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

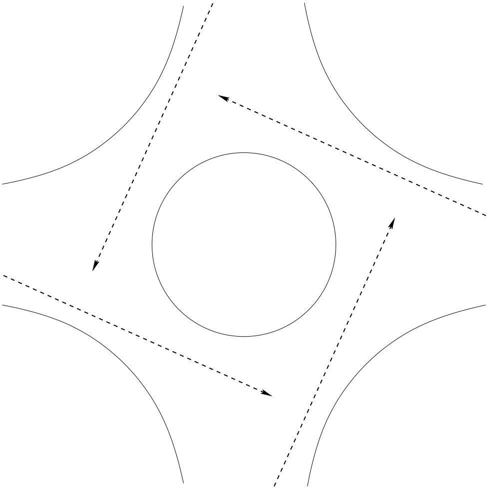
Outline

Outline

- **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



Bugs in concurrent systems (2)

Deadly embrace

We have two global resources p (a printer) and s (a scanner), and two concurrent processes running in memory

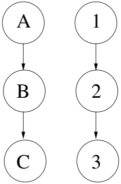
process A	**process B**
1 $getPrinter(p)$	1 $getScanner(s)$
2 $getScanner(s)$	2 $getPrinter(p)$
3 $\ldots$	3 $\ldots$
4 $releasePrinter()$	4 $releasePrinter()$
5 $releaseScanner()$	5 $releaseScanner()$
end	**end**

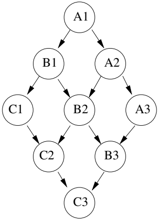
The sequence A1, B1, A2, B2 generates a deadlock

Spin

Execution semantic

- In Spin we can specify processes, which are assumed to run **concurrently**
- Since we are dealing with multiple processes, at any point in time a number of instructions can be executed (unless deadlock)
- Spin chooses **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 perfomance, that **exhaustively** checks the validity of the property (or the absence of deadlocks)

Outline

Data types

Basic types

Type	Range	Example
`bit`	[0,1]	`bit flag = 0;`
`bool`	`false, true`	`bool flag = false;`
`byte`	[0,255]	`byte b = 0;`
`chan`	[1,255]	see later ...
`mtype`	[1,255]	see later ...
`pid`	[0,255]	`pid p = 0;`
`short`	[-2^{15},$2^{15}-1$]	`short s = 0;`
`int`	[-2^{31},$2^{31}-1$]	`int i = 0;`
`unsigned`	[0,2^n-1]	`unsigned u : 8;`

Default value for uninitialized variables is “0”

Data types

Compound types

- Arrays (one dimension):

  ```
  byte vec[10];
  ```

- Records:

  ```
  typedef rec
  {
    bit a;
    byte b;
  }
  ```

- `mtype`: used to declare user-defined constants (similar to C `enum`)

  Example:

  ```
  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: `count = count + 1`
  - **Always** executable: `count` is updated
- Conditions: `visitors == 10`
  - Executable **only if** they hold. Nothing is changed.
- Example: `busy.pml`
- Statements are separated by `;`
  - `x = x + 2; y = x + z`
- or by a `->` (usually when `condition -> assignment`)
  - `x == 0 ; y = x + z`
  - `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

```
do
::  cond_1 -> list of statements 1
::  cond_2 -> list of statements 2
::  ...
od
```

- Similar to `if`, but repeated over time
- 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 `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

Example

```
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:

$$\begin{array}{l} \text{while } n \neq 1 \\ \quad \text{if } n \text{ is even then } n := n/2 \\ \quad \text{else } n := 3 \cdot n + 1 \end{array}$$

- 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:

while $n \neq 1$
  if n is even then $n := n/2$
  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)

Communication

Channels (1)

- 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:

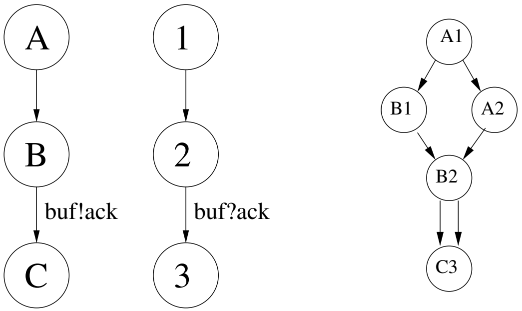
- **send** an element `e` into the channel `c`
  - Syntax: `c!e`
  - Executable if the channel is not full
- **receive** an element from the channel `c`, and store it into `e`
  - Syntax: `c?e`
  - Executable if the channel is not empty
- Example `prod_cons.pml`

Communication

Channels (2)

- **Empty** channels can be declared to define **rendez-vous** points

```
chan buf = [0] of { mtype };
mtype { ack, nack };
```



Exercise 2

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