Software Engineering using Formal Methods

Verification with Spin

Wolfgang Ahrendt

02 September 2016

Spin: Previous Lecture vs. This Lecture

Previous lecture

Spin appeared as a Promela simulator

This lecture

Intro to Spin as a model checker

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

MC's search for counter examples is exhaustive.

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

MC's search for counter examples is exhaustive.

⇒ Finding no counter example proves stated correctness properties.

What does ‘exhaustive search’ mean here?

exhaustive search

=

resolving non-determinism in all possible ways

What does ‘exhaustive search’ mean here?

exhaustive search

=

resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

What does ‘exhaustive search’ mean here?

exhaustive search

=

resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

- explicit, local:
  if/**do** statements

```
:: guardX -> ...
:: guardY -> ...
```

What does ‘exhaustive search’ mean here?

exhaustive search

=

resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

- explicit, local:
  if/**do** statements

  ```
  :: guardX -> ...
  :: guardY -> ...
  ```

- implicit, global:
  scheduling of concurrent processes
  (see next lecture)

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

Model Checker for This Course: SPIN

SPIN: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of SPIN:

- simulating a model (randomly/interactively)
- generating a verifier

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing model checking:

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing model checking:

- exhaustively checks Promela model against correctness properties

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing model checking:

- exhaustively checks Promela model against correctness properties
- in case the check is negative:
  generates a failing run of the model

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

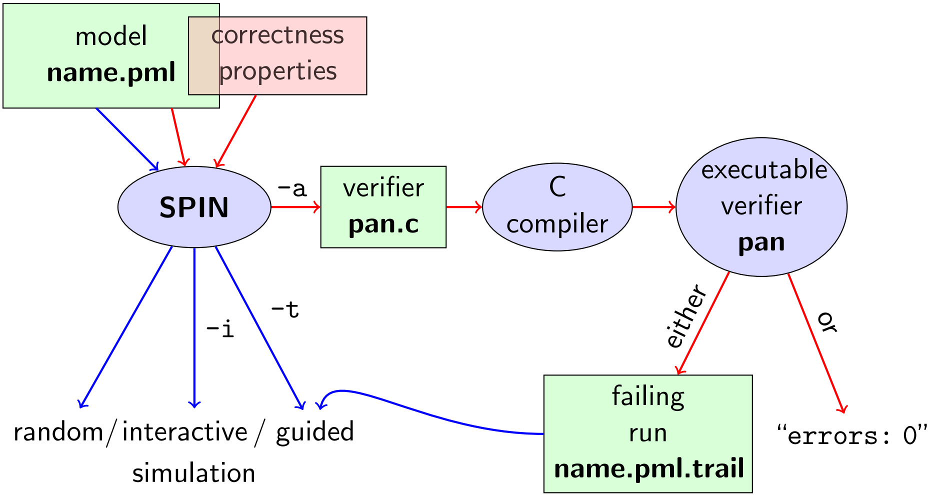
- simulating a model (randomly/interactively/guided)
- generating a verifier

verifier generated by Spin is a C program performing

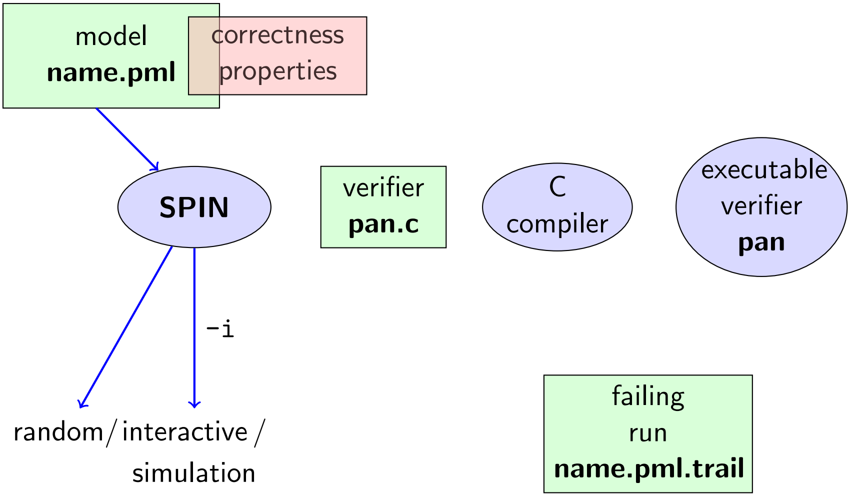
model checking:

- exhaustively checks Promela model against correctness properties
- in case the check is negative:
  generates a failing run of the model, to be simulated by Spin

Spin Workflow: Overview



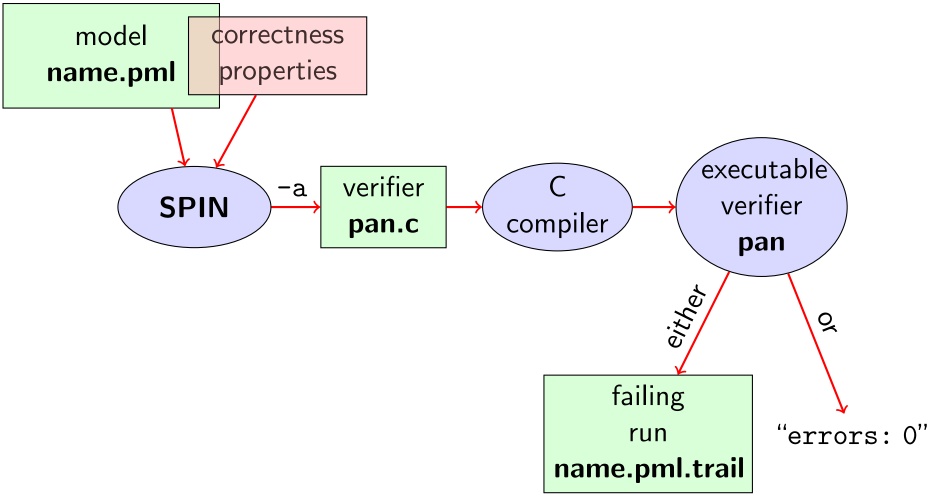
Plain Simulation with SPIN



Rehearsal: Simulation Demo

- run example, random and interactive
  `zero.pml`

Model Checking with Spin



Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  ($R_{M,C_i} \subseteq R_M$)

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M \subseteq (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M \subseteq (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.
- If M is not correct wrt. $C_1, \ldots, C_n$, then
  each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M \subseteq (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.
- If M is not correct wrt. $C_1, \ldots, C_n$, then
  each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

We know how to write models M.

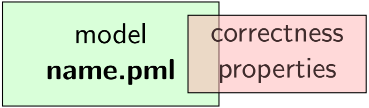
Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

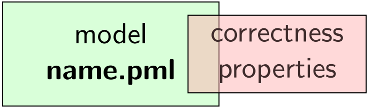
- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  ($R_{M,C_i} \subseteq R_M$)
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M \subseteq (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.
- If M is not correct wrt. $C_1, \ldots, C_n$, then
  each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

We know how to write models M.
But how to write Correctness Properties?

Stating Correctness Properties

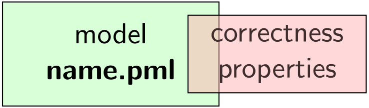


Stating Correctness Properties



Correctness properties can be stated within, or outside, the model.

Stating Correctness Properties

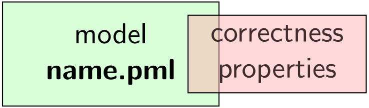


Correctness properties can be stated within, or outside, the model.

stating properties within model, using

- assertion statements

Stating Correctness Properties

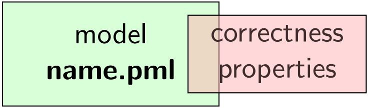


Correctness properties can be stated within, or outside, the model.

stating properties within model, using

- assertion statements
- meta labels
  - `end` labels
  - `accept` labels
  - `progress` labels

Stating Correctness Properties



Correctness properties can be stated within, or outside, the model.

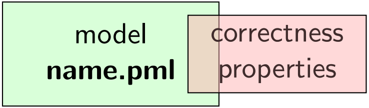
stating properties within model, using

- assertion statements
- meta labels
  - `end` labels
  - `accept` labels
  - `progress` labels

stating properties outside model, using

- never claims
- temporal logic formulas

Stating Correctness Properties



Correctness properties can be stated within, or outside, the model.

stating properties within model, using

- assertion statements (today)
- meta labels
  - end labels (today)
  - accept labels
  - progress labels

stating properties outside model, using

- never claims
- temporal logic formulas

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

$$\mathbf{assert}(\mathit{expr})$$

were *expr* is any PROMELA expression.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

assert(*expr*)

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

assert(*expr*)

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

assert(*expr*)

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

```
...
stmt1;
assert(max == a);
stmt2;
...
```

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

assert(*expr*)

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

```
...
stmt1;
assert(max == a);
stmt2;
...
```

```
...
if
  :: b1 -> stmt3;
           assert(x < y)
  :: b2 -> stmt4
...
```

Meaning of Boolean Assertion Statements

assert(*expr*)

- has no effect if *expr* evaluates to true
- triggers an error message if *expr* evaluates to false

This holds in both, simulation and model checking mode.

Meaning of General Assertion Statements

assert(*expr*)

- has no effect if *expr* evaluates to non-zero value
- triggers an error message if *expr* evaluates to 0

This holds in both, simulation and model checking mode.

Meaning of General Assertion Statements

assert(*expr*)

- has no effect if *expr* evaluates to non-zero value
- triggers an error message if *expr* evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

bool **true** **false** is syntactic sugar for

Meaning of General Assertion Statements

assert(*expr*)

- has no effect if *expr* evaluates to non-zero value
- triggers an error message if *expr* evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

bool	**true**	**false**	is syntactic sugar for
bit	1	0	

Meaning of General Assertion Statements

assert(*expr*)

- has no effect if *expr* evaluates to non-zero value
- triggers an error message if *expr* evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

bool	**true**	**false**	is syntactic sugar for
bit	1	0	

⇒ general case covers Boolean case

Instead of using 'printf's for Debugging ...

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

Instead of using 'printf's for Debugging ...

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

Command Line Execution

(simulate, inject fault, simulate again)

```
> spin [-i] max.pml
```

... we can employ Assertions

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
assert( max == (a>b -> a : b) )
```

... we can employ Assertions

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
assert( max == (a>b -> a : b) )
```

Now, we have a first example with a formulated correctness property.

… we can employ Assertions

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
assert( max == (a>b -> a : b) )
```

Now, we have a first example with a formulated correctness property.

We can do model checking, for the first time!

... we can employ Assertions

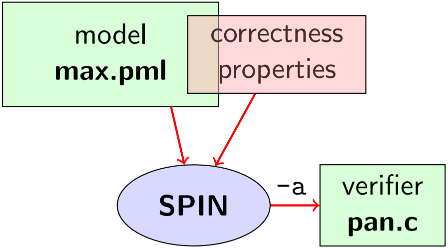
```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
assert( max == (a>b -> a : b) )
```

Now, we have a first example with a formulated correctness property.

We can do model checking, for the first time!

(Historic moment in the course.)

Generate Verifier in C



Command Line Execution

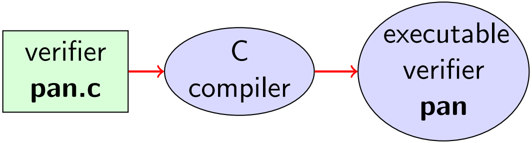
Generate Verifier in C

```
> spin -a max2.pml
```

SPIN generates Verifier in C, called **pan.c**

(plus helper files)

Compile To Executable Verifier

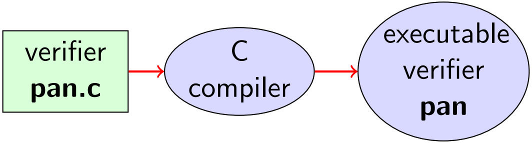


Command Line Execution

compile to executable verifier

```
> gcc -o pan pan.c
```

Compile To Executable Verifier



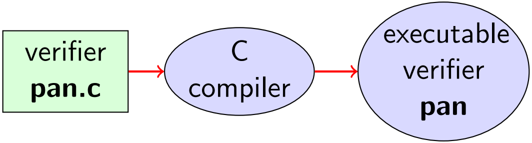
Command Line Execution

compile to executable verifier

```
> gcc -o pan pan.c
```

C compiler generates executable verifier **pan**

Compile To Executable Verifier



Command Line Execution

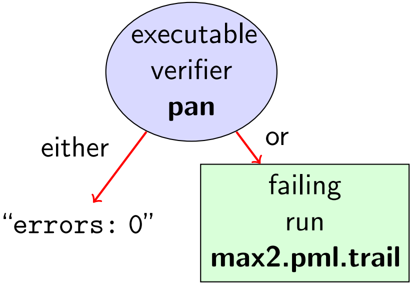
compile to executable verifier

```
> gcc -o pan pan.c
```

C compiler generates executable verifier **pan**

pan: historically "**p**rotocol **an**alyzer", now "**p**rocess **an**alyzer"

Run Verifier (= Model Check)

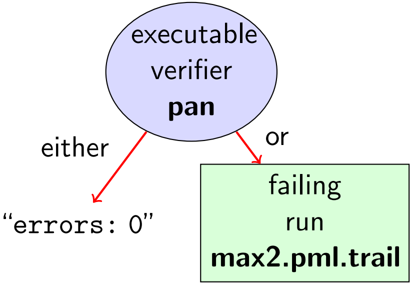


Command Line Execution

run verifier **pan**

> ./pan   or   > pan

Run Verifier (= Model Check)



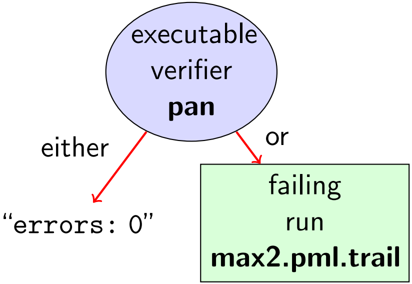
Command Line Execution

run verifier **pan**

> ./pan   or   > pan

- prints "errors: 0"

Run Verifier (= Model Check)



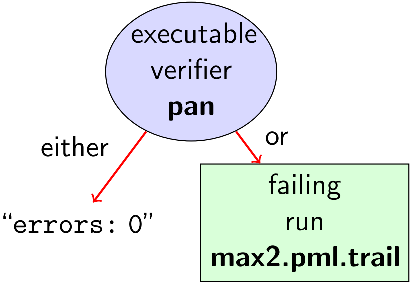
Command Line Execution

run verifier **pan**

> ./pan   *or*   *> pan*

- prints "errors: 0" ⇒ Correctness Property verified!

Run Verifier (= Model Check)



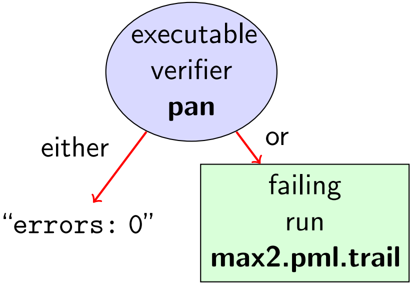
Command Line Execution

run verifier **pan**

> ./pan *or* *> pan*

- prints "errors: 0", or
- prints "errors: n" $(n > 0)$

Run Verifier (= Model Check)



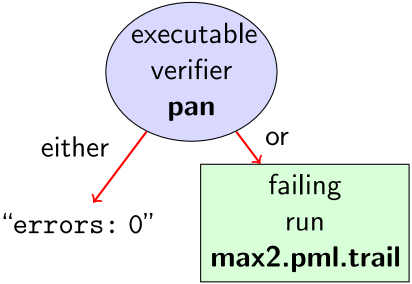
Command Line Execution

run verifier **pan**

`> ./pan` *or* `> pan`

- prints "`errors: 0`", or
- prints "`errors:` n" $(n > 0)$ $\Rightarrow$ counter example found!

Run Verifier (= Model Check)



Command Line Execution

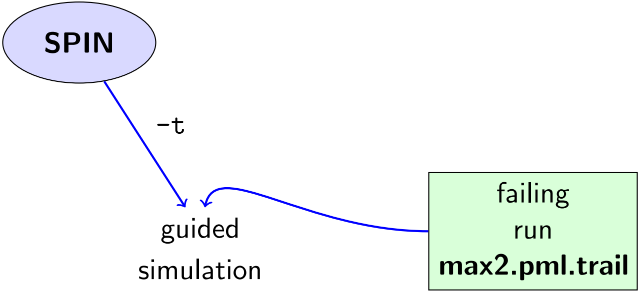
run verifier **pan**

```
> ./pan   or   > pan
```

- prints "errors: 0", or
- prints "errors: n" $(n > 0)$ ⇒ counter example found!
  records failing run in **max2.pml.trail**

Guided Simulation

To examine failing run: employ simulation mode, "guided" by trail file.



Command Line Execution

inject a fault, re-run verification, and then:

```
> spin -t -p -l max2.pml
```

Output of Guided Simulation

can look like:

```
Starting P with pid 0
1: proc 0 (P) line  8 "max2.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max2.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max2.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max2.pml" (state 14) [max = a ]
                P(0):max = 1
spin: max2.pml:22, Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

Output of Guided Simulation

can look like:

```
Starting P with pid 0
1: proc 0 (P) line  8 "max2.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max2.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max2.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max2.pml" (state 14) [max = a ]
                P(0):max = 1
spin: max2.pml:22, Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

assignments in the run

Output of Guided Simulation

can look like:

```
Starting P with pid 0
1: proc 0 (P) line  8 "max2.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max2.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max2.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max2.pml" (state 14) [max = a ]
                P(0):max = 1
spin: max2.pml:22, Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

assignments in the run

values of variables whenever updated

Output of Guided Simulation

can look like:

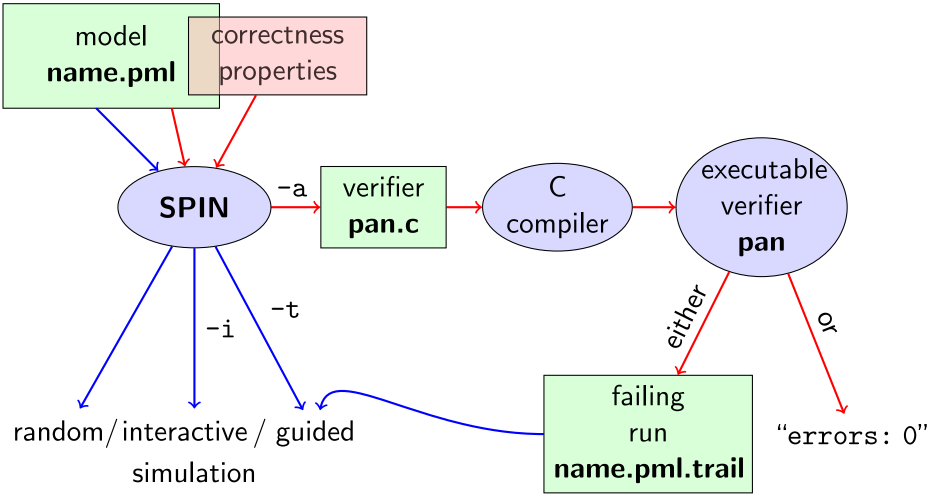
```
Starting P with pid 0
1: proc 0 (P) line  8 "max2.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max2.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max2.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max2.pml" (state 14) [max = a ]
                P(0):max = 1
spin: max2.pml:22, Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

assignments in the run
values of variables whenever updated

(If output doesn't mention `max` variable, re-verify with `./pan -E`)

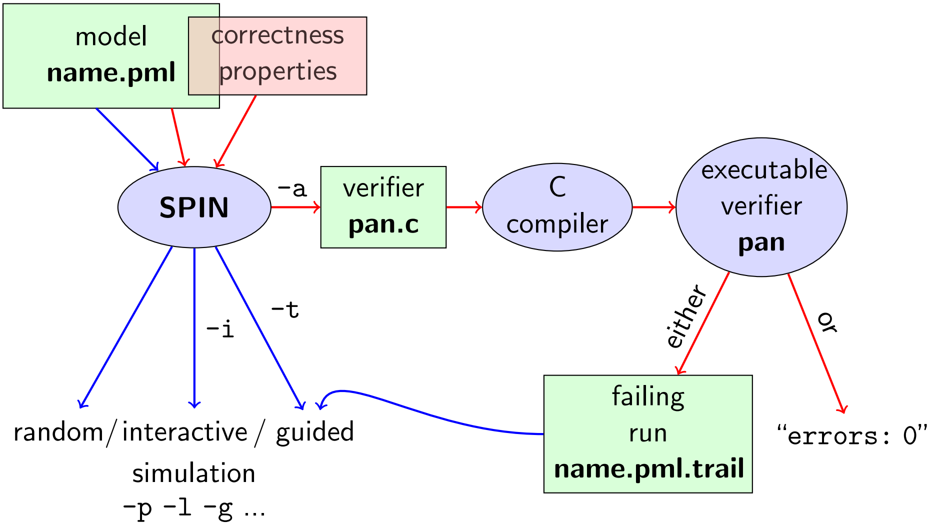
What did we do so far?

following whole cycle (most primitive example, assertions only)



What did we do so far?

following whole cycle (most primitive example, assertions only)



Further Examples: Integer Division

```
int dividend = 15;
int divisor  = 4;
int quotient, remainder;

quotient = 0;
remainder = dividend;
do
  :: remainder > divisor ->
     quotient++;
     remainder = remainder - divisor
  :: else ->
     break
od;
printf("%d␣divided␣by␣%d␣=␣%d,␣remainder␣=␣%d\n",
       dividend, divisor, quotient, remainder)
```

Further Examples: Integer Division

```
int dividend = 15;
int divisor  = 4;
int quotient, remainder;

quotient = 0;
remainder = dividend;
do
  :: remainder > divisor ->
     quotient++;
     remainder = remainder - divisor
  :: else ->
     break
od;
printf("%d␣divided␣by␣%d␣=␣%d,␣remainder␣=␣%d\n",
       dividend, divisor, quotient, remainder)
```

simulate, put assertions, verify, change values, ...

Further Examples: Greatest Common Divisor

greatest common divisor of x and y

```
int a, b;
a = x; b = y;
do
  :: a > b -> a = a - b
  :: b > a -> b = b - a
  :: a == b -> break
od;
printf("The␣GCD␣of␣%d␣and␣%d␣=␣%d\n", x, y, a)
```

Further Examples: Greatest Common Divisor

greatest common divisor of x and y

```
int a, b;
a = x; b = y;
do
  :: a > b -> a = a - b
  :: b > a -> b = b - a
  :: a == b -> break
od;
printf("The␣GCD␣of␣%d␣and␣%d␣=␣%d\n", x, y, a)
```

full functional specification w. assertion not possible (why?)

Further Examples: Greatest Common Divisor

greatest common divisor of x and y

```
int a, b;
a = x; b = y;
do
  :: a > b -> a = a - b
  :: b > a -> b = b - a
  :: a == b -> break
od;
printf("The␣GCD␣of␣%d␣and␣%d␣=␣%d\n", x, y, a)
```

full functional specification w. assertion not possible (why?)

still, assertions can perform sanity check

Further Examples: Greatest Common Divisor

greatest common divisor of x and y

```
int a, b;
a = x; b = y;
do
  :: a > b -> a = a - b
  :: b > a -> b = b - a
  :: a == b -> break
od;
printf("The␣GCD␣of␣%d␣and␣%d␣=␣%d\n", x, y, a)
```

full functional specification w. assertion not possible (why?)

still, assertions can perform sanity check

⇒ typical for model checking

Typical Command Lines

typical command line sequences:

random simulation

```
spin name.pml
```

Typical Command Lines

typical command line sequences:

random simulation

```
spin name.pml
```

interactive simulation

```
spin -i name.pml
```

Typical Command Lines

typical command line sequences:

random simulation

```
spin name.pml
```

interactive simulation

```
spin -i name.pml
```

model checking

```
spin -a name.pml
gcc -o pan pan.c
./pan
```

Typical Command Lines

typical command line sequences:

random simulation

```
spin name.pml
```

interactive simulation

```
spin -i name.pml
```

model checking

```
spin -a name.pml
gcc -o pan pan.c
./pan
```

and in case of error

```
spin -t -p -l -g name.pml
```

Spin Reference Card

Ben-Ari produced Spin Reference Card, summarizing

- typical command line sequences
- options for
    - Spin
    - gcc
    - pan
- Promela
    - datatypes
    - operators
    - statements
    - guarded commands
    - processes
    - channels
- temporal logic syntax

Spin Reference Card

Ben-Ari produced Spin Reference Card, summarizing

- typical command line sequences
- options for
  - Spin
  - gcc
  - pan
- Promela
  - datatypes
  - operators
  - statements
  - guarded commands
  - processes
  - channels
- temporal logic syntax

⇒ available from course page (see 'Links, Papers, and Software')

Why Spin?

- Spin targets software, instead of hardware verification ("*Software* Engineering using Formal Methods")
- 2001 ACM Software Systems Award (other winning software systems include: Unix, TCP/IP, WWW, Tcl/Tk, Java, GCC, TeX, Coq)
- used for safety critical applications
- distributed freely as research tool, well-documented, actively maintained, large user-base in academia and in industry
- annual Spin user workshops series held since 1995
- based on standard theory of (ω-)automata and linear temporal logic

Why SPIN? (Cont'd)

- PROMELA and SPIN are rather simple to use
- good to understand a few systems really well, rather than many systems poorly
- availability of good course book (Ben-Ari)
- availability of front end JSPIN (also Ben-Ari)

Why SPIN? (Cont'd)

- PROMELA and SPIN are rather simple to use
- good to understand a few systems really well, rather than many systems poorly
- availability of good course book (Ben-Ari)
- availability of front end JSPIN (also Ben-Ari)
- and now: availability of Bart's web interface

What is jSpin?

- graphical user interface for Spin
- developed for pedagogical purposes
- written in Java
- simple user interface
- Spin options automatically supplied
- fully configurable
- supports graphics output of transition system

What is JSPIN?

- graphical user interface for SPIN
- developed for pedagogical purposes
- written in JAVA
- simple user interface
- SPIN options automatically supplied
- fully configurable
- supports graphics output of transition system
- makes back-end calls transparent

jSpin Demo

Command Line Execution

calling jSpin

```
> java -jar /usr/local/jSpin/jSpin.jar
```

(with path adjusted to your setting)

or use shell script:

```
> jspin
```

jSpin **Demo**

Command Line Execution

calling jSpin

```
> java -jar /usr/local/jSpin/jSpin.jar
```

(with path adjusted to your setting)

or use shell script:

```
> jspin
```

play around with similar examples ...

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

simulate a few times

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
        a, b, max)
```

simulate a few times

⇒ crazy "`timeout`" message sometimes

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

simulate a few times

⇒ crazy "`timeout`" message sometimes

generate and execute **pan**

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

simulate a few times

⇒ crazy "`timeout`" message sometimes

generate and execute **pan**

⇒ reports "`errors: 1`"

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the maximum of %d and %d is %d\n",
       a, b, max)
```

simulate a few times
⇒ crazy "`timeout`" message sometimes

generate and execute **pan**
⇒ reports "`errors: 1`"

????

Catching A Different Type of Error

quoting from file **max3.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a
  :: b <= a -> max = b
fi;
printf("the␣maximum␣of␣%d␣and␣%d␣is␣%d\n",
       a, b, max)
```

simulate a few times
⇒ crazy "`timeout`" message sometimes

generate and execute **pan**
⇒ reports "`errors: 1`"

Note: no assert in **max3.pml**.

Catching A Different Type of Error

Further inspection of **pan** output:

```
...
pan: invalid end state  (at depth 1)
pan: wrote max3.pml.trail
...
```

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical,
for concurrent and distributed models (i.p. protocols).

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical,
for concurrent and distributed models (i.p. protocols).

But

It is *illegal* if a process blocks while no other process can proceed.

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical,
for concurrent and distributed models (i.p. protocols).

But

It is *illegal* if a process blocks while no other process can proceed.

$\Rightarrow$ "Deadlock"

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical,
for concurrent and distributed models (i.p. protocols).

But

It is *illegal* if a process blocks while no other process can proceed.

$\Rightarrow$ “Deadlock”

In **max3.pml**, there exists a blocking run where no process can take over.

Legal and Illegal Blocking

A process may *legally* block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical,
for concurrent and distributed models (i.p. protocols).

But

It is *illegal* if a process blocks while no other process can proceed.

$\Rightarrow$ "Deadlock"

In **max3.pml**, there exists a blocking run where no process can take over.

(Fix error)

Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.

Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)

End locations of a process `P` are:

- `P`'s textual end

Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)

End locations of a process `P` are:

- `P`'s textual end
- each location marked with an end label: "`end`*xxx*`:`"

Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)

End locations of a process `P` are:

- `P`'s textual end
- each location marked with an end label: "`end`*xxx*`:`"

End labels not useful in **max3.pml**, but elsewhere, they are.
Example: `end.pml`

Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)

End locations of a process `P` are:

- `P`'s textual end
- each location marked with an end label: "`end`*xxx*`:`"

End labels not useful in **max3.pml**, but elsewhere, they are.
Example: `end.pml`

Can get SPIN to ignore 'invalid end state' error: `./pan -E`

Literature for this Lecture

Ben-Ari Chapter 2, Sections 4.7.1, 4.7.2