Software Engineering using Formal Methods

Introduction to PROMELA

Wolfgang Ahrendt

04 September 2014

Towards Model Checking

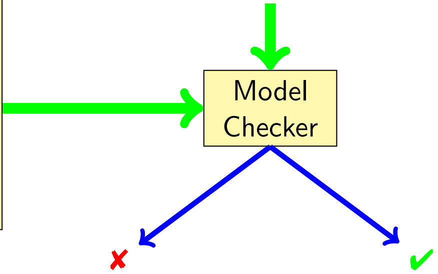
System Model

Promela Program

```
byte n = 0;
active proctype P() {
  ...
}
active proctype Q() {
  ...
}
```

System Property

P, Q are never in their
critical section at the same time



criticalSectP = 0 1 1
criticalSectQ = 1 0 1

What is PROMELA?

PROMELA is an acronym

PROcess MEta-LAnguage

PROMELA is a language for modelingmodeling concurrentconcurrent systems

- multi-threaded
- synchronisation and message passing
- few control structures, pure (side-effect free) expressions
- data structures with finite and fixed bound

What is PROMELA **Not**?

PROMELA is not a programming language

Very small language, not intended to program real systems
(we will master most of it in today's lecture!)

- No pointers
- No methods/procedures
- No libraries
- No GUI, no standard input
- No floating point types
- No data encapsulation
- Non-deterministic

A First PROMELA Program

```
active proctype P() {
  printf("Hello␣world\n")
}
```

Command Line Execution

Simulating (i.e., interpreting) a PROMELA program

```
> spin hello.pml
      Hello world
1 process created
```

First observations

- keyword **proctype** declares process named `P`
- C-like command and expression syntax
- C-like (simplified) formatted print

Arithmetic Data Types

```
active proctype P() {
   int val = 123;
   int rev;
   rev = (val % 10) * 100 + /* % is modulo */
         ((val / 10) % 10) * 10 + (val / 100);
   printf("val␣=␣%d,␣rev␣=␣%d\n", val, rev)
}
```

Observations

- Data types **byte**, **short**, **int**, **unsigned** with operations +,−,∗,/,%
- Expressions computed as **int**, then converted to container type
- No floats, no side effects, C/Java-style comments
- No string variables (strings only in print statements)

Booleans and Enumerations

```
bit  b1 = 0;
bool b2 = true;
```

Observations

- **bit** is small numeric type containing 0, 1
- **bool**, **true**, **false** syntactic sugar for **bit**, 1, 0

Enumerations

```
mtype = { red, yellow, green } //in global context

active proctype P() {
  mtype light = green;
  printf("the␣light␣is␣%e\n", light)
}
```

Observations

- literals represented as non-0 **byte**: at most 255
- **mtype** stands for message type (first used for message names)
- There is at most one **mtype** per program
- `%e` "prints" mtype constant

Control Statements

Sequence	using `;` as *separator* (not terminator like in C/Java)
Guarded Command:	
— Selection	non-deterministic choice of an alternative
— Repetition	loop until `break` (or forever)
Goto	jump to a label

Guarded Commands: Selection

```
active proctype P() {
  byte a = 5, b = 5;
  byte max, branch;
  if
    :: a >= b -> max = a; branch = 1
    :: a <= b -> max = b; branch = 2
  fi
}
```

Command Line Execution

Trace of random simulation of multiple runs

```
> spin -v max.pml
> spin -v max.pml
> ...
```

Observations

- Guards may “overlap” (more than one can be true at the same time)
- Any alternative whose guard is true is randomly selected

Guarded Commands: Selection Cont'd

```
active proctype P() {
  bool p = ...;
  if
    :: p    -> ...
    :: true -> ...
  fi
}
```

Second alternative can be selected anytime, regardless of whether p is true

```
active proctype P() {
  bool p = ...;
  if
    :: p    -> ...
    :: else -> ...
  fi
}
```

Second alternative can be selected only if p is false

Guarded Statement Syntax

```
:: guard -> command
```

Observations

- `->` is synonym for `;`
- Therefore: can use `;` instead of `->`
  (Relation guards vs. statements will get clearer later)
- First statement after `::` used as guard
- `-> command` can be omitted
- (`->` overloaded, see conditional expressions)

Guarded Commands: Repetition

```
active proctype P() { /* computes gcd */
  int a = 15, b = 20;
  do
    :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
  od
}
```

Command Line Execution

Trace with values of local variables

```
> spin -p -l gcd.pml
> spin --help
```

Observations

- Any alternative whose guard is true is randomly selected
- Only way to exit loop is via **break** or **goto**

Counting Loops

Counting loops such as for-loops as usual in imperative programming languages are realized with **break** after the termination condition:

```
#define N 10 /* C-style preprocessing */
active proctype P() {
  int sum = 0; byte i = 1;
  do
    :: i > N -> break              /* test */
    :: else -> sum = sum + i; i++ /* body,increase */
  od
}
```

Observations

- Don't forget else, otherwise strange behaviour

Arrays

```
#define N 5
active proctype P() {
  byte a[N];
  a[0] = 0;a[1] = 10;a[2] = 20;a[3] = 30;a[4] = 40;
  byte sum = 0, i = 0;
  do
    :: i > N-1 -> break
    :: else     -> sum = sum + a[i]; i++
  od;
}
```

Observations

- Array indexes start with 0 as in JAVA and C
- Arrays are scalar types: `a` and `b` always different arrays
- Array bounds are constant and cannot be changed
- Only one-dimensional arrays (there is an ugly workaround)

Record Types

```
typedef DATE {
  byte day, month, year;
}
active proctype P() {
  DATE D;
  D.day = 1; D.month = 7; D.year = 62
}
```

Observations

- may include previously declared record types, but no self-references
- Can be used to realize multi-dimensional arrays:

```
typedef VECTOR {
  int vector[10]
};
VECTOR matrix[5]; /* base type array in record */
matrix[3].vector[6] = 17;
```

Jumps

```
#define N 10
active proctype P() {
   int sum = 0; byte i = 1;
   do
     :: i > N -> goto exitloop
     :: else -> sum = sum + i; i++
   od;
exitloop:
   printf("End␣of␣loop")
}
```

Observations

- Jumps allowed only within a process
- Labels must be unique for a process
- Can't place labels in front of guards (inside alternative ok)
- Easy to write messy code with **goto**

Inlining Code

PROMELA has no method or procedure calls

```
typedef DATE {
  byte day, month, year;
}
inline setDate(D, DD, MM, YY) {
  D.day = DD; D.month = MM; D.year = YY
}
active proctype P() {
  DATE d;
  setDate(d,1,7,62)
}
```

The inline construct

- macro-like abbreviation mechanism for code that occurs multiply
- creates no new scope for locally declared variables
  - avoid to declare variables in **inline** — they are visible

Non-Deterministic Programs

Deterministic PROMELA programs are trivial

Assume PROMELA program with one process and no overlapping guards

- All variables are (implicitly or explictly) initialized
- No user input possible
- Each state is either blocking or has exactly one successor state

Such a program has exactly one possible computation!

Non-trivial PROMELA programs are non-deterministic!

Possible sources of non-determinism

1. Non-deterministic choice of alternatives with overlapping guards
2. Scheduling of concurrent processes

Non-Deterministic Generation of Values

```
byte x;
if
  :: x = 1
  :: x = 2
  :: x = 3
  :: x = 4
fi
```

Observations

- assignment statement used as guard
  - assignment statements (here used as guards) always succeed
  - side effect of guard is desired effect of this alternative
- selects non-deterministically a value in $\{1, 2, 3, 4\}$ for x

Non-Deterministic Generation of Values Cont'd

Generation of values from explicit list impractical for large range

```
#define LOW  0
#define HIGH 9
byte range = LOW;
do
  :: range < HIGH -> range++
  :: break
od
```

Observations

- In each iteration, equal chance for increase of `range` and loop exit
- Chance of generating n in random simulation is $2^{-(n+1)}$
  - Obtain no representative test cases from random simulation!
  - OK for verification, because all computations are generated

Sources of Non-Determinism

1. Non-deterministic choice of alternatives with overlapping guards
2. Scheduling of concurrent processes

Concurrent Processes

```
active proctype P() {
  printf("Process␣P,␣statement␣1\n");
  printf("Process␣P,␣statement␣2\n")
}

active proctype Q() {
  printf("Process␣Q,␣statement␣1\n");
  printf("Process␣Q,␣statement␣2\n")
}
```

Observations

- Can declare more than one process (need unique identifier)
- At most 255 processes

Execution of Concurrent Processes

Command Line Execution

Random simulation of two processes

```
> spin interleave.pml
```

Observations

- Scheduling of concurrent processes 'on one processor'
- Scheduler randomly selects process to make next step
- Many different computations are possible: non-determinism
- Use `-p`/`-g`/`-l` options to see more execution details

Sets of Processes

```
active [2] proctype P() {
  printf("Process␣%d,␣statement␣1\n", _pid);
  printf("Process␣%d,␣statement␣2\n", _pid)
}
```

Observations

- Can declare set of identical processes
- Current process identified with reserved variable `_pid`
- Each process can have its own local variables

Command Line Execution

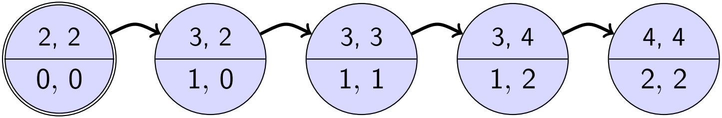
Random simulation of set of two processes

```
> spin interleave_set.pml
```

PROMELA **Computations**

```
1 active [2] proctype P() {
2   byte n;
3   n = 1;
4   n = 2
5 }
```

One possible computation of this program



Notation

- Program pointer (line #) for each process in upper compartment
- Value of all variables in lower compartment

Computations are either infinite or terminating or blocking

Admissible Computations: Interleaving

Definition (Interleaving of independent computations)

Assume n independent processes $P_1, \ldots, P_n$ and process i has computation $c^i = (s^i_0, s^i_1, s^i_2, \ldots)$.

The computation $(s_0, s_1, s_2, \ldots)$ is an interleaving of $c^1, \ldots, c^n$ iff

- for all $j \in \mathbb{N}$ there exist $i \in \{1, \ldots, n\}$ and $j' \in \mathbb{N}$ such that $s_j = s^i_{j'}$ (no junk)
- for all j, k with $j < k$, if $s_j = s^i_{j'}$ and $s_k = s^i_{k'}$ (for some $i \in \{1, \ldots, n\}$, $j', k' \in \mathbb{N}$), then $j' < k'$ (order preserving)

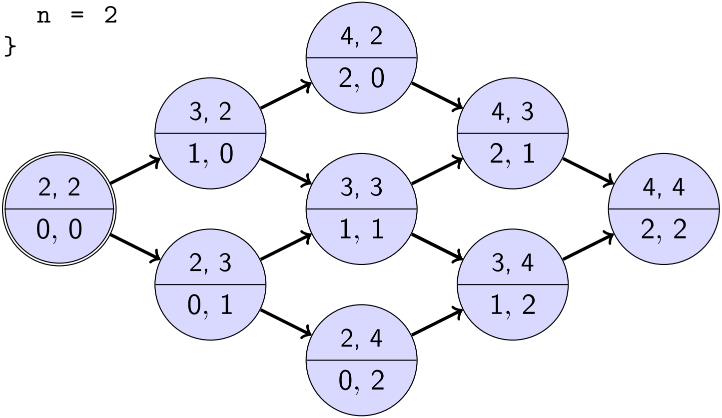
Observations

- Semantics of concurrent PROMELA program is the set of its interleavings
- Not universal: in JAVA certain reorderings allowed

Interleaving Cont'd

Can represent possible interleavings in a DAG

```
active [2] proctype P() {
  byte n;
  n = 1;
  n = 2
}
```



Atomicity

At which granularity of execution can interleaving occur?

Definition (Atomicity)

An expression or statement of a process that is executed entirely without the possibility of interleaving is called atomic.

Atomicity in PROMELA

- Assignments, jumps, skip, and expressions are atomic
  - In particular, conditional expressions are atomic:
    `(p -> q : r)`, C-style syntax, brackets required
- Guarded commands are not atomic

Atomicity Cont'd

```
int a,b,c;
active proctype P() {
  a = 1; b = 1; c = 1;
  if
    :: a != 0 -> c = b / a
    :: else -> c = b
  fi
}
active proctype Q() {
  a = 0
}
```

Command Line Execution

Interleaving into selection statement forced by interactive simulation

```
> spin -p -g -i zero.pml
```

Atomicity Cont'd

How to prevent interleaving?

1. Consider to use expression instead of selection statement:

   ```
   c = (a != 0 -> (b / a): b)
   ```

2. Put code inside scope of **atomic**:

   ```
   active proctype P() {
     a = 1; b = 1; c = 1;
     atomic {
     if
       :: a != 0 -> c = b / a
       :: else -> c = b
     fi
     }
   }
   ```

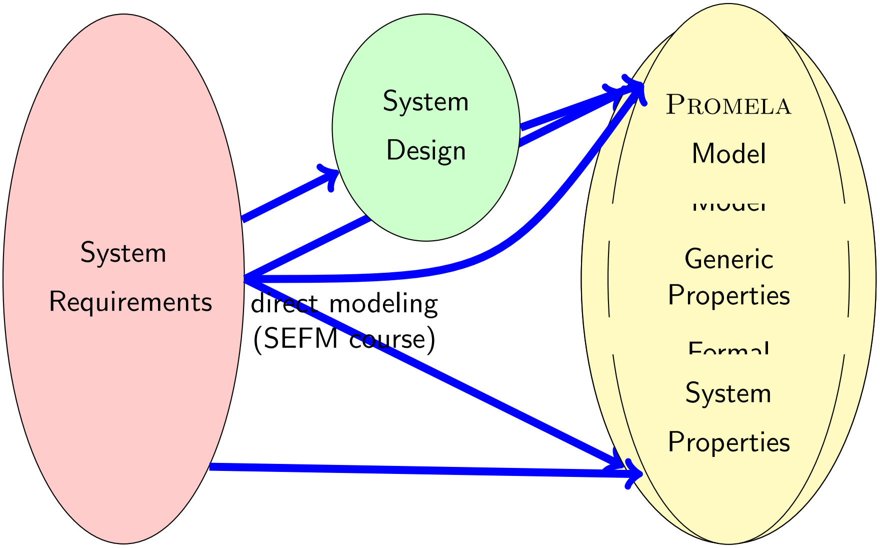
   Remark: Blocking statement in **atomic** may lead to interleaving (Lect. "Concurrency")

Usage Scenario of PROMELA

1. Model the essential features of a system in PROMELA
    - abstract away from complex (numerical) computations
        - make usage of non-deterministic choice of outcome
    - replace unbounded data structures with finite approximations

2. Select properties that the PROMELA model must satisfy
    - Generic Properties (discussed in later lectures)
        - Mutual exclusion for access to critical resources
        - Absence of deadlock
        - Absence of starvation
    - System-specific properties
        - Event sequences (e.g., system responsiveness)

Formalisation with PROMELA Abstraction



Usage Scenario of PROMELA Cont'd

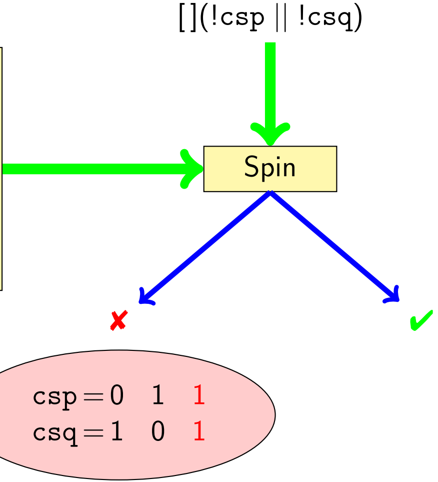
1. Model the essential features of a system in PROMELA
   - abstract away from complex (numerical) computations
     - make usage of non-deterministic choice of outcome
   - replace unbounded datastructures with finite approximations
   - assume fair process scheduler
2. Select properties that the PROMELA model must satisfy
   - Mutal exclusion for access to critical resources
   - Absence of deadlock
   - Absence of starvation
   - Event sequences (e.g., system responsiveness)
3. Verify that all possible runs of PROMELA model satisfy properties
   - Typically, need many iterations to get model and properties right
   - Failed verification attempts provide feedback via counter examples
   - Topic of next week's lecture

Verification: Work Flow (Simplified)

PROMELA Program

Properties

```
byte n = 0;
active proctype P() {
  n = 1
}
active proctype Q() {
  n = 2
}
```



Literature for this Lecture

Ben-Ari Chapter 1, Sections 3.1–3.3, 3.5, 4.6, Chapter 6

Spin Reference card (linked from Links, Papers, and Software section of course homepage)

jspin User manual, file `doc/jspin-user.pdf` in distribution