Formale Methoden im Software Entwurf

Verifikation mit Spin / Verification with Spin

Reiner Hähnle

29 Oktober 2018

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 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 several iterations to get model and properties right
   - Failed verification attempts provide feedback via counter examples

SPIN: Previous Lecture vs. This Lecture

Previous lecture

SPIN presented as a PROMELA simulator (one run at a time)

This lecture

Intro to SPIN as a model checker (all runs & properties)

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the designer 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?

Absence of any counter example proves stated correctness properties

MC's search for counter examples is exhaustive

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the designer 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?

Absence of any counter example proves stated correctness properties

MC's search for counter examples is exhaustive

What Does "Exhaustive Search" Mean?

Exhaustive search
=
To resolve any non-determinism in all possible ways

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

- Explicit, local:
  Overlapping guards in **if**/**do** statements

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

- Implicit, global:
  Scheduling of concurrent processes
  (see next lecture)

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

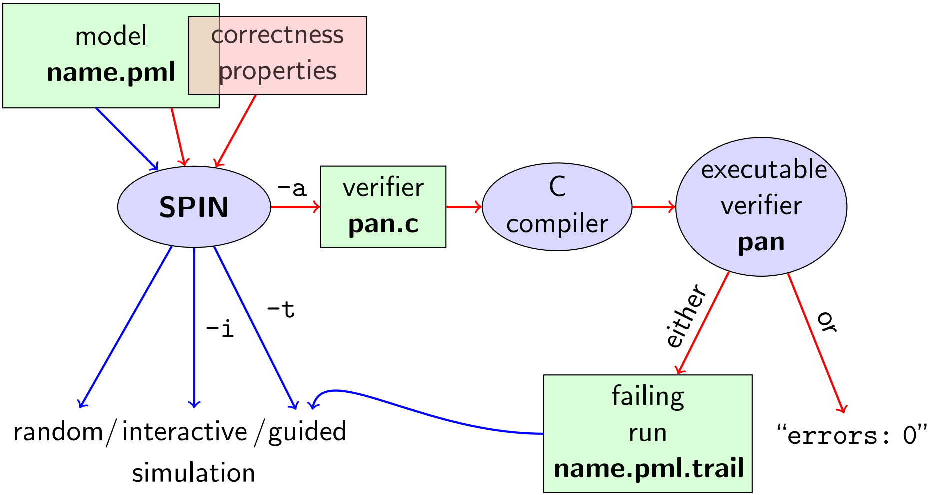
The name is a serious understatement!
Main functionalities of Spin:

1. Simulating a model (randomly/interactively/guided)
   as seen in previous lecture
2. Generating a verifier

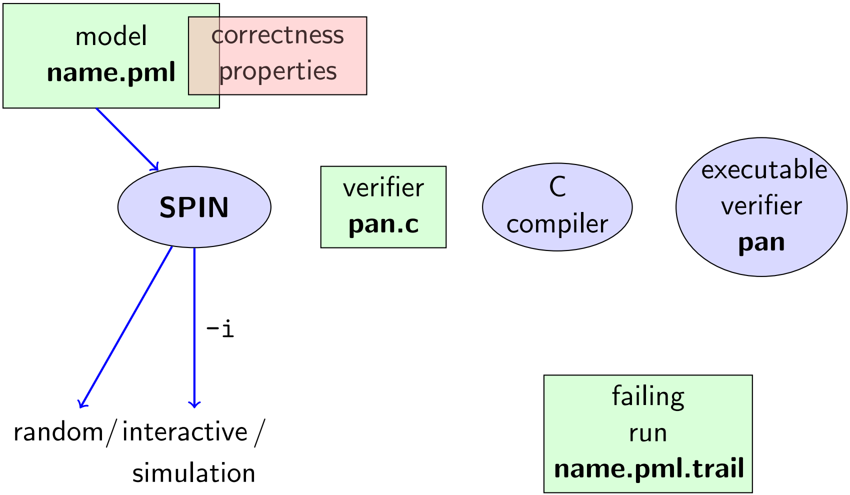
Verifier generated by Spin is a C program performing model checking:

- 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

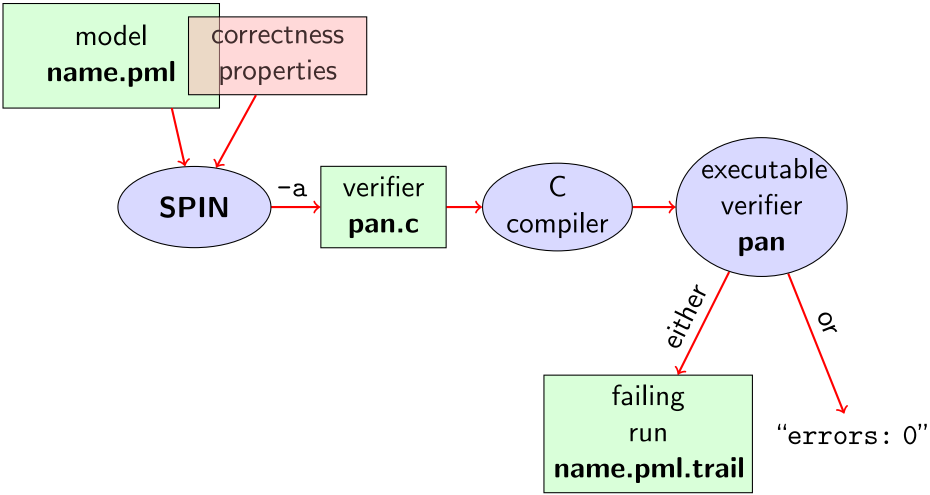
Spin Workflow: Overview, Refined



Plain Simulation with SPIN (Previous Lecture)



Model Checking with Spin



Meaning of Correctness relative to Properties

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

Assume there is a check whether a PROMELA run R satisfies property C

Definition (Correctness of PROMELA model relative to property)

Let R_M be set of all possible runs of PROMELA model M

- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i (clearly, $R_{M,C_i} \subseteq R_M$)
- M is correct relative to $C_1, \ldots, C_n$ iff $R_M = (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$
- If M is not correct, 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 in PROMELA
But how to write correctness properties?

Stating Correctness Properties

model
name.pml

correctness
properties

Correctness properties can be stated within or outside a PROMELA model

Stating properties within PROMELA model

- Assertion statements (today)
- Meta labels ("Markierungen")
  - `end` labels (today)
  - `accept` labels
  - `progress` labels

Stating properties outside PROMELA model

- Never claims
- Temporal logic formulas

Assertion ("Zusicherung") Statements

Definition (Assertion statement)

Assertion statements in PROMELA have the form

assert(*expr*)

where *expr* is any PROMELA expression.

Typically, (but not necessarily) *expr* is of type **bool**

assert(*expr*) in PROMELA can take any statement position:

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

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

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

$\Rightarrow$ General case covers Boolean case

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

```
byte a, b, max;
select(a: 1 .. 3);
select(b: 1 .. 3);
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
printf("maximum␣of␣%d␣and␣%d␣is␣%d\n", a, b, max)
```

Command Line Execution

(simulate, check whether printed outcome is correct — oracle problem)

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

... we can employ Assertions

Quoting from file `maxA.pml`:

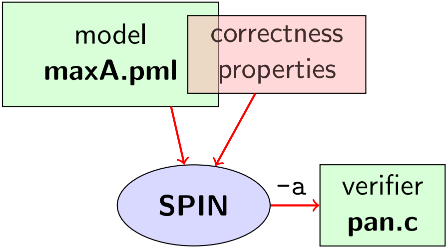
```
byte a, b, max;
select(a: 1 .. 3);
select(b: 1 .. 3);
if
  :: a >= b -> max = a
  :: a <= b -> max = b
fi;
assert( max == (a>b -> a : b) )
```

First example with a formal correctness property

We can do model checking, for the first time!

(Historic moment in the course)

Model Checking, First Step: Generate Verifier in C



Command Line Execution

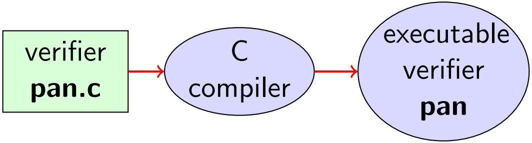
Generate Verifier in C

```
> spin -a maxA.pml
```

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

(plus auxiliary files)

Second Step: Compile To Executable Verifier



Command Line Execution

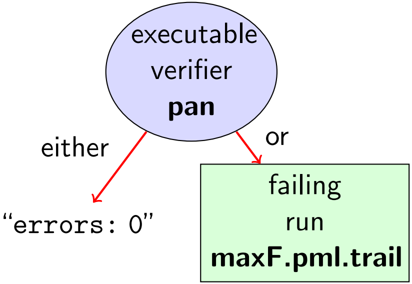
Compile to executable verifier

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

C compiler generates executable verifier **pan**

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

Third Step: Run Verifier (= Model Checking)



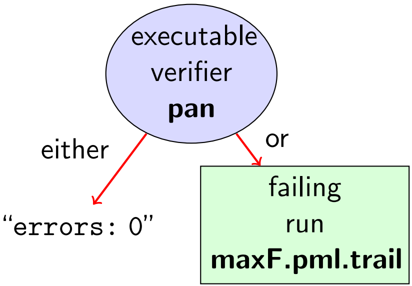
Command Line Execution

Run verifier **pan**

> ./pan or > pan (if current directory in search path)

- prints "errors: 0" (buried in a lot of extra info) ⇒ Correctness Property verified!

Third Step: Run Verifier (= Model Checking)



Command Line Execution

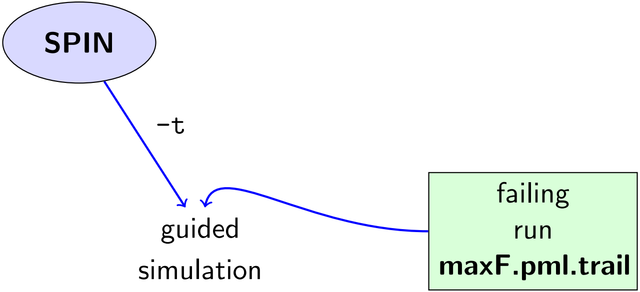
Run verifier **pan**

> ./pan   or   > pan (if current directory in search path)

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

Errors Found: Guided Simulation

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



Command Line Execution

Run verification of a faulty program, and then:

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

Output of Guided Simulation

Can look like:

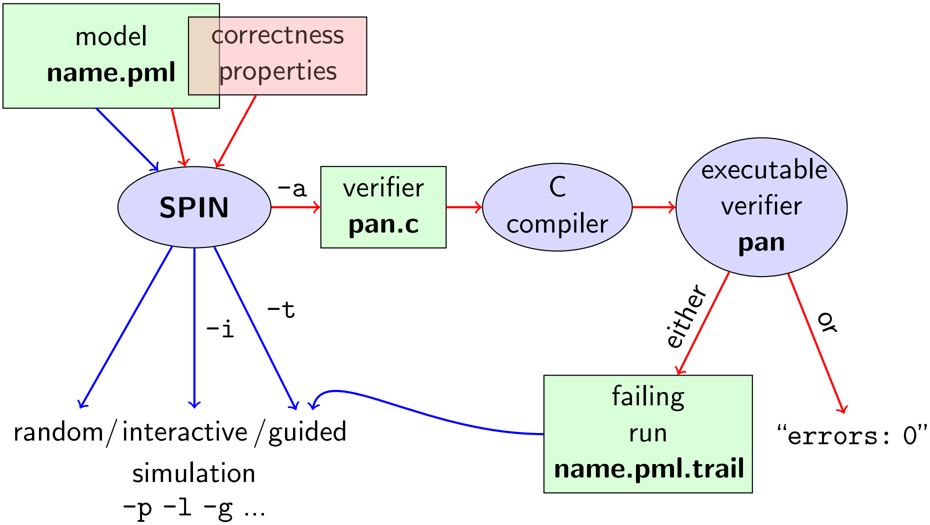
```
  1: proc  0 (P) maxF.pml:7 (state 1)  [a = 1]
                P(0):a = 1
  2: proc  0 (P) maxF.pml:12 (state 6) [b = 1]
                P(0):b = 1
  3: proc  0 (P) maxF.pml:20 (state 13) [((a<=b))]
  3: proc  0 (P) maxF.pml:20 (state 14) [max = b+1]
                P(0):max = 2
spin: maxF.pml:25, Error: assertion violated
spin: text of failed assertion:
     assert((max==( ((a>b)) -> (a) : (b) )))
```

Assignment statements executed in the run

Values of variables whenever updated

What did we do so far?

Following whole cycle (simple example, assertions only)



More 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 `divide1`, come up with assert in `divide2`
- With iSpin: verify `divide2`, inspect trail, analyse, fix, verify

More Examples: Greatest Common Divisor

```
int x = 15, y = 20;
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 verification not possible here

- To express "greatest" property need quantification in expressions

Still, assertions can perform sanity check/partial verification (`gcd.pml`)

- Typical situation when employing model checking

Typical Command Lines

Some patterns for frequently used command line sequences:

Random simulation

`spin name.pml` or `spin -v -l name.pml`

Interactive simulation

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

Model checking

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

and in case of errors:

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

. . . or use iSpin!

generates commands automatically

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

Why SPIN?

Industrial-strength, "mainstream" software model checking

- SPIN targets software, instead of hardware verification
- 2001 ACM Software Systems Award (other winning software systems include: Unix, TCP/IP, WWW, Tcl/Tk, Java)
- Used for safety critical applications
- Annual SPIN user workshops series held since 1995
- Based on standard theory of ω-automata and linear temporal logic

Why SPIN? (Cont'd)

Suitable for Teaching

- PROMELA and SPIN are rather simple to use
- Distributed freely as research tool, well-documented
  actively maintained, large user-base in academia and in industry
- Good portability, quite easy to install, local installation ok
- Understand one representative system well,
  rather than many systems superficially
- Availability of good course book (Ben-Ari)
- Availability of GUI-based front end

Catching A Different Type of Error

Look again at `max2.pml`:

```
byte a, b, max;
select(a: 1 .. 3);
select(b: 1 .. 3);
if
  :: a >= b -> max = a;
  :: b <= a -> max = b;
fi;
printf("maximum␣of␣%d␣and␣%d␣is␣%d\n", a, b, max)
```

Simulate a few times

⇒ strange occasional "`timeout`" message

Generate and execute verifier **pan**

⇒ reports "`errors: 1`"

What is going on here?

Catching A Different Type of Error

Further inspection of **pan** output:

```
...
pan: invalid end state  (at depth 1)
pan: wrote max2.pml.trail
...
```

Legal and Illegal Blocking

There is no guard that covers `a < b` $\Rightarrow$ process blocks!

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 (for example, protocols)

But it is an error if a process blocks while no other process can proceed

$\Rightarrow$ "Deadlock"

In `max2.pml` there exist runs when no process can proceed

Valid End States

Definition (Valid End State)

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

Definition (End Location)

End locations of a process `P` are:

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

End labels not useful in **max2.pml**, but elsewhere, they are
Example: verify `end.pml` (without/with end labels)

Checking for invalid end states can be disabled in `pan` (option `-E`)

There are runs without any end state (e.g., non-terminating loop)

Literature for this Lecture

Ben-Ari Chapter 2
Section 4.7.1
Section 4.7.2