Computer Aided Verification 2015
The SPIN model checker

Grigory Fedyukovich
<grigory.fedyukovich@usi.ch>

Universita’ della Svizzera Italiana

March 11, 2015

Material borrowed from Roberto Bruttomesso
Outline

1 Introduction

2 PROcess MEta LAnguage
   ■ Data types
   ■ Control structures
   ■ Channels
Outline

1 Introduction

2 PROCess MEta LAnguage
   - Data types
   - Control structures
   - Channels
Spin

Introduction

- **Explicit-state** model checker
  - works on-the-fly (no need to represent the entire structure of the system upfront)

- Targeted toward the verification of **concurrent** systems
  - real world situations involving multiple agents acting at the same time on the same resources
  - hardware or software with interacting components

- Input language: **PROMELA** (PRocess MEta LAnguage)
  - Designed for describing processes
  - Different from an imperative language

- Properties: **LTL** (Linear Temporal Logic)
Bugs in concurrent systems (1)

Circular blocking

[Diagram showing a circular blocking scenario]
We have two global resources \( p \) (a printer) and \( s \) (a scanner), and two concurrent processes running in memory.

**process A**
1. `getPrinter(p)`
2. `getScanner(s)`
3. `...`
4. `releasePrinter()`
5. `releaseScanner()`
end

**process B**
1. `getScanner(s)`
2. `getPrinter(p)`
3. `...`
4. `releasePrinter()`
5. `releaseScanner()`
end

The sequence A1, B1, A2, B2 generates a deadlock.
In Spin we can specify processes, which are assumed to run \textbf{concurrently}.

Since we are dealing with multiple processes, at any point in time a number of instructions can be executed (unless deadlock).

Spin chooses \textbf{non-deterministically} one instruction to be executed among the ones available.
Spin
Two modes

- Simulator (first half): runs a guided or a random simulation of the model defined by the user. Can be used to quickly check the behavior of a model

- Verifier (second half): generates a C program, optimized for performance, that **exhaustively** checks the validity of the property (or the absence of deadlocks)
Outline

1 Introduction

2 PROcess MEta LAnguage
   - Data types
   - Control structures
   - Channels
## Data types

### Basic types

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>[0,1]</td>
<td>bit flag = 0;</td>
</tr>
<tr>
<td>bool</td>
<td>false, true</td>
<td>bool flag = false;</td>
</tr>
<tr>
<td>byte</td>
<td>[0,255]</td>
<td>byte b = 0;</td>
</tr>
<tr>
<td>chan</td>
<td>[1,255]</td>
<td>see later ...</td>
</tr>
<tr>
<td>mtype</td>
<td>[1,255]</td>
<td>see later ...</td>
</tr>
<tr>
<td>pid</td>
<td>[0,255]</td>
<td>pid p = 0;</td>
</tr>
<tr>
<td>short</td>
<td>([-2^{15},2^{15} - 1])</td>
<td>short s = 0;</td>
</tr>
<tr>
<td>int</td>
<td>([-2^{31},2^{31} - 1])</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>unsigned</td>
<td>([0,2^n - 1])</td>
<td>unsigned u : 8;</td>
</tr>
</tbody>
</table>

Default value for uninitialized variables is “0”
Data types

**Compound types**

- **Arrays (one dimension):**
  ```c
  byte vec[10];
  ```

- **Records:**
  ```c
  typedef rec
  {
    bit a;
    byte b;
  }
  ```

- **mtype**: used to declare user-defined constants (similar to C `enum`)
  Example:
  ```c
  mtype = \{ ack, nack, error \};
  ```
Control structures

Processes

- In Promela we can only define processes
- `init` is a process which is automatically started at the beginning
- `proctype` can be used to define a process
- A process can be started with `run`
- `active` can be used to specify a `proctype` that is automatically started at the beginning
Control structures

Basic Statements

- Assignments: \texttt{count = count + 1}
  - \textit{Always} executable: \texttt{count} is updated

- Conditions: \texttt{visitors == 10}
  - Executable \textbf{only if} they hold. Nothing is changed.

- Example: \texttt{busy.pml}

- Statements are separated by \texttt{;}
  - \texttt{x = x + 2; y = x + z}

- or by a \texttt{->} (usually when condition \texttt{->} assignment)
  - \texttt{x == 0 ; y = x + z}
  - \texttt{x == 0 -> y = x + z}
Control structures
Case selection

if
:: cond_1 -> list of statements 1
:: cond_2 -> list of statements 2
:: ...
fi

- Only **one** option from the list will be executed
- It is chosen randomly among the satisfied conditions
- If no condition holds, the process blocks until a condition becomes true
- Example **if.pml**
Control structures
Repetition

\[
\text{do} \\
:: \ \text{cond}_1 \rightarrow \text{list of statements 1} \\
:: \ \text{cond}_2 \rightarrow \text{list of statements 2} \\
:: \ ... \\
\text{od}
\]

- Similar to \textbf{if}, but repeated over time
- Only \textbf{one} option from the list will be executed
- It is chosen randomly among the satisfied conditions
- If no condition holds, the process blocks until a condition becomes \textbf{true}
- Example \texttt{do.pml}
Control structures
Other useful operators

- **else**: in *if-fi* or *do-od* statements become true if any other condition fails
- **goto ...**: can be used to jump to another process location
- **break**: force exit from a *do-od* loop
- **timeout**: becomes true if no other statement can be executed
- **atomic { ... }**: process a list of statements without interleaving
- **skip**: does nothing
int iter;

proctype counter( byte count )
{
    do
        :: (count != 0) ->
            if
                :: count = count + 1
                :: count = count - 1
            fi
        :: (count == 0) -> break
    od;
    printf( "Iterations done: %d" iter );
}
Exercise 1
Collatz Conjecture (3x+1 Problem)

Let $n$ be a natural number. Consider the following loop:

1. while $n \neq 1$
2. if $n$ is even then $n := n/2$
3. else $n := 3 \cdot n + 1$

Example: $n = 5$ generates the sequence, 5, 16, 8, 4, 2, 1

Exercise:

Implement the loop in PROMELA and return as output the number of iterations done

(use “%” to compute the remainder)
Exercise 1
Collatz Conjecture (3x+1 Problem)

Let $n$ be a natural number. Consider the following loop:

1. while $n \neq 1$
2. if $n$ is even then $n := n/2$
3. else $n := 3 \cdot n + 1$

Example: $n = 5$ generates the sequence, 5, 16, 8, 4, 2, 1

Exercise:

Implement the loop in PROMELA and return as output the number of iterations done

extra Write a routine that computes the number between 1 and 100 that takes the highest number of steps

(use “%” to compute the remainder)
Data can be exchanged via global variables (as seen so far) or using **channels**

- a channel is FIFO queue

Example declaration:

```
chan c = [2] of { byte }
```

**Operations:**

- **send** an element `e` into the channel `c`
  - Syntax: `c!e`
  - Executable if the channel is not full

- **receive** an element from the channel `c`, and store it into `e`
  - Syntax: `c?e`
  - Executable if the channel is not empty

- Example **prod_cons.pml**
Empty channels can be declared to define rendez-vous points

```plaintext
c chan buf = [0] of { mtype };
mtype { ack, nack };
```
Modify `prod_cons.pml` as follows:

- `producer()` sends odd and even numbers into the buffer
- `write consumer_odd()` that prints only odd numbers
- `write consumer_even()` that prints only even numbers
- `producer()` should send termination signal to both consumers
- every number must be printed