

Tool demonstration:

Spin

Spin

Spin is a model checker which implements the LTL model-checking procedure described previously (and much more besides).

Developed by **Gerard Holzmann** of **Bell Labs**

Has won the prestigious ACM Software System Award in 2001
(other winners include Unix, TeX, Java, TCP/IP, Apache, PostScript, ...)

WWW homepage: <http://spinroot.com>

Literature: Holzmann, [The Spin Model Checker](#)
(available in the library of the computer science building,
Semesterapparat Esparza)

Using Spin

Transition systems described with **Promela** (Protocol Meta Language)

Can describe finite-state systems.

Useful for describing concurrent processes with synchronous and asynchronous communication, variables, advanced datatypes (e.g., records).

Example: Mutual Exclusion algorithm by Dekker

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;

active proctype p0() {
    ...
}

active proctype p1() {
    ...
}
```

Example: Contents of Process p0

```
active proctype p0() {
again:  flag0 = true;
       do
         :: flag1 ->
           if
             :: turn == 1 ->
               flag0 = false;
               (turn != 1) -> flag0 = true;
             :: else -> skip;
           fi
         :: else -> break;
       od;

       crit0 = true; /* critical section */ crit0 = false;

       turn = 1; flag0 = false;
       goto again;
}
```

Process p1: like p0, but swap 0 and 1

Promela syntax (1/5)

Declaration of variables:

```
bit turn;  
bool flag0, flag1;  
bool crit0, crit1;
```

`turn` can assume values `0` or `1`.

`flag1` etc. can assume values `true` or `false`.

initial values: `0` and `false`, respectively

other data types: `byte` or others, e.g. records

Promela syntax (2/5)

Declaration of processes:

```
active proctype p0() {  
  ...  
}
```

`proctype` defines a **process type**

(of which multiple instances may be activated at the same time)

`active` means that initially one instance of this process type is active.

One can also activate multiple instances initially, e.g. by

```
active [2] proctype my_process() {  
  ...  
}
```

Promela syntax (3/5)

Jump labels / assignments / jumps:

```
again:  flag0 = true;  
...  
        goto again;
```

Empty statement:

```
skip
```

Promela syntax (4/5)

Loops:

```
do
  :: flag1 -> ...
  :: else -> break;
od;
```

`flag1` and `else` are so-called **guards**.

Execution continues non-deterministically in a branch whose guard is true.

`else` branch can be executed iff no other guard is true.

`break` leaves the `do` loop.

Promela syntax (5/5)

Branching:

```
if
:: turn == 1 -> ...
:: else -> ...;
fi
```

Syntax and semantics as in `do`, but only one single execution.

```
(turn != 1) -> ...
```

Guarded commands: Blocks execution until `turn` is not equal to `1`.

Using Spin: Example

Let us use Spin to check whether Dekker's algorithm indeed fulfils the mutual exclusion property, i.e. both processes should not be in their critical sections at the same time:

$$G \neg(\text{crit0} \wedge \text{crit1})$$

(interpreting the atomic propositions `crit0` and `crit1` naturally, i.e. they hold in states in which the respective boolean variables are true).

Spin syntax for the property: `[] !(crit0 && crit1)`

Step-by-step instructions

Construct the **promela model** and save it to a file (e.g. `dekker.pml`).

Create a **Büchi automaton** (for the negated property) using Spin's `-f` option:

```
spin -f '![ ] !(crit0 && crit1)' > buechi
```

Construct the **cross product** (two steps):

- Build a C program (`pan`) that computes the product:

```
spin -a -N buechi dekker.pml
```

- Compile the program: `make pan`

Check for emptiness (run the program): `./pan -a -n`

If there is an error, **get the counterexample**: `spin -p -t dekker.pml`

The course homepage has a **script** for all this: [spinLTL](#)

Example: Checking properties

Checking the mutual exclusion property on the example (e.g. using

```
./spinLTL dekker.pml '[ ]!(crit0 && crit1)'
```

yields that the property holds.

Let us try another property: “Whenever p0 tries to enter its critical section, it should eventually succeed.”

Spin syntax for the property: `[] (flag0 -> <> crit0)`

Checking the property using `spinLTL` yields a violation of the property!

Example: Fairness

In the counterexample, p0 cannot enter its critical section because p1 does not get any computation time to set `flag1` back to `false`.

Such a computation is “unfair”.

We can restrict attention to ‘fair’ computations in which both processes get unlimited computation time:

$$(\mathbf{G F} \text{“p0 computes”} \wedge \mathbf{G F} \text{“p1 computes”}) \rightarrow \mathbf{G}(\text{flag0} \rightarrow \mathbf{F} \text{crit0})$$

For the ‘fairness’ part of this property (to the left of the implication) there is a special switch in Spin (see the script [spinFairLTL](#) on the course homepage).

Using [spinFairLTL](#) we manage to verify the property as correct.

More about Spin

See the Spin book! (cf slide 2)

More demonstrations to come during the course.

There is also a graphical interface called [XSpin](#).