Exercise session

The Spin model checker

Introduction

Spin is a model checker for finite-state concurrent models.

Main features:

- The tool can check for assertion violations, deadlocks, unreachable code, and verify LTL properties.

- Several options for storing visited states:
  - hash table.
  - collapsed representation.
  - bitstate hashing (reduced memory footprint, possibly incomplete search).
  - minimized automaton (memory efficient and exact, but much slower).

- Choice of two search strategies:
  - depth-first search (DFS).
  - breadth-first search (BFS, less memory efficient, but produces shortest trace).

- Partial-order reductions for reducing the search space.

- Simple graphical front-end: iSpin.

For installation instructions, documentation, and examples: https://spinroot.com.

The Promela modeling language

- Processes are defined by `proctype` declarations, and are created dynamically using the `run` statement.

  Note: Processes corresponding to `active` proctypes are automatically created from the beginning.

- Processes communicate via shared variables and communication channels.

- An `init` declaration defines an initial process, that usually
  - initializes global variables, and
  - creates other processes.

Example:

```
int x1, x2;

proctype p1()
{
  printf("p1, x1: %d\n", x1);
}

proctype p2()
{
  printf("p2, x2: %d\n", x2);
}

init
{
  x1 = 1;
  x2 = 1;

  run p1();
  run p2();
}
```

Data types and variable declarations

Primitive:

`bool` or `bit`	1 bit	$[0, 1]$
`byte`	8 bits	$[0, 255]$
`short`	16 bits	$[-2^{15}, 2^{15} - 1]$
`int`	32 bits	$[-2^{31}, 2^{31} - 1]$

```
byte c1, c2 = 2, c3;
```

Arrays

```
bool table[16];
int v[8] = 1;
```

Symbolic constants

```
mtype = { MSG, ACK, NACK };

mtype message = MSG;
```

Communication channels:

```
chan xmit = [3] of { mtype, short };

xmit!MSG,10;
xmit?MSG,nb;
```

Structures:

```
typedef message
{
  bit b[8];
  int nb
};

message m;
```

Statements

- The body of a proctype takes the form of a sequence of statements. Statements are separated by semicolons (“`;`”).

- An any moment during execution, a statement is either
  - executable, or
  - blocked.

- An assignment (e.g., `x = a + b`) is always executable.

- An expression (e.g., `a + b`) can also be used as a statement. It is executable if its evaluation returns a non-zero value.

Special statements

- `skip` is always executable, and does nothing.
- `run` instantiates a new process. Such a statement is executable only if the maximum number of processes has not yet been reached.
- `printf` is always executable, and displays debugging information. This statement has no effect during verification.

Example:

```
int x1 = 0;

init
{
  run p1(10);
  x1 != 0;  // to become executable, another process must modify x1
  skip
}
```

- `assert` is always executable. It evaluates an expression, and generates an error if the returned value is equal to 0.

  Example: `assert(nb <= 10)`

The `if` statement

Syntax:

```
if
  ::  guard1 -> instructions1;
  ::  guard2 -> instructions2;
   ⋮
fi
```

Notes:

- This statement selects non-deterministically one sequence

  $$\texttt{guard}\,i\ \texttt{->}\ \texttt{instructions}\,i$$

  among those for which `guard`i is executable.

- If no guard is executable, then the `if` statement itself is not executable.

- There exists a special guard `else` that becomes executable only if none of the other guards is executable.

- The “`->`” symbol is equivalent to “`;`”. It is used by convention for separating the guards from the instructions.
- There is no need for the guards to be mutually exclusive.
- There is also a special guard `timeout` that only becomes executable if no other process is executable in the current state.

The do statement

Syntax:

```
do
  ::  guard1 -> instructions1;
  ::  guard2 -> instructions2;
   ⋮
od
```

Notes:

- The modalities are similar to those of the `if` statement. The difference is that the `do` statement repeats the operation after each execution of a sequence `guard`i `->` `instructions`i.

- The instruction `break` makes it possible to exit the loop. This instruction is always executable.

- Another possibility of exiting the loop is to use the `goto` instruction.

Atomicity

In Promela, every individual statement is executed atomically. Sequences of operations, however, can be interleaved with operations performed by other processes. There are two ways to modify this default mode of execution:

- The statement

  ```
  atomic { instructions }
  ```

  attempts to execute `instructions` without interleaving operations from other processes.

  Notes:

  - This statement is executable if the first statement of `instructions` is executable.
  - If a subsequent statement of `instructions` becomes blocked, atomicity is lost.

- The statement

  ```
  d_step { instructions }
  ```

  is similar to an `atomic` block, but

  - executes `instructions` in a single step, without generating intermediate states,
  - imposes that the block `instructions` is executed deterministically,
  - does not allow to jump in or out of `instructions`,
  - does not allow any statement inside `instructions` except the first one to become blocked.

Note: Very often, the use of `atomic` and `d_step` blocks makes it possible to greatly reduce the search space.

Example: Simple mutual exclusion

```
int s = 1;
int nb = 0;

proctype p()
{
  do
    :: skip ->
       atomic { s > 0; s-- }
       nb++;
       assert(nb == 1);
       nb--;
       s++;
  od
}

init
{
  run p();
  run p();
  run p();
}
```

Exercise 1

1. Load this example in iSpin.
2. Open the *Verification* tab, and run the model checker. Is the assertion validated?
3. Make the same experiment after removing the `atomic` keyword.
4. In the *Simulate/Replay* tab, try to generate an error trace.
5. How would you obtain the shortest possible error trace?

Verifying a LTL property

Principles:

- A LTL property can be specified in the model, with the syntax `ltl { property }`.

  Example:

  ```
  ltl { always eventually (nb == 1) }
  ```

- The negation of this property is then automatically translated into a never claim.

- To verify such a property, one must select the option *use claim* and the search mode *depth-first search* in iSpin.

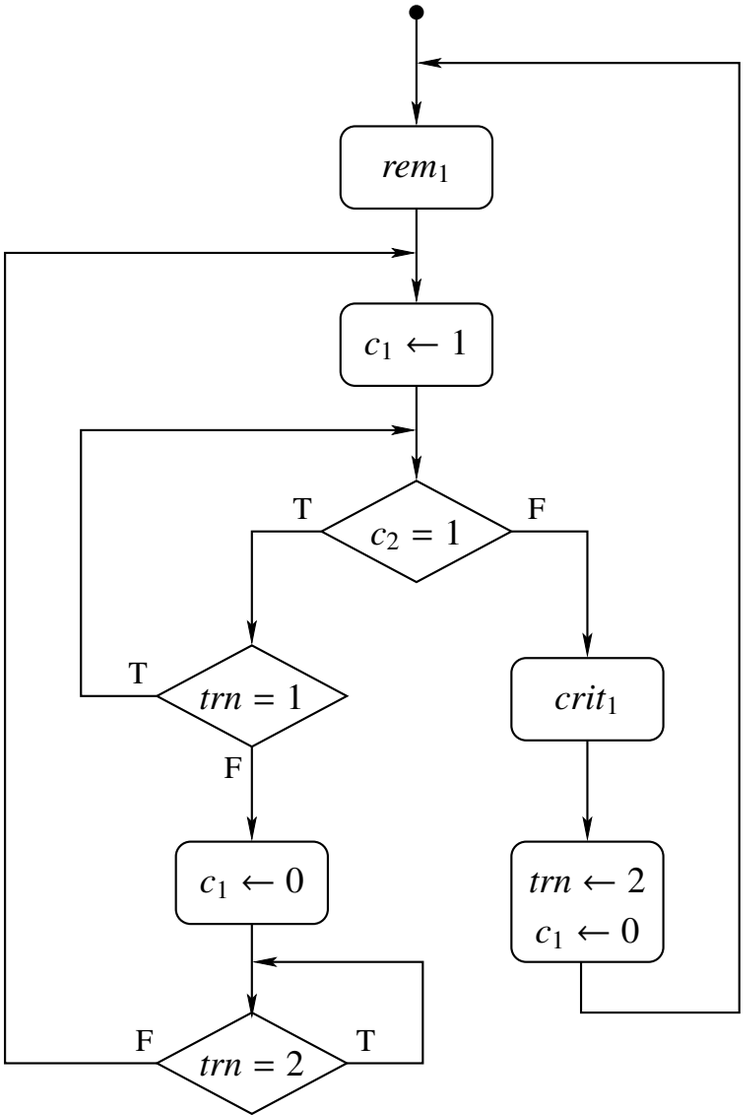
- There is a technical restriction: in order for the partial-order reductions to be correct, the property to be verified must be stutter-invariant. This is achieved by forbidding the use of the *Next* operator in LTL properties.

- If needed, there is an option for enforcing a weak fairness constraint.

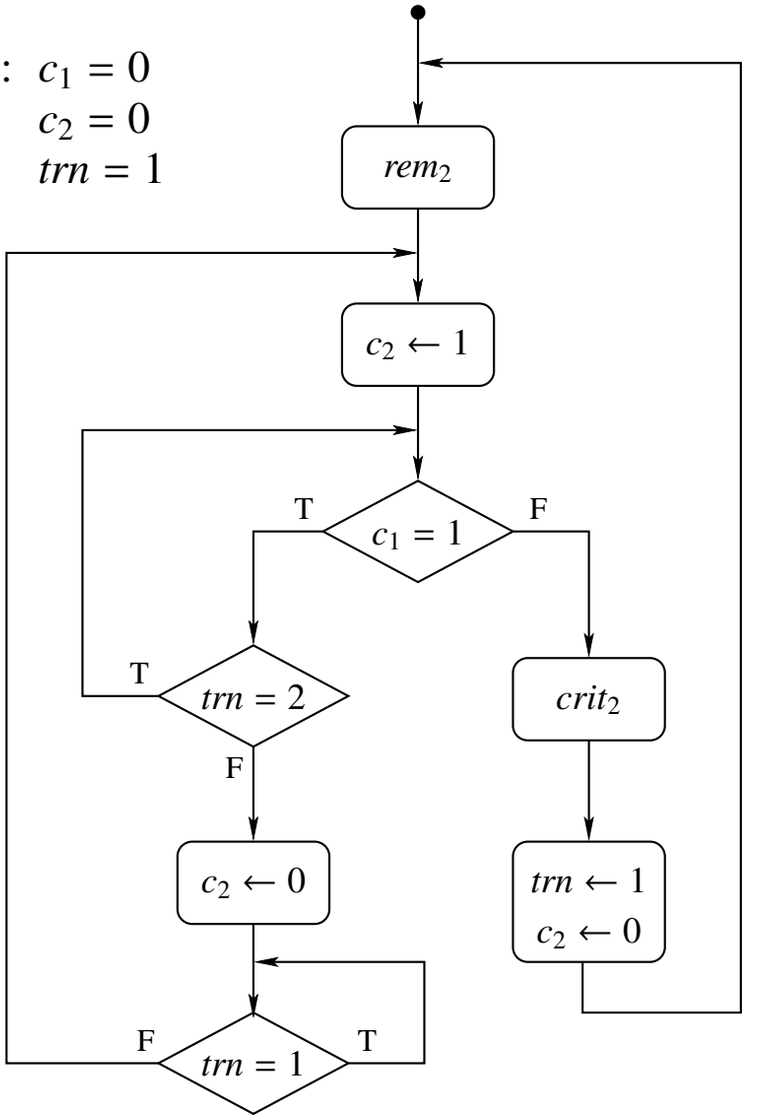
Exercise 2

Add a LTL property to the model obtained for the previous exercise, and verify it.

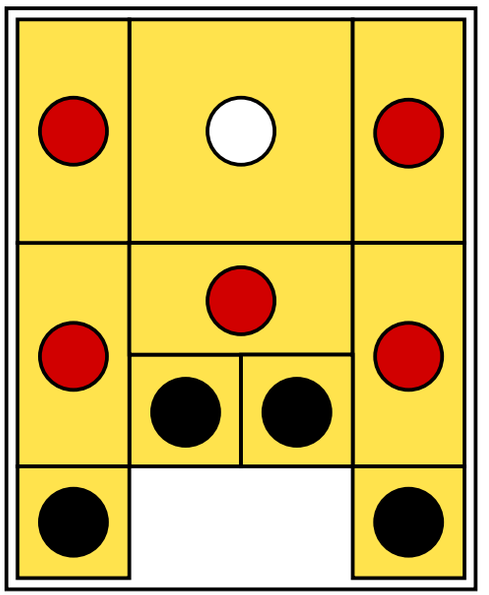
Exercise 3: Model and verify Dekker's algorithm



Initially: $c_1 = 0$
$c_2 = 0$
$trn = 1$



Exercise 4: Klotski



The goal is to slide the blocks so as to move the white one to the bottom position.

Could you find a solution with the help of Spin?

Guidelines:

- Each block can be modeled by a separate process, trying non-deterministically all possible moves.
- The simplest approach is to define one `proctype` for each block shape.
- Two-dimensional arrays cannot be directly defined in Promela. To represent the free cells on the board, you can
  - define a `typedef` corresponding to a row (as an array of Booleans), and
  - represent the board as an array of rows.
- You will probably have to modify the default parameters (suggestions: 2 GB, 5.10^6 states, use collapse compression).

References

- R. Gerth, *Concise Promela reference*, Eindhoven University, 1997.
- *Promela language reference*, `http://spinroot.com/spin/Man/promela.html`.
- *Basic Spin Manual*, `http://spinroot.com/spin/Man/Manual.html`.
- Th. C. Ruys, *SPIN beginner's tutorial*, Proc. SPIN Workshop, 2002.