Introduction To Promela and Spin Model Checker

Ehsan Khamespanah

Fall 2022

What is Promela?

- Promela is an acronym: **Pro**cess **Me**ta-**La**nguage
- Promela is a language for modeling concurrent systems
  - Multi-threaded
  - Synchronisation and message passing
  - Few control structures, pure (no side-effects) expressions
  - Data structures with finite and fixed bound

Promela Is 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
  - Fair scheduling policy (during verification) I No data encapsulation
  - Non-deterministic

The First Promela Program!

```
active prototype P() {
  printf("Hello World\n");
}
```

```
> spin hello.pml
Hello World
```

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

Arithmetic Data Types

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

- Data types byte, short, int, unsigned with operations +, -, *, /, %
- All declarations implicitly at beginning of process (avoid to have them anywhere else!)
- Expressions computed as int, then converted to container type
- Arithmetic variables implicitly initialized to 0
- No floats, no side effects, C/Java-style comments
- No string variables (only in print statements)

Booleans and Enumerations

- **bit** is actually small numeric type containing 0,1 (unlike C, Java), **bool**, **true**, **false** syntactic sugar for **bit**, 0, 1

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

- Defining new types
  - 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

```
mytype = { Red, Yellow, Green }
mytype light = Green;
printf("the light is %e\n", light);
```

Control Statements

- Sequence: using ; as separator; C/Java-like rules
- Guarded Command
- Selection: non-deterministic choice of an alternative
- Repetition: loop until break (or forever)
- Goto: jump to a label

Syntax of Guard Statements

```
:: guard-statement -> command;
```

- Symbol -> is overloaded in Promela
- Semicolon optional
- first statement after :: used as guard
- :: guard is admissible (empty command)
- Can use ; instead of -> (avoid!)

Guarded Commands: Selection

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

- Trace of random simulation of multiple runs

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

Guarded Commands: Selection

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

- Guards may “overlap” (more than one can be true at the same time)
- Any alternative whose guard is true is randomly selected
- When no guard true: process blocks until one becomes true

Default Option for Selection

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

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

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

Second alternative can be selected only if p is false

Guarded Commands: Repetition

- Any alternative whose guard is true is randomly selected
- Only way to exit loop is via break or goto
- When no guard true: loop blocks until one becomes true

```
active prototype P() {
    /* computes gcd */
  byte a = 15, b = 20;
  if
   :: a > b -> a = a - b
   :: a < b -> b = b - a
   :: a == b -> break
  fi
}
```

- Trace with values of local variables

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

Arrays

- Arrays start with 0 as in Java and C
- Array bounds are constant and cannot be changed
- Only one-dimensional arrays (there is an (ugly) workaround)

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

Record Types

- C-style syntax
- Can be used to realize multi-dimensional arrays:
  typedef VECTOR {
      int vector[10]
  };

```
typedef DATE {
  byte day, month, year;
}

active prototype P() {
  Date d;
  d.day = 1; d.month = 3; d.year = 2022;
}
```

- VECTOR matrix[5]; /* base type array in record */
  matrix[3].vector[6] = 17;

Jumps

- 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

```
#define N 10

active prototype P() {
  byte sum = 0, i = 0;
  do
    :: i > N-> goto exitloop
    :: else -> sum = sum + i; i++;
  od
exitloop:
  printf("End of Loop\n");
}
```

Inline Code

- Promela has no method or procedure calls
- macro-like abbreviation mechanism for code that occurs multiply
- creates new local variables for parameters, but no new scope
- avoid to declare variables in inline — they are visible

```
typedef DATE {
  byte day, month, year;
}
inline setDate(D, DD, MM, YY) {
  D.day = DD; D.month = MM; D.year = YY;
}
active prototype P() {
  Date d;
  setDate(d, 1, 7, 2022);
  d.day = 1; d.month = 3; d.year = 2022;
}
```

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-Deterministic Programs

- Non-trivial Promela programs are non-deterministic!
- Possible sources of non-determinism
  - Non-deterministic choice of alternatives with overlapping guards
  - Scheduling of concurrent processes

Non-Deterministic Generation of Values

- Assignment statement used as guard
  - Assignment statement always succeeds (guard is true)
  - Side effect of guard is desired effect of this alternative
  - Could also write :: true -> range = 1, etc.
- Selects non-deterministically a value in {1,2,3,4} for range

```
byte range;
do
  :: range = 1;
  :: range = 2;
  :: range = 3;
  :: range = 4;
od
```

Non-Deterministic Generation of Values

- Generation of values from explicit list impractical for large range
- Increase of range and loop exit selected with equal chance
- 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

```
#define LOW 0
#define HIGH 1
byte range = LOW;
do
  :: range < HIGH -> range++
  :: break
od
```

Sources of Non-Determinism

- Non-deterministic choice of alternatives with overlapping guards
- Scheduling of concurrent processes
  - Can declare more than one process (at most 255 processes)

```
active prototype P() {
  printf(“Process P Statement 1\n");
  printf(“Process P Statement 2\n");
}

active prototype Q() {
  printf(“Process Q Statement 1\n");
  printf(“Process Q Statement 2\n");
}
```

Execution of Concurrent Processes

- Scheduling of concurrent processes on one processor
- Scheduler selects process randomly where next statement executed
- Many different computations are possible: non-determinism
- Use `-p` and `-g` options to see more execution details

- Random simulation of two processes:

```
> spin interleave.pml
```

Sets of Processes

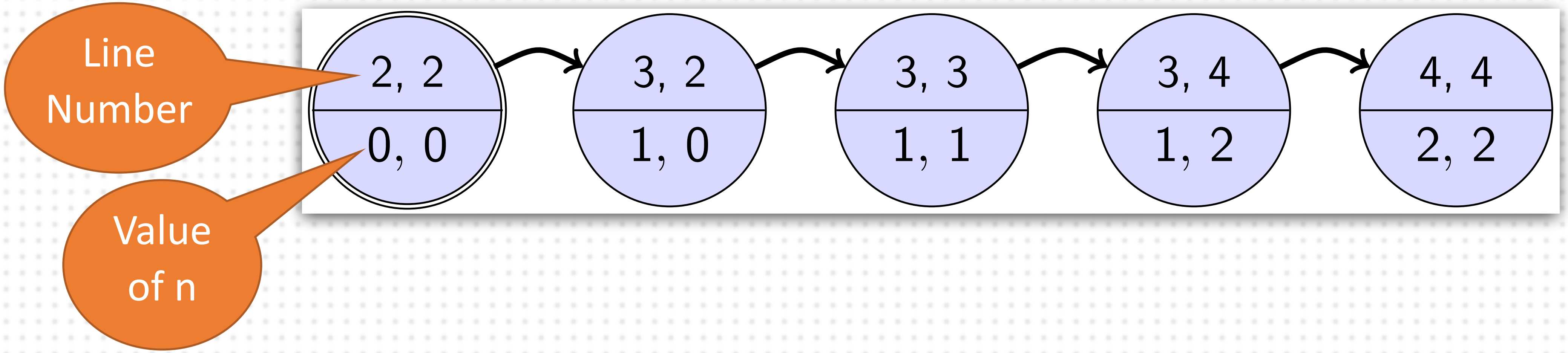
- Can declare set of identical processes
- Current process identified with reserved variable _pid
- Each process can have its own local variables

```
active [2] prototype P() {
  printf(“Process %d Statement 1\n”, _pid);
  printf(“Process %d Statement 2\n", _pid);
}
```

Promela Computations

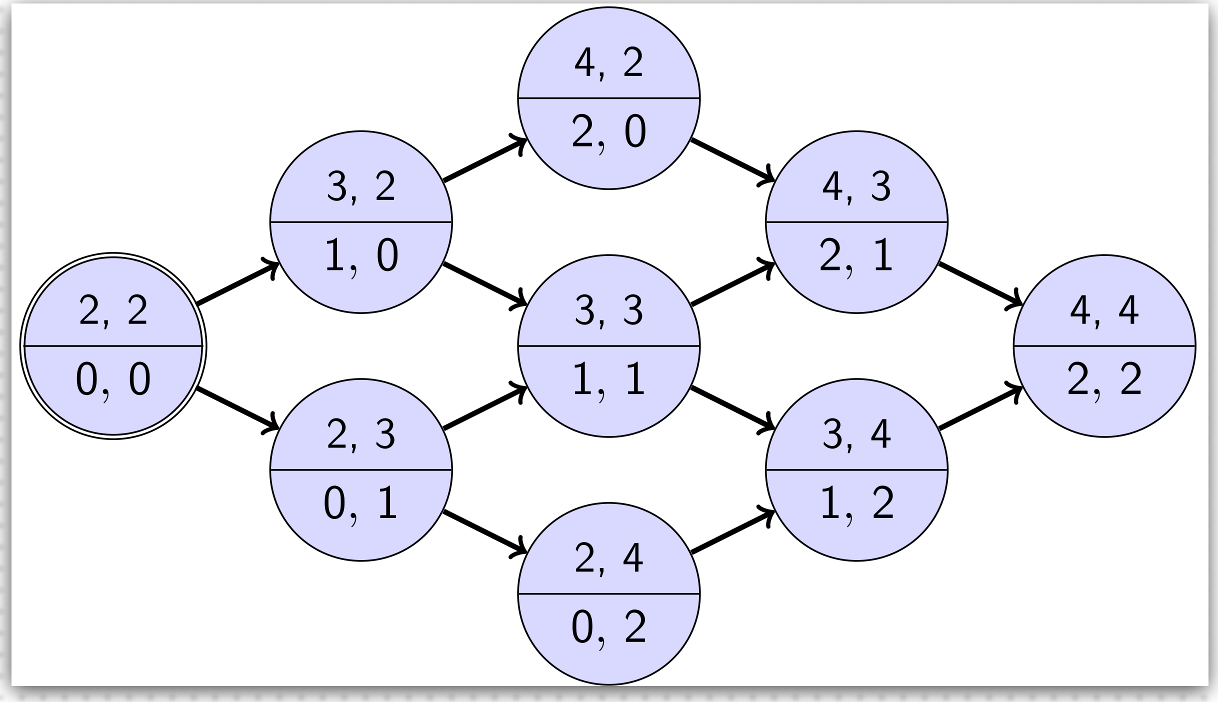
```
active [2] prototype P() {
  byte n;
  n = 1;
  n = 2;
}
```

- One possible computation of this program



Promela Computations

- Semantics of concurrent Promela program are all its interleavings
- Called interleaving semantics of concurrent programs
- Not universal: in Java certain reorderings allowed



Atomicity

- At which granularity of execution can interleaving occur?
- 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

```
int a, b, c;

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

- Interleaving into selection statement forced by interactive simulation

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

Atomicity

```
int a, b, c;

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

Communication Among Processes

- Asynchronous (buffered, default is FIFO) Channels used for passing messages
- Synchronously (rendez-vous)

```
chan qid = [4] of {byte};
chan synch[3] = [0] of {int};

/* sending through channels*/
qid ! byte-expression
synch ! int-expression

/* receiving value from channels*/
qid ? variable-name
synch ? variable-name
```

Type of Messages

Channel Name

Capacity

Literature for this Lecture

- Bernhard Beckert: Formal Specification and Verification
  Introduction to Promela
- Ben-Ari: Chapter 1, Sections 3.1–3.3, 3.5, 4.6, Chapter 6
- Spin Reference card (linked from Exercises)
- jspin User manual, file doc/jspin-user.pdf in distribution