Rigorous Methods for Software Engineering (F21RS-F20RS) Promela (Part 1)

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- Basic building blocks of **Promela** programs.
- Executing **Promela** programs via **iSpin**
- Structured data types.
- Process definition, instantiation & execution.
- Concurrency and **Promela** programs.
- Non-deterministic behaviour & basic synchronization.

Note: **Spin** stands for **S**imple **P**romela **In**terpreter

Promela Programs

- The basic building blocks of **Promela** programs are:
  - processes
  - channels
  - variables
- Processes model the behaviour of components of a system and are by definition **global** objects.
- Channels and variables define the environment in which processes exist and can be either **local** of **global**.

Process Types

- A **Promela** program is represented by a set of process declarations.
- A process declaration begins with the keyword `proctype` and contains:
  - process identifier;
  - formal parameter list (optional);
  - sequence of local variable declarations & statements
- Syntactically a process declaration has the following form:

```
proctype name( /* formal parameter list */ )
{
   /* local declarations and statements */
}
```

Note: `/*` and `*/` delimit comments in **Promela**.

active proctype

- To instantiate a `proctype` that has no parameters, the keyword `active` can be used:

```
active proctype name()
{
   /* local declarations and statements */
}
```

  Note that the empty paramember list is denotes by `()`.

- Multiple instances of the same `proctype` declaration can be generated using an optional array suffix, *i.e.*

```
active [N] proctype name()
{
   /* local declarations and statements */
}
```

  where `N` must denote a positive number.

A Simple Example – `Hello World` Revisited with a Twist!

- A single instance version:

```
active proctype hello(){ printf("Hello\n") }
active proctype world(){ printf("World\n") }
```

- A multiple instances version:

```
active [4] proctype hello(){ printf("Hello\n") }
active [2] proctype world(){ printf("World\n") }
```

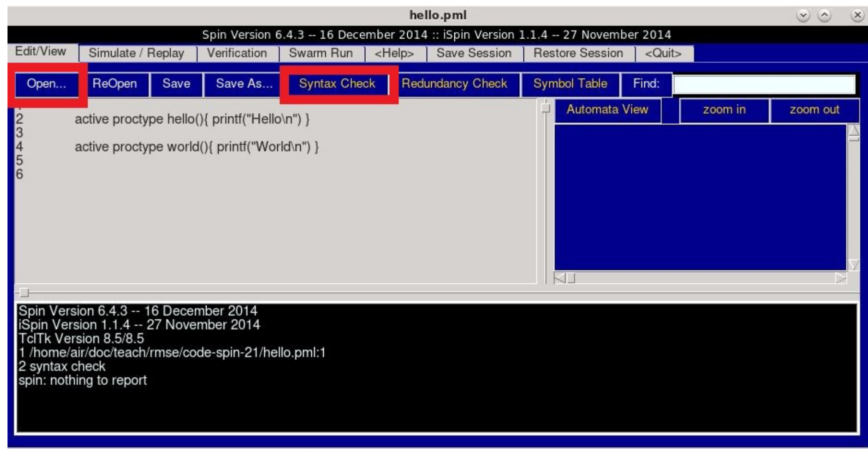
  Note: the above will generate 4 instances of `hello` and 2 instances of `world`.

- All processes implicitly have a variable `_pid`.
- Each time a process is instantiated the value associated with its `_pid` is one more than the last, i.e. the first process has its `_pid` set to 0, the second to 1, etc.

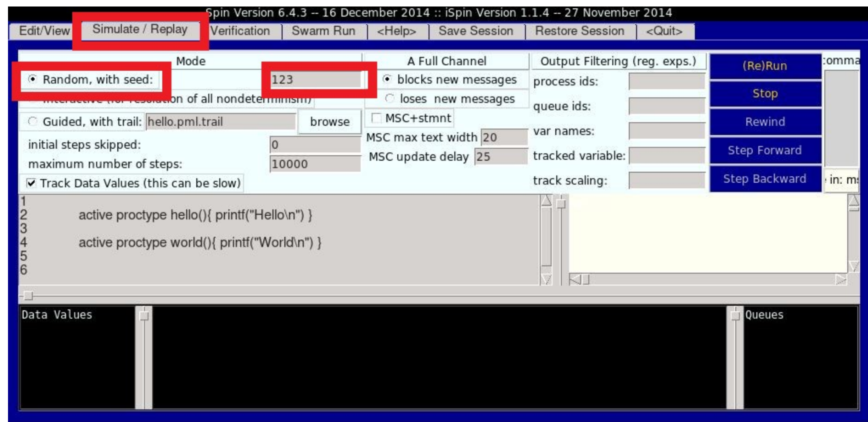
Executing a Promela Program

- The execution of a **Promela** program is called a **simulation**.
- **Spin** supports three simulation modes, first we consider **random simulation**, i.e. within the **Simulate/Replay** tab of **iSpin** select **Random**.
- Technically, it is **pseudo random simulation**, i.e. it requires a seed value, and for a given seed value the same behaviour will be exhibited by your program – which is what you want when debugging!

Hello World



Hello World



Within **Simulate/Replay** tab select **Random** – note that the default seed value is 123.

Hello World

hello.pml

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View | Simulate / Replay | Verification | Swarm Run | <Help> | Save Session | Restore Session | <Quit>

Mode
- Random, with seed: 123
- Interactive (for resolution of all nondeterminism)
- Guided, with trail: hello.pml.trail browse

initial steps skipped: 0
maximum number of steps: 10000
Track Data Values (this can be slow)

A Full Channel
- blocks new messages
- loses new messages
- MSC+stmnt

MSC max text width 20
MSC update delay 25

Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:

(Re)Run | Stop | Rewind | Step Forward | Step Backward

command
pin -p -s -r -X -v -n123 -l -g -u1000 0 hello.pml
ave in: msc.p

```
1
2       active proctype hello(){ printf("Hello\n") }
3
4       active proctype world(){ printf("World\n") }
5
6
```

Simulation Panel

```
0:    proc  - (:root:) creates proc  0 (hello)
      proc  - (:root:) creates proc  1 (world)
Hello
      proc  0 (hello:1) hello.pml:2 (state 1) [printf('Hello\\n')]
World
      proc  1 (world:1) hello.pml:4 (state 1) [printf('World\\n')]
2:    proc  1 (world:1) terminates
2:    proc  0 (hello:1) terminates
2 processes created
```

Queues

Press the **(Re)Run** button.

Hello World

hello.pml

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View | Simulate / Replay | Verification | Swarm Run | <Help> | Save Session | Restore Session | <Quit>

Mode: Random, with seed: 42

```
1
2    active proctype hello(){ printf("Hello\n") }
3
4    active proctype world(){ printf("World\n") }
5
6
```

```
0:    proc  - (:root:) creates proc  0 (hello)
      proc  - (:root:) creates proc  1 (world)
World
      proc  1 (world:1) hello.pml:4 (state 1) [printf('World\\n')]
Hello
      proc  0 (hello:1) hello.pml:2 (state 1) [printf('Hello\\n')]
2:    proc  1 (world:1) terminates
2:    proc  0 (hello:1) terminates
2 processes created
```

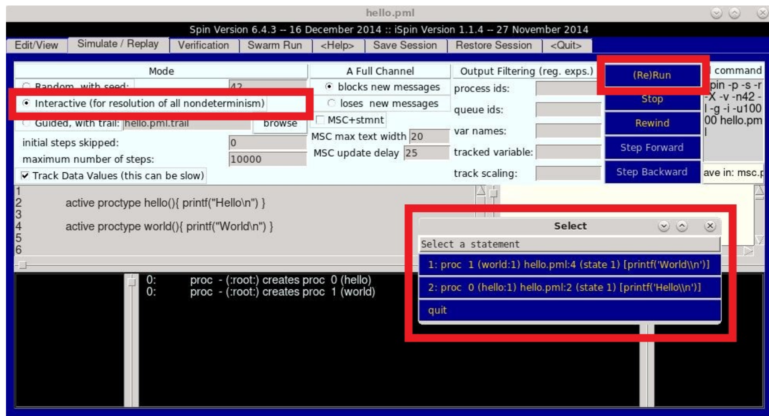
Here the seed value is changed to 42.

Deterministic & Non-Deterministic Behaviour

This `Hello World` program is an exhibits **non-deterministic behaviour**.

- **Deterministic behaviour:** a process is deterministic if for a given start state it behaves in exactly the same way if supplied with the same stimuli from its environment.
- **Non-deterministic behaviour:** a process is non-deterministic if it need not always behave in exactly the same way each time it executes from a given start state with the same stimuli from its environment.

Hello World



- The execution of a **Promela** program is called a **simulation**.
- **Spin** also supports **interactive simulation**, i.e. within **iSpin** select **Interactive** option.
- In **Interactive** option, every non-deterministic choice point within a simulation is presented to the user – puts you back into control!

Executability of Statements

- **Promela** does not make a distinction between a **condition** and a **statement**, *e.g.* the simple boolean condition `a == b` represents a statement in **Promela**.
- **Promela** statements are either **executable** or **blocked**. The execution of a statement is conditional on it not being blocked.
- **Promela**'s notion of statement executability provides the basic means by which process synchronization can be achieved.

```
while (a != b) skip /* conventional busy wait */

(a == b)            /* {\bf Promela} equivalent */
```

Variables and Basic Data Types

- **Promela** variables provide the means of storing information about the system being modelled.
- A variable may hold global information on the system or information that is local to a particular component (process).
- **Promela** supports five basic data types:

Name	Size (bits)	Usage	Range
`bit`	1	unsigned	$0 \dots 1$
`bool`	1	unsigned	$0 \dots 1$
`byte`	8	unsigned	$0 \dots 255$
`short`	16	signed	$-2^{15} - 1 \dots 2^{15} - 1$
`int`	32	signed	$-2^{31} - 1 \dots 2^{31} - 1$

Variable Declarations

- Like all well-structured programming languages, **Promela** requires that variables must be declared before they can be used.
- Variable declarations follow the style of the C programming language, *i.e.* a basic data type followed by one or more identifiers and optional initializer:
  ```
  byte count, total = 0;
  ```
  An initializer must be an expression of the appropriate basic type.
- By default all variables of the basic types are initialized to 0. Note that as in C, 0 (zero) is interpreted as false while any non-zero value is interpreted as true.

Structured Data Types

- Arrays – an array type is declared as follows:
  ```
  int table[max]
  ```
  Note that this generates an array of `max-1` integers, *i.e.* `table[0], table[1], ...  table[max-1]`
- Enumerated Types – a set of symbolic constants is declared as follows:
  ```
  mtype = {LINE_CLEAR, TRAIN_ON_LINE, LINE_BLOCKED}
  ```
  Note: a program can only contain one `mtype` declaration which must be global.
- Structures – a record data type is declared as follows:
  ```
  typdef msg {byte data[4], byte checksum}
  ```
  Note: Structure access is as in C:
  ```
  msg message; ... message.data[0]
  ```

Identifiers, Constants & Expressions

- Identifiers: An identifier is a single letter, a period symbol, or underscore followed by zero or more letters, digits, periods or underscores.
- Constants: A constant is a sequence of digits that represents a decimal integer. Symbolic constants can be defined by means of `mtype` or via a C-style macro definition, *e.g.*

  ```
  #define MAX 999
  ```

- ...

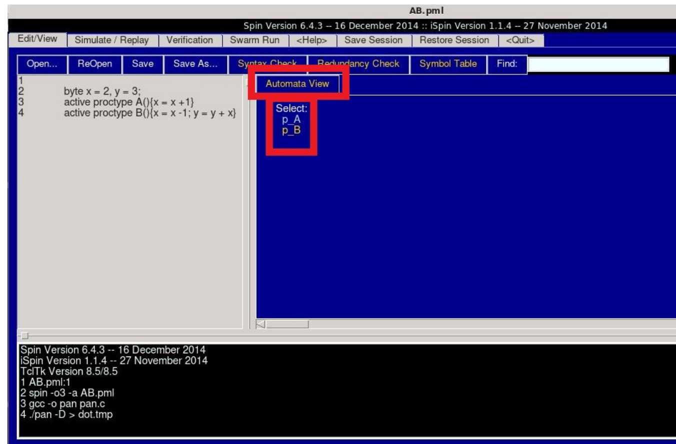
Identifiers, Constants & Expressions

- Expressions: An expression is built up from variables, identifiers and constants using the following operators:

```
+, -, *, /, %, --, ++,   arith
>, >=, <, <=, ==, !=,    relational
&&, ||, !,               logicals
&, |, ~, ^, >>, <<,      bits
!, ?,                    channels
(), [],                  group/index
```

Processes as Automata

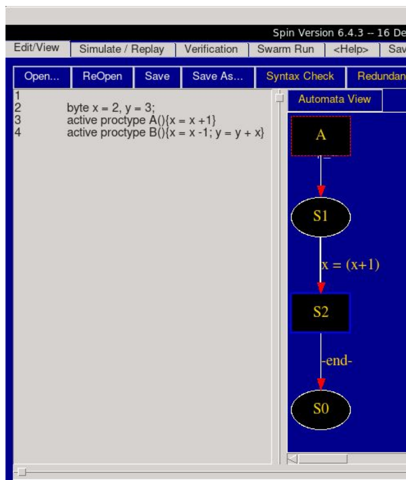
```
1. byte x = 2, y = 3;
2. active proctype A(){x = x + 1}
3. active proctype B(){x = x - 1; y = y + x}
```



- Press the **Automata View** button then select a process ID.

Processes as Automata

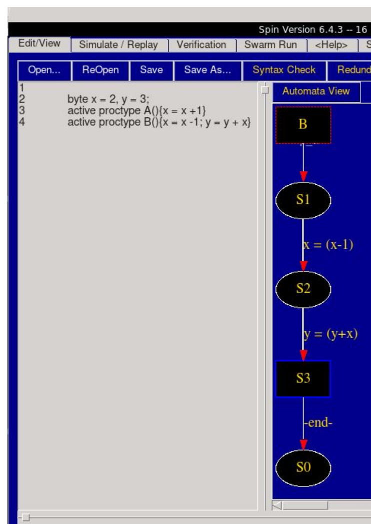
```
1. byte x = 2, y = 3;
2. active proctype A(){x = x + 1}
3. active proctype B(){x = x - 1; y = y + x}
```



- Each process is represented as a distinct **automata**, i.e. nodes and directed edges.
- Nodes denote **states** and directed edges denote **state transitions**.
- Each state transition is labeled with the associated statement in the code.

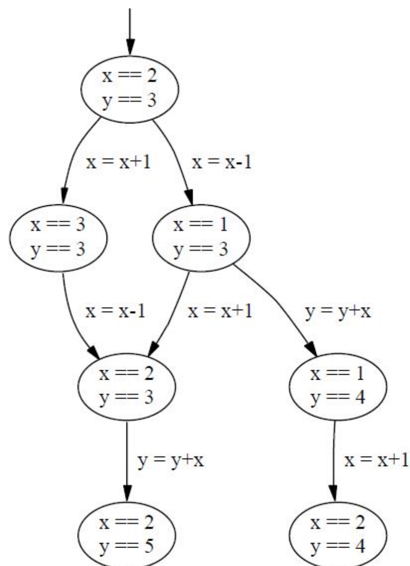
Processes as Automata

```
1. byte x = 2, y = 3;
2. active proctype A(){x = x + 1}
3. active proctype B(){x = x - 1; y = y + x}
```



- Each process is represented as a distinct **automata**, i.e. nodes and directed edges.
- Nodes denote **states** and directed edges denote **state transitions**.
- Each state transition is labeled with the associated statement in the code.

Concurrency via Interleaving



- Given a **Promela** program **Spin** can explore all reachable states, i.e. all possible interleaving of atomic statements.
- The reachable states are visualized opposite as s **state transition diagram**.
- Nodes denote reachable **states**.
- Directed edges exist between nodes if and only if there exists statement that performs the associated **state transition**.

More Non-Deterministic Behaviour

- Consider the following two process system:

  ```
  byte state = 1;
  active proctype P()
     { (state == 1) -> state = state + 1 }
  active proctype Q()
     { (state == 1) -> state = state - 1 }
  ```

  note that `S1 -> S2` and `S1; S2` are equivalent.
- Note that if process `P` (or `Q`) terminates before process `Q` (or `P`) begins execution then `Q` (or `P`) will be blocked forever on the condition, *i.e.* `(state == 1)`.
- Note that if both `P` and `Q` execute their respective conditions (i.e. `(state == 1)`) then both processes will terminate and the final value of `state` will be equal to `1`.
- But in general the final value of `state` is unpredictable, *i.e.* it can be either 0, 1 or 2.

Atomic Sequences

- **Promela** provides another means of avoiding the undesirable interleaving problem illustrated above via the `atomic` operator.
- Consider the following refinement to the two process system:

```
byte state = 1;
active proctype P()
   { atomic{ (state == 1) -> state = state + 1 } }
active proctype Q()
   { atomic{ (state == 1) -> state = state - 1 } }
```

- The final value of the global variable `state` will be either 2 or 0, depending upon which process executes.
- Note that an atomic sequence restricts the level of interleaving so reduces the complexity when it comes to validating a **Promela** model.

Summary

Learning outcomes:

- To be able to understand and construction simple **Promela** programs exploiting both local and global data objects;
- To understand the **Promela** model for concurrent process execution;
- To be able to model synchronous behaviour between processes;

Recommended reading:

- **Spin homepage:** `http://spinroot.com`
- **Inspiring applications of Spin:** `http://Spinroot.com/spin/success.html`