that it matches the description.

Solution:

```
mtype { SENTENCE, DENY, CONFESS };
chan rooms[2] = [0] of { mtype, byte };

proctype prisoner(byte i)
{
  mtype v;
  byte years;
  if
    :: rooms[i] ! CONFESS, 0 ->
      printf("Prisoner %d confessed.\n", i);
    :: rooms[i] ! DENY, 0;
      printf("Prisoner %d denied.\n", i);
  fi;
  rooms[i] ? SENTENCE, years;
  printf("Prisoner %d was sentenced to %d years of detention.\n", i, years);
}

proctype policeman()
{
  mtype m1, m2;
  byte v1, v2;
  rooms[0] ? m1, 0;
  rooms[1] ? m2, 0;
  if
    :: m1 == CONFESS && m2 == CONFESS ->
      v1 = 5; v2 = 5;
    :: m1 == CONFESS && m2 == DENY ->
      v1 = 0; v2 = 20;
    :: m1 == DENY && m2 == CONFESS ->
      v1 = 20; v2 = 0;
    :: m1 == DENY && m2 == DENY ->
      v1 = 1; v2 = 1;
  fi
  atomic {
    printf("Sentence decided.\n")
    rooms[0] ! SENTENCE, v1;
    rooms[1] ! SENTENCE, v2;
  }
}

init {
  run policeman(2); run prisoner(0); run prisoner(1);
}
```

2 nuXmv

Encode the following `encryptDecrypt` function for `3-bit` arrays in NuSMVor nuXmv as a **module** `encryptDecrypt(arr)`:

```
string encryptDecrypt(arr) {
l1:   i = 0;
l2:   while (i < 3) {
l3:     arr[i] = arr[i] ^ key[i]; // ^ : xor
l4:     i++;
      }
l5:   // done!
    }
```

Hints:

- use ‘`pc`‘ to keep track of the possible state values { `l1, l2, l3, l4, l5` }
- assume ‘`pc`‘ remains equal to ‘`l5`‘ once it reaches this value, and initialize it to ‘`l5`‘
- **define** ‘`key[0]`‘, ‘`key[1]`‘ and ‘`key[2]`‘ to be equal to ‘`0d4_7`‘, ‘`0d4_13`‘ and ‘`0d4_2`‘ respectively.
- ensure that the content of ‘`arr`‘ does not change if ‘pc != l3‘
- double check that the content of ‘`arr`‘ is correctly changed whenever ‘pc = l3‘

Extend the previous **module** to accept a ‘`reset`‘ signal as input, i.e. `encryptDecrypt(arr, reset)`; modify the transition relation so that whenever ‘`reset`‘ is `true` **and** ‘pc = l5‘ then the next value of ‘`pc`‘ is ‘`l1`‘.

Create a module ‘`main`‘ with 3 variables:

- ‘`arr`‘, an array of 3 elements, each of which is of type ‘`word[4]`‘
- ‘`enc`‘, an instance of the encryption module initialized with parameters ‘`arr, reset`‘
- ‘`state`‘ with values in { `CALL, EXEC` }

Define ‘`reset`‘ to be `true` iff ‘state = CALL‘. Initialize ‘`arr[0]`‘, ‘`arr[1]`‘, ‘`arr[2]`‘ and ‘`state`‘ to the values ‘`0d4_15`‘, ‘`0d4_9`‘, ‘`0d4_4`‘ and ‘`CALL`‘ respectively. The value of ‘`state`‘ changes according to these **ordered** rules:

- if ‘`state`‘ is equal to ‘`CALL`‘, then its next value is ‘`EXEC`‘
- if the value of ‘`pc`‘ in ‘`enc`‘ is equal to ‘`l5`‘, then its next value is ‘`CALL`‘
- Otherwise, ‘`state`‘ keeps its value

Encode the following properties and check that they are all verified:

- whenever the encrypting function is started, sooner or later it terminates its execution
- sooner or later, the content of ‘`arr`‘ is equal to { ‘`0ud4_8`‘, ‘`0ud4_4`‘, ‘`0ud4_6`‘ }
- the content of ‘`arr`‘ is equal to its initialization values infinitely often
- whenever ‘`state = CALL`‘, then in the next state the encrypting function starts executing (i.e. ‘`enc.pc = l1`‘)

Solution:

```
MODULE encryptDecrypt(arr, n, reset)
VAR
  pc : { l1, l2, l3, l4, l5 };
  i  : 0..n;

DEFINE
  done := pc = l5;
  key[0] := 0d4_7;
  key[1] := 0d4_13;
  key[2] := 0d4_2;

ASSIGN
  init(pc) := l5;
  next(pc) := case
    pc = l1          : l2;
    pc = l2 & i < n  : l3;
    pc = l2          : l5;
    pc = l3          : l4;
    pc = l4          : l2;
    pc = l5 & reset  : l1;
    TRUE             : pc;
  esac;

  init(i) := 0;
  next(i) := case
    pc = l1          : 0;
    pc = l4 & i < n  : i + 1;
    TRUE             : i;
  esac;

TRANS
  pc != l3 -> next(arr[0]) = arr[0] &
    next(arr[1]) = arr[1] & next(arr[2]) = arr[2];

TRANS
  pc = l3 -> (
    (i = 0 -> next(arr[0]) = (arr[0] xor key[0]) &
        next(arr[1]) = arr[1] & next(arr[2]) = arr[2]) &
    (i = 1 -> next(arr[1]) = (arr[1] xor key[1]) &
        next(arr[0]) = arr[0] & next(arr[2]) = arr[2]) &
    (i = 2 -> next(arr[2]) = (arr[2] xor key[2]) &
        next(arr[0]) = arr[0] & next(arr[1]) = arr[1])
  );
```

```
MODULE main
VAR
  arr   : array 0..2 of word[4];
  enc   : encryptDecrypt(arr, 3, reset);
  state : { CALL, EXEC };

DEFINE
  reset := state = CALL;

INIT
  (arr[0] = 0d4_15) & (arr[1] = 0d4_9) & (arr[2] = 0d4_4) & (state = CALL);

ASSIGN
  next(state) := case
    state = CALL : EXEC;
    enc.pc = l5  : CALL;
    TRUE         : state;
  esac;

-- - whenever the encrypting function is started, sooner or later
--   it terminates its execution
LTLSPEC G ((enc.pc = l1) -> F enc.done)

-- - sooner or later, the content of ‘arr‘ is equal to the array
--   { ‘0ud4_8‘, ‘0ud4_4‘, ‘0ud4_6‘}
LTLSPEC F ((arr[0] = 0ud4_8) & (arr[1] = 0ud4_4) & (arr[2] = 0ud4_6));

-- - the content of ‘arr‘ is equal to its initialization values
--   infinitely often
LTLSPEC G F ((arr[0] = 0ud4_15) & (arr[1] = 0ud4_9) & (arr[2] = 0ud4_4));

-- - whenever ‘state = CALL‘, then in the next state the encrypting function starts
--   executing (e.g. ‘enc.pc = l1‘)
LTLSPEC G (state = CALL -> X enc.pc = l1);
```