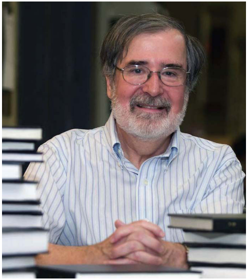
Formale Methoden im Software Entwurf

Einführung in PROMELA / Introduction to PROMELA

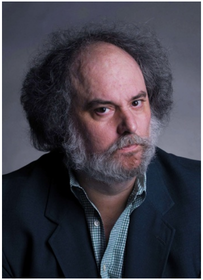
Reiner Hähnle

22 Oktober 2018

Model Checking:
A Success Story of Formal Methods in CS and EE



Edmund Clarke



E. Allen Emerson

Joseph Sifakis

Turing Award Winners 2007

Towards Model Checking

System Model

System Property

Promela Program

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

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

Model Checker

✘

✔

criticalSectP = 0 1 1
criticalSectQ = 1 0 1

What is PROMELA?

PROMELA is an acronym

Process meta-language

PROMELA: a language for modeling concurrent systems

- Multi-threaded “nebenläufig”
- Synchronisation and message passing
- Few control structures, pure expressions (no side-effects)
- 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 discuss most of PROMELA just in today's lecture!)

- No reference types
- No methods/procedures (but macros)
- No libraries
- No GUI, no standard input (but has output)
- No floating point types
- No data encapsulation
- Assumes fair scheduling policy (during verification)
- Non-deterministic (but executable, e.g., with random scheduler)

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 a process named P
- C-like command and expression syntax
- C-like (simplified) formatted print

Arithmetic Data Types

```
active proctype P() {
   int   val = 123;
   short 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 `+`,`-`,`*`,`/`,`%`
  - Semantics based on underlying C data types
  - Expressions computed as **int**, then converted to container type
  - Non-initialized arithmetic variables set to 0
- No floats, no side effects, C/Java-style comments
- No string variables (merely string literals in print statements)
- Compiler moves all declarations to start of process (SPIN $<$ 6.0)
  - Avoid to put declarations anywhere else than at start

Booleans

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

Observations

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

```
active proctype P() {
  bit  b1 = 0;
  bool b2 = 1;
  printf("%d\n", b2 == 3); /* is false */
  printf("%d\n", b1 < b2); /* is true */
  b2 = 3;                  /* raises error */
  printf("%d\n", b2);      /* b2 truncated to 1 */
}
```

```
> spin bool.pml
```

Enumerations

```
mtype = { red, yellow, green };
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 usage was for message names)
- There is at most one **mtype** per program

Control Statements (Complex Commands)

Sequencing	using ; as separator; C/JAVA-like rules
Guarded Commands:	
Selection (**if**)	non-deterministic choice of an alternative
Repetition (**do**)	loop until `break` (or forever)
For-Loops	C-like, translated to do-loop
Goto	jump to a label

Guarded Commands (1): Selection (if)

```
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
> ...
```

Guarded Commands (1): Selection (if)

```
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
}
```

Observations

- Guards may "overlap" (more than one can be true at the same time)
- Any alternative whose guard is true is indiscriminately selected
- When no guard true: process blocks until one becomes true
  - this may never happen—we come back to this situation

Guarded Statement Syntax

```
:: guard-statement  -> command

:: guard-statement  ;  command
```

Observations

- Symbol `->` is overloaded in Promela
  - Also conditonal expression (`boolean-guard -> then : else`)
  - Brackets mandatory, don't confuse with usage in guarded command
- First statement after `::` evaluated as guard
  - "`:: guard`" is admissible (empty command)
  - Can use `;` instead of `->` (confusing, avoid!)

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

So far, all our programs execute each code sequence at most once:

We need loops

Guarded Commands (2): Repetition (do)

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

Guarded Commands (2): Repetition (do)

```
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
}
```

Observations

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

For-Loops (SPIN Version $\geq$ 6)

```
#define N 10 /* C-style preprocessing */
active proctype P() { // sum1.pml
  int i;
  int sum = 0;
  for (i : 1 .. N) {
    sum = sum + i
  }
}
```

Observations

- C-style syntax
- For-loops can be nested
- Available since SPIN version 6, not in Ben-Ari's book
- Compiler translates for-loop into do-loop with break/goto
  - Translation see SPIN man page of for-loop (weblink)

Arrays

```
#define N 5
active proctype P() { // sum2.pml
  byte a[N];
  byte i; byte sum = 0;
  a[0] = 0;a[1] = 10;a[2] = 20;a[3] = 30;a[4] = 40;
  for (i in a) {
    sum = sum + a[i]
} }
```

Observations: Arrays

- Array indices start with 0 as in JAVA and C
- Array entries initialized with 0
- Arrays are value types: a≠b always different arrays
- Array bounds are constant and cannot be changed
  - There are no unbounded data types in PROMELA
- Only one-dimensional arrays (there is an (ugly) workaround)

Arrays

```
#define N 5
active proctype P() { // sum2.pml
  byte a[N];
  byte i; byte sum = 0;
  a[0] = 0;a[1] = 10;a[2] = 20;a[3] = 30;a[4] = 40;
  for (i in a) {
    sum = sum + a[i]
} }
```

Observations: For-loops over arrays

- Index variable of for-loop runs over array indices, not elements

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() { // sum3.pml
  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 the same 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 ever after

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!
(no point to execute it more than once . . .)

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

```
// range.pml
byte range;
if
  :: range = 1
  :: range = 2
  :: range = 3
  :: range = 4
fi
```

Observations

- 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`

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

- 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

Non-Deterministic Generation of Values: Select

```
#define LOW  0
#define HIGH 9
active proctype P() {
  int i;
  select(i: LOW .. HIGH);
  printf("I␣selected␣%d\n", i)
}
```

Observations

- Available since SPIN Version 6.0 (not in Ben-Ari's book)

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

- Multi-threading: concurrent processes scheduled on one processor
- Scheduler selects process randomly where next statement executed
  - In command line output selected process visualized by indentation
- Many different interleavings are possible: non-determinism
- Use −p option 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 processes with identical code
- Current process identified with reserved variable `_pid`
- Each process can have its own local variables

Command Line Execution

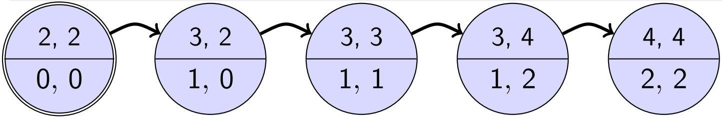
Random simulation of set of two processes

```
> spin interleave_set.pml
```

PROMELA **Computations "Ausführungen"**

```
1 active [2] proctype P() { // interleave_set_n.pml
2   byte n = 0;
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 "Verschränkung"

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 $s_j = s^i_{j'}$ and $s_k = s^i_{k'}$ with $j < k$ it is the case that $j' < k'$.

The interleaved state sequence
respects the execution order of each process

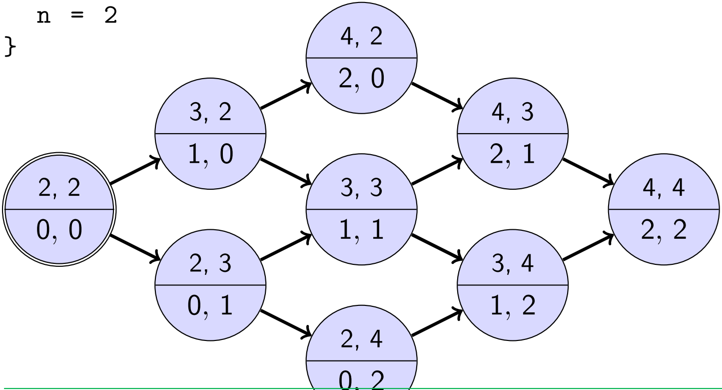
Observations

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

Interleaving Cont'd

Can represent possible interleavings as a directed graph

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



Atomicity

At which granularity of execution can interleaving occur?

Definition (Atomicity [slightly simplified, to be refined later])

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

The iSpin GUI

A GUI for Spin

- Written in Tcl/Tk, should just run when version $\geq$ 8.4 installed
- Launch from command line with `./ispin` (must be executable)
- No need to recall options + more readable output + management
  - Spin options executed in background displayed
  - Most Spin output displayed in structured manner
- Choose `Simulate / Replay` menu, `Interactive` mode

Command Line Execution

```
> ispin zerotest.pml
> ispin 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() {
     b = 1; c = 1;
     atomic {
     a = 1;
     if
       :: a != 0 -> c = b / a
       :: else -> c = b
     fi
     }
   }
   ```

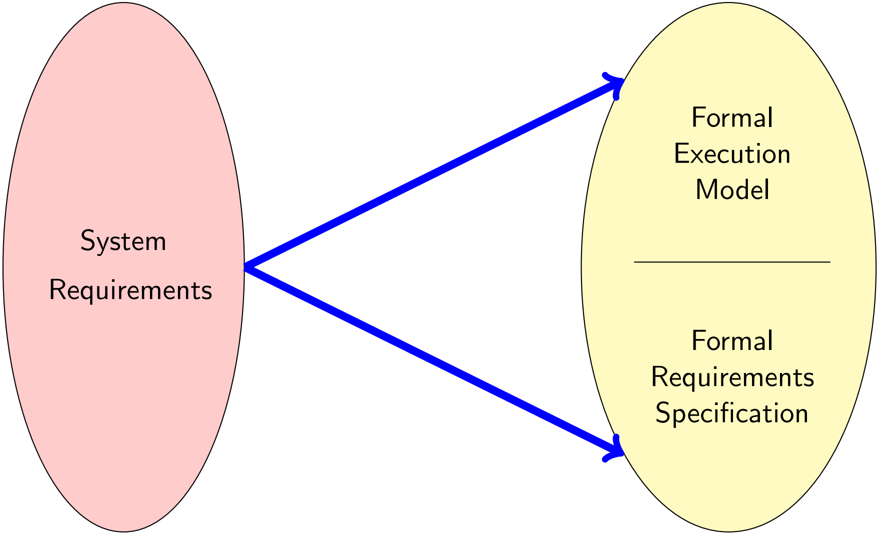
   Remark: Blocking statement in **atomic** may lead to interleaving (Lecture "Concurrency")

Outlook: Usage Scenario of PROMELA

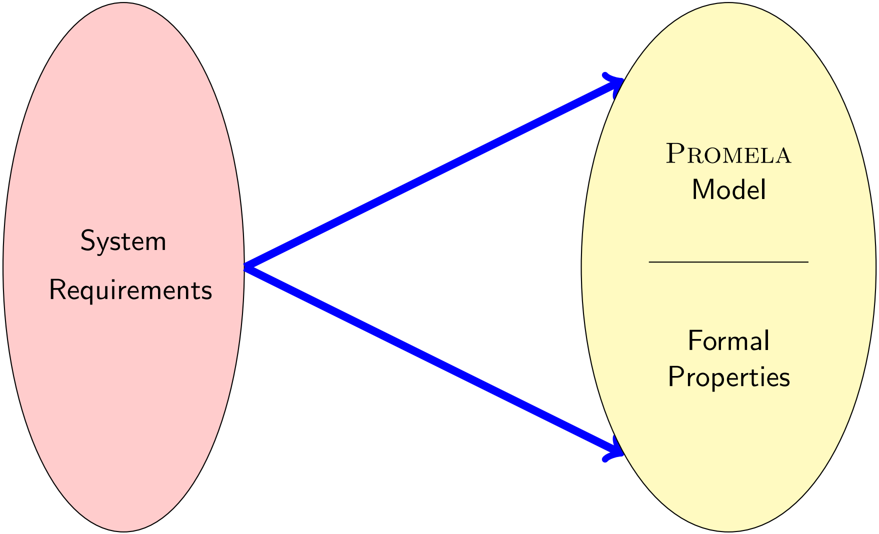
1. Model the essential features of a system in PROMELA
   - Abstract away from complex (numerical) computations
     - Make use of non-deterministic choice of outcome
   - Replace unbounded data structures with finite approximations
   - Assume fair process scheduler

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)

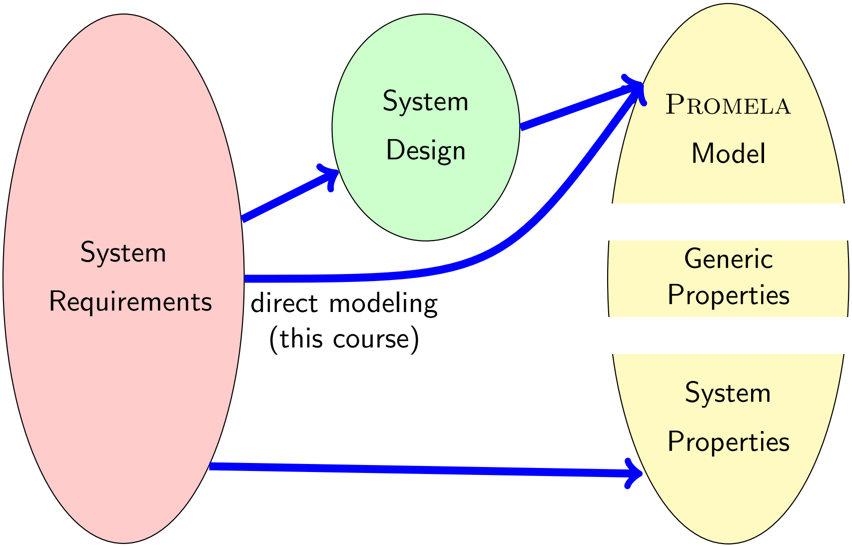
Formalisation with Promela



Formalisation with PROMELA



Formalisation with PROMELA



Usage Scenario of PROMELA Cont'd

1. Model the essential features of a system in PROMELA
   - Abstract away from complex (numerical) computations
     - Make use 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
   - Mutual 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

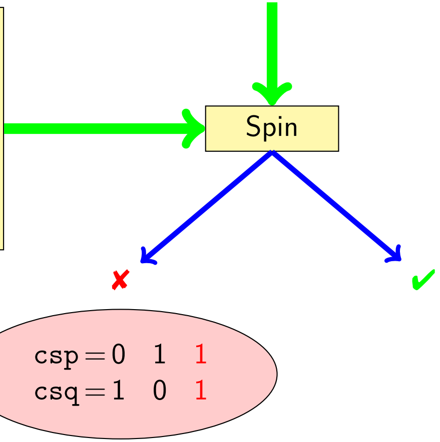
Verification: Work Flow (Simplified)

PROMELA Program

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

Properties

[](!csp || !csq)



Literature for this Lecture

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

Spin Reference card
(link to moodle)

Think: What Are Useful Properties?