

CS477 Formal Software Development Methods

Elsa L. Gunter
2112 SC, UIUC
egunter@illinois.edu
<http://courses.engr.illinois.edu/cs477>

Slides based in part on previous lectures by Mahesh Vishwanathan, and by Gul Agha

April 30, 2014

SPIN Commandline Options

The following are some useful commandline options:

- **-a**: Generate code for project-specific verifier
 - Default: run SPIN as a simulator
- **-p**: Print at each state which process took which step
- **-s**: Print send statements and their effects
- **-r**: Print receive statements and their effects
- **-v**: verbose
- **-nN**: Use *N* as random seed, instead of clock (good for reproducibility)
- **l**: Show changes to local variables
- **g**: Show changes to global variables
- **-uN**: Limit number of steps taken to *N*
- **-t**: Run simulation driven by an error trail
- **-kfilename**: use the trail file stored in *filename*

Common SPIN Workflow

- Write SPIN model; put in file *filename*
- Debug syntax with: `spin -u1000 filename`
- Check assertions, bad end states with:
 - `spin -a filename`
 - `gcc -o pan pan.c`
 - `./pan`
 - Read the output
 - If you have an error trail: `spin -t -p filename`
- To see if an LTL formula does not hold:
 - Put LTL formula in file *ltlfile*
 - `spin -F ltlfile i neverclaimfile`
 - `spin -a -N neverclaimfile filename`
 - `gcc -o pan pan.c`
 - `./pan`
 - Read the output
 - If you have an error trail: `spin -t -p filename`

Traffic Light Example

```
/* File: trafficlight.pml */

mtype = {NS, EW, Red, Yellow, Green};
bit Turn = 0;
mtype Color[2];

proctype Light(bit myId) {
  bit otherId = 1 - myId;
  do
  :: Turn == myId && Color[myId] == Red
  -> Color[myId] = Green
  :: Color[myId] == Green
  -> Color[myId] = Yellow
  :: Color[myId] == Yellow
  -> Color[myId] = Red; Turn = otherId
  od
}
```

```
init { atomic{Color[0] = Red; Color[1] = Red};
      atomic{run Light(0); run Light(1)}
}
```

/* End of File: trafficlight.pml */

Can test this with

```
bash-3.2$ spin -p -l -g -u50 trafficlight.pml
0: proc - (:root:) creates proc 0 (:init:)
1: proc 0 (:init:) trafficlight.pml:18 (state 1) [Color[0] = Red
Color[0] = Red
Color[1] = 0
2: proc 0 (:init:) trafficlight.pml:19 (state 2) [Color[1] = Red
Color[0] = Red
Color[1] = Red
Starting Light with pid 1
3: proc 0 (:init:) creates proc 1 (Light)
3: proc 0 (:init:) trafficlight.pml:20 (state 5) [(run Light(0))
Starting Light with pid 2
4: proc 0 (:init:) creates proc 2 (Light)
4: proc 0 (:init:) trafficlight.pml:20 (state 4) [(run Light(1))
```

LTL to Never Claim

```
bash-3.2$ spin -f '<>(!(Color[0] == Red
|| Color[1] == Red))' >& trafficlightnever.pml
bash-3.2$ cat trafficlightnever.pml
never /* <>(!(Color[0] == Red || Color[1] == Red)) */
T0_init:
do
:: atomic ((!(Color[0] == Red || Color[1] == Red))
-> assert(!(!(Color[0] == Red || Color[1] == Red))))
:: (1) -> goto T0_init
od;
accept_all:
skip
```

Using never Claim in Separate File

To use file containing `never` claim:

```
bash-3.2$ spin -a -N trafficlightnever.pml trafficlight.pml
bash-3.2$ gcc -o pan pan.c
bash-3.2$ ./pan omissions
```

Full statespace search for:

```
never claim          + (never_0)
assertion violations + (if within scope of claim)
acceptance cycles   - (not selected)
invalid end states  - (disabled by never claim)
```

```
State-vector 44 byte, depth reached 26, errors: 0
 13 states, stored
  1 states, matched
 14 transitions (= stored+matched)
  1 atomic steps
hash conflicts:          0 (resolved)
```

```
unreached in proctype Light
./trafficlight.pml:17, state 17, "--end-"
(1 of 17 states)
unreached in init
(0 of 4 states)
unreached in claim never_0
./trafficlightnever.pml:9, state 10, "--end-"
(1 of 10 states)
```

```
pan: elapsed time 0.02 seconds
pan: rate          650 states/second
```

- Process Light never ends, so its end state never reached

Properties (1)

- Model checking tools **automatically** verify whether $M \models \phi$ holds, where M is a (finite-state) **model** of a system and **property** ϕ is stated in some formal notation.
- With SPIN one may **check** the following type of properties:
 - **deadlocks** (invalid endstates)
 - **assertions**
 - **unreachable code**
 - **LTL formulae**
 - **liveness properties**
 - non-progress cycles (livelocks)
 - acceptance cycles



Historical Classification

Properties (2)

safety property

- “nothing bad ever happens”
- **invariant**
x is always less than 5
- **deadlock freedom**
the system never reaches a state where no actions are possible
- **SPIN**: find a trace leading to the “bad” thing. If there is **not** such a trace, the property is **satisfied**.

liveness property

- “something good will eventually happen”
- **termination**
the system will eventually terminate
- **response**
if action X occurs then eventually action Y will occur
- **SPIN**: find a (infinite) loop in which the “good” thing does not happen. If there is **not** such a loop, the property is **satisfied**.



Properties (3)

- LTL formulae are used to specify liveness properties.
LTL \equiv **propositional logic** + **temporal operators**
 - **[] P** always P
 - **<> P** eventually P
 - **P U Q** P is true until Q becomes true
- Some LTL patterns
 - invariance **[] (p)**
 - response **[] ((p) -> (<> (q)))**
 - precedence **[] ((p) -> ((q) U (x)))**
 - objective **[] ((p) -> <>((q) || (x)))**

Xspin contains a special “LTL Manager” to edit, save and load LTL properties.



Properties (4)

- Suggested **further reading** (on temporal properties):
 - [Bérard et. al. 2001]**
 - Textbook on **model checking**.
 - One part of the book (six chapters) is devoted to “Specifying with Temporal Logic”.
 - Also available in **French**.
 - [Dwyer et. al. 1999]**
 - **classification** of temporal logic properties
 - **pattern-based** approach to the **presentation, codification** and **reuse** of property specifications for finite-state verification.

Note: although this tutorial focuses on how to construct an effective Promela model M , the definition of the **set of properties** which are to be verified is **equally important!**



Invariance

[!P]

- [!P] where **P** is a **state property**
 - safety property
 - **invariance** \equiv global **universality** or global **absence** [Dwyer et. al. 1999]:
 - 25% of the properties that are being checked with model checkers are **invariance properties**
 - BTW, 48% of the properties are **response properties**
 - examples:
 - [! **af**lag
 - [! **mutex** != 2
- SPIN supports (at least) **7 ways** to **check** for invariance.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



81

Elsa L. Gunter

CS477 Formal Software Development Method

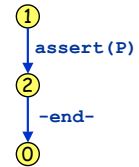
/ 19

variant 1+2 - monitor process (single **assert**)

[!P]

- **proposed** in SPIN's documentation
- **add** the following **monitor** process to the Promela model:

```
active proctype monitor()
{
  assert(P);
}
```



- Two variations:

- 1. monitor process is created **first**
- 2. monitor process is created **last**

If the monitor process is created **last**, the **-end-** transition will be executable after executing **assert(P)**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



82

Elsa L. Gunter

CS477 Formal Software Development Method

/ 19

variant 3 - guarded monitor process

[!P]

- **Drawback** of solution "1+2 monitor process" is that the **assert** statement is enabled in **every** state.

```
active proctype monitor()
{
  assert(P);
}

active proctype monitor()
{
  atomic {
    !P -> assert(P);
  }
}
```

- The **atomic** statement **only** becomes **executable** when **P** itself is **not true**.

We are searching for a state where **P** is not true. If it does not exist, [!P] is true.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



83

Elsa L. Gunter

CS477 Formal Software Development Method

/ 19

variant 4 - monitor process (**do assert**)

[!P]

- From an operational viewpoint, the following monitor process **seems less effective**:

```
active proctype monitor()
{
  do
  :: assert(P)
  od
}
```



- But the number of **states** is clearly advantageous.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



84

Elsa L. Gunter

CS477 Formal Software Development Method

/ 19

variant 5 - never claim (**do assert**)

[!P]

- also **proposed** in SPIN's documentation

```
never {
  do
  :: assert(P)
  od
}
```

SPIN will synchronise the **never claim** automaton with the automaton of the system. SPIN uses never claims to verify **LTL formulae**.

... but SPIN will issue the following unnerving **warning**:

warning: for p.o. reduction to be valid the never claim must be stutter-closed (never claims generated from LTL formulae are stutter-closed)

... and this never claim has not been generated...



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



85

Elsa L. Gunter

CS477 Formal Software Development Method

/ 19

variant 6 - LTL property

[!P]

- The **logical** way...
- SPIN translates the **LTL formula** to an accepting **never claim**.

```
never { ![]P
  TO_init:
  if
  :: (!P) -> goto accept_all
  :: (1) -> goto TO_init
  fi;
  accept_all:
  skip
}
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



86

Elsa L. Gunter

CS477 Formal Software Development Method

/ 19

variant 7 - unless {!P -> ...}

[!P]

- Enclose the **body** of (at least) one of the processes into the following **unless** clause:

```
{ body } unless { atomic { !P -> assert(P) ; } }
```

- Discussion

- + **no extra process** is needed: saves 4 bytes in state vector
- + **local variables** can be used in the property P
- definition of the process has to be **changed**
- the **unless** construct can **reach** inside **atomic** clauses
- **partial order reduction** may be **invalid** if rendez-vous communication is used within **body**
- the **body** is **not allowed** to end

This is quite restrictive

Note: **disabling partial reduction (-DNOREDUCE)** may have severe **negative** consequences on the **effectiveness** of the verification run.

