

Rigorous Methods for Software Engineering (F21RS-F20RS) Spin – Formal Analysis (Part 2)

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- ▶ Introduce temporal logic.
- ▶ Focus on SPIN's temporal reasoning capabilities, *i.e.* model checking.

The Story So Far ...

- ▶ Verifying properties with respect to particular points within a process execution (local assertions) or across the whole execution of a system (global assertions).
- ▶ Verifying properties with respect to complete execution cycles, both desirable (end-states & no non-progression cycles) and undesirable (acceptance cycles).
- ▶ But what if we want to reason about how properties change over time, *i.e.* reason about the temporal ordering of events? This calls for **temporal logic**.

Linear Temporal Logic (LTL)

- ▶ LTL = Propositional Logic + Temporal Operators
- ▶ Propositional constants:
 - true, false
 - any name that starts with a lowercase letter
- ▶ Propositional operators:
 - && conjunction || disjunction
 - > implication ! negation
- ▶ Temporal operators:
 - [] always <> eventually U until

Some Generic Temporal Properties

- ▶ Invariance (safety): $\square p$

During any execution trace all states satisfy p , e.g.

$\square!(\text{doors}==\text{open} \ \&\& \ \text{lift}==\text{moving})$

- ▶ Response: $\square(p \rightarrow \langle \rangle q)$

Every state that satisfies p is eventually followed by a state that satisfies q , e.g. $\square(\text{call_lift} \rightarrow \langle \rangle(\text{lift_arrives}))$

- ▶ Precedence: $\square(p \rightarrow (q \cup r))$

Every state that satisfies p is followed by a sequence of states that satisfy q and the sequence is terminated with a state that satisfies r , e.g.

$\square(\text{start_lift} \rightarrow(\text{lift_running} \cup \text{stop_running}))$

Temporal Reasoning in iSPIN

- ▶ Step 1: Embed LTL formulas in your Promela program, i.e.

```
ltl [ <name> ] { <formula> }
```

For example:

```
ltl p1 { [] p }
```

```
ltl p2 { [] (p -> <> q) }
```

```
ltl p3 { [] (p -> (q U r)) }
```

- ▶ Step 2: Propositional conditions are defined via macros, e.g.

```
#define p (x > y)
```

```
#define q (len(in_data) < max)
```

```
#define r ( x > 0 && x < max)
```

- ▶ Step 3: Within the **Verification** tab select **use claim** (see **Never Claims** section) and select **acceptance cycles** (see **Liveness** section), then click the **Run** button.

Note: LTL formula can be selectively enabled via the **claim name (opt)** field, i.e. enter the ID of a LTL formula, e.g. p2.

TrainWare Revisited: Safety Property

```
[] (len(TunnelAB) < 2 &&  
    len(TunnelBC) < 2 &&  
    len(TunnelCD) < 2 &&  
    len(TunnelDA) < 2)
```

This property should hold on **all executions**, *i.e.* always the case that none of the tunnels is occupied by more than one train.

```
#define q (len(TunnelAB) < 2 &&  
        len(TunnelBC) < 2 &&  
        len(TunnelCD) < 2 &&  
        len(TunnelDA) < 2)
```

```
ltl p1 { [] q }
```

Note that LTL formula cannot make use of `empty`, `nempty`, `full`, `nfull`.

TrainWare Revisited: Verification Set-up

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View Simulate / Replay **Verification** Swarm Run <Help> Save Session Restore Session <Quit>

Safety	Storage Mode	Search Mode
<input type="radio"/> safety	<input checked="" type="radio"/> exhaustive	<input checked="" type="radio"/> depth-first search
<input checked="" type="checkbox"/> + invalid endstates (deadlock)	<input type="checkbox"/> + minimized automata (slow)	<input checked="" type="checkbox"/> + partial order reduction
<input checked="" type="checkbox"/> + assertion violations	<input type="checkbox"/> + collapse compression	<input type="checkbox"/> + bounded context switching
<input type="checkbox"/> + xr/xs assertions	<input type="radio"/> hash-compact <input type="radio"/> bitstate/supertrace	with bound: 0
Liveness	Never Claims	<input type="checkbox"/> + iterative search for short trail
<input type="radio"/> non-progress cycles	<input type="radio"/> do not use a never claim or ltl property	<input type="radio"/> breadth-first search
<input checked="" type="radio"/> acceptance cycles	<input checked="" type="radio"/> use claim	<input checked="" type="checkbox"/> + partial order reduction
<input type="checkbox"/> enforce weak fairness constraint	claim name (opt):	<input checked="" type="checkbox"/> report unreachable code
	<input type="button" value="Run"/> <input type="button" value="Stop"/>	<input type="button" value="Save Result in: pan.out"/>

```
22
23 chan TunnelAB = [2] of { byte };
24 chan TunnelBC = [2] of { byte };
25 chan TunnelCD = [2] of { byte };
26 chan TunnelDA = [2] of { byte };
27
28 #define q (len(TunnelAB) < 2 && len(TunnelBC) < 2 && len(TunnelCD) < 2)
29
30 ltl p1 { [] q }
31
32 proctype Station(chan in_track, out_track)
33 {
34     byte train;
35
36     do
37     :: in_track?train; out_track!train
38     od
39 }
40
```


TrainWare Revisited: Verification Run and Result

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View Simulate / Replay Verification Swarm Run <Help> Save Session Restore Session <Quit>

Safety	Storage Mode	Search Mode
<input type="radio"/> safety	<input checked="" type="radio"/> exhaustive	<input checked="" type="radio"/> depth-first search
<input checked="" type="checkbox"/> + invalid endstates (deadlock)	<input type="checkbox"/> + minimized automata (slow)	<input checked="" type="checkbox"/> + partial order reduction
<input checked="" type="checkbox"/> + assertion violations	<input type="checkbox"/> + collapse compression	<input type="checkbox"/> + bounded context switching
<input type="checkbox"/> + xr/xs assertions	<input type="checkbox"/> hash-compact <input type="checkbox"/> bitstate/supertrace	with bound: 0
Liveness	Never Claims	<input type="checkbox"/> + iterative search for short trail
<input type="checkbox"/> non-progress cycles	<input type="checkbox"/> do not use a never claim or ltl property	<input type="radio"/> breadth-first search
<input checked="" type="radio"/> acceptance cycles	<input checked="" type="radio"/> use claim	<input checked="" type="checkbox"/> + partial order reduction
<input type="checkbox"/> enforce weak fairness constraint		<input checked="" type="checkbox"/> report unreachable code

Run Stop Save Result in: pan.out

Show Error Trapping Options Show Advanced Parameter Settings

```
**/
22
23 chan TunnelAB = [2] of { byte };
24 chan TunnelBC = [2] of { byte };
25 chan TunnelCD = [2] of { byte };
26 chan TunnelDA = [2] of { byte };
27
28 #define q (len(TunnelAB) < 2 && len(TunnelBC) < 2 && len(TunnelCD) < 2)
29
30 ltl p1 { [] q }
31
32 proctype Station(chan in_track, out_track)
33 {
34     byte train;
35
36     do
37     :: in_track?train; out_track!train
38     od
39 }
40
41 proctype Setup(chan track; byte train)
42 {
43     track!train;
44 }
45
46 init { atomic{
47     run Setup(TunnelBC, 1); /* introduce train 1 before station C */
48     run Setup(TunnelDA, 2); /* introduce train 2 before station A */
49
50     run Station(TunnelDA, TunnelAB); /* station A */
51     run Station(TunnelAB, TunnelBC); /* station B */
52     run Station(TunnelBC, TunnelCD); /* station C */
53     run Station(TunnelCD, TunnelDA); /* station D */
54 }
```

```
pan:1: assertion violated !(((q_len(TunnelAB)<2)&&(q_len(TunnelBC)<2))&&(q_len(TunnelCD)<2)))) (at depth 107)
pan: wrote trainware.pml.trail

(Spin Version 6.4.3 -- 16 December 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim + (p1)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 112 byte, depth reached 110, errors: 1
58 states, stored
12 states, matched
70 transitions (= stored+matched)
5 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.008 equivalent memory usage for states (stored*(State-vector + overhead))
0.290 actual memory usage for states
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
128.730 total actual memory usage

pan: elapsed time 0 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Random, with seed: 123
 Interactive (for resolution of all nondeterminism)
 Guided, with trail: trainware.pml.trail browse
 Initial steps skipped: 0
 maximum number of steps: 10000
 Track Data Values (this can be slow)

blocks new messages
 loses new messages
 MSC+stmnt
 MSC max text width: 20
 MSC update delay: 25

Output Filtering (reg. exps.) (ReRun)
 process ids:
 queue ids:
 var names:
 tracked variable:
 track scaling:

Background command executed:
 spin -p -s -r -X -v -n123 -l -g -k trainware.pml
 tail -u10000 trainware.pml

Save in: msc.ps

```

28 #define q (len(TunnelAB) < 2 && len(TunnelBC)
29 < 2 && len(TunnelCD) < 2)
30 !ll p1 { [] q }
31
32 proctype Station(chan in_track, out_track)
33 {
34   byte train;
35
36   do
37     :: in_track?train; out_track!train
38   od
39 }
40
41 proctype Setup(chan track; byte train)
42 {
43   track!train;
44 }
45
46 init { atomic{
47   run Setup(TunnelBC, 1); /* introduce train 1
48   before station C */
49   run Setup(TunnelDA, 2); /* introduce train 2
50   before station A */
51
52   run Station(TunnelDA, TunnelAB); /* station
53   A */
54   run Station(TunnelAB, TunnelBC); /* station
55   B */
56   run Station(TunnelBC, TunnelCD); /* station
57   C */
58   run Station(TunnelCD, TunnelDA); /* station
59   D */
60 }
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
  
```

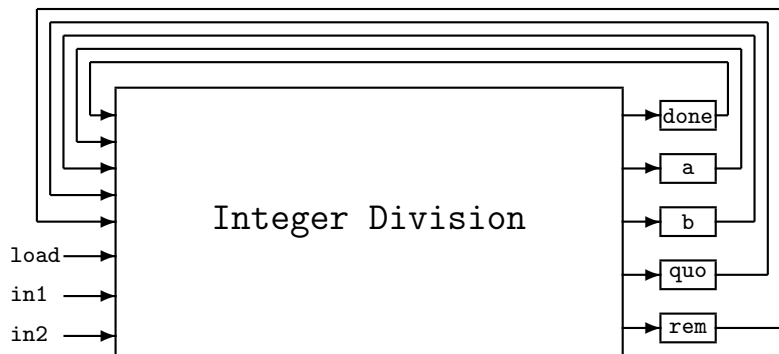
```

[variable values, step 105]
Station(3):train = 1
Station(4):train = 1
Station(5):train = 1
Station(6):train = 1
90: proc 5 (Station:1) trainware.pml:37 (state 1) [in_track?train]
91: proc 4 (Station:1) trainware.pml:37 (state 1) [out_track!train]
92: proc - (p1:1) spin_nvr.tmp:4 (state 4) [(1)]
93: proc 4 (Station:1) trainware.pml:37 (state 2) [out_track!train]
94: proc - (p1:1) spin_nvr.tmp:4 (state 4) [(1)]
95: proc 5 (Station:1) trainware.pml:37 (state 1) [in_track?train]
96: proc - (p1:1) spin_nvr.tmp:4 (state 4) [(1)]
97: proc 5 (Station:1) trainware.pml:37 (state 2) [out_track!train]
98: proc - (p1:1) spin_nvr.tmp:4 (state 4) [(1)]
  
```

```

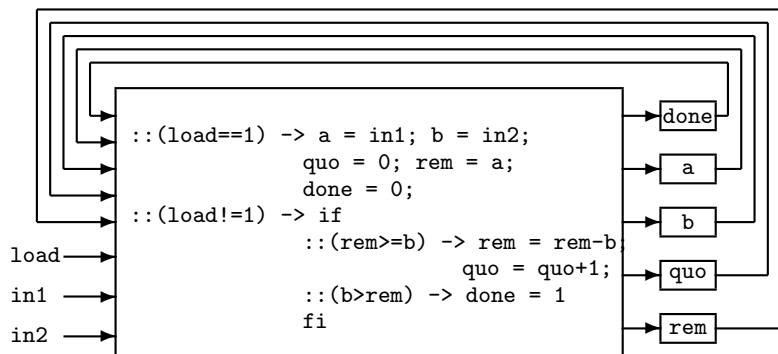
[queues, step 107]
q 1 :: (TunnelBC):
q 2 :: (TunnelDA):
q 3 :: (TunnelAB):
q 4 :: (TunnelCD): [2][1]
  
```

Modelling Hardware



Note: based upon an example by Mike Gordon (Cambridge Computer Lab, <http://www.cl.cam.ac.uk/users/mjcg>).

Modelling Input-Output Relation



Complete Model of Division Algorithm

```
byte in1, in2;
byte a, b, quo, rem;
bit  load = 0, done = 1;

proctype quo_rem()
{
    do
        ::(load == 1) -> a = in1; b = in2;
                        quo = 0; rem = a; done = 0;
        ::(load != 1) -> if
                        :: (rem >= b) -> rem = rem-b;
                        quo = quo+1;
                        :: (b > rem)  -> done = 1
                        fi
    od
}
```

Modelling Hardware Environment

- ▶ Environment (env) initiates register (a, b, quo, rem) initialization by setting load to 1.
- ▶ While load is 1, hardware (quo_rem) sets registers using the input values (in1, in2), done is set to 0 when complete.
- ▶ Environment (env) initiates calculation by setting load to 0, load is held at this value until done becomes 1.

```
proctype env()  
{  
  in1 = 7;  in2 = 2; load = 1;      /* init inputs */  
  done == 0; load = 0; done == 1; /* read results */  
  printf("quotient = %d\n", quo);  
  printf("remainder = %d\n", rem)  
}  
init { atomic{ run quo_rem(); run env() } }
```

Verifying Responsiveness

- ▶ Desired property:
 - ▶ In every state in which `load` is 1, `a` equals `in1` and `b` equals `in2`, then eventually `done` will become 1 and the registers will satisfy $(a == ((quo * b) + rem))$.
 - ▶ $\square((load == 1 \ \&\& \ in1 == a \ \&\& \ in2 == b) \rightarrow \langle \rangle (done == 1 \ \&\& \ a == ((quo * b) + rem)))$
- ▶ Verification failure:
 - ▶ LTL verifier will fail to prove this property because SPIN's default execution model does not guarantee **fairness**.
 - ▶ In particular, if `env` never gets to set `load` to 0 then the calculation will never progress beyond the initialization phase.

Fairness

- ▶ Fairness is a special case of liveness and relates to the how the underlying process scheduler deals with contention, *i.e.* clients competing for the same computational resource.
- ▶ Notions of fairness:
 - ▶ Weak-fairness (just): a process that continuously makes a request will eventually be serviced.
 - ▶ Strong-fairness (compassionate): a process that makes a request infinitely often will eventually be serviced.

Specifying Weak Fairness in SPIN

- ▶ SPIN supports a weak-fairness model that can be selected via the “Liveness” panel of the “Verification” tab of iSPIN, i.e. “enforce weak fairness constraint”.
- ▶ Alternatively, the weak-fairness requirement can be expressed explicitly within the LTL property:

```
[ ] (done == 0 -> load == 0) ->  
  [ ] ((load == 1 && in1 == a && in2 == b) ->  
    <> (done == 1 && a == ((quo * b) + rem)))
```

Note that the extra condition ensures that whenever done is set to 0 then load will be 0, *i.e.* the env process will not be continuously blocked. Of course it is up to the implementor to ensure this assumption becomes a reality!

Integer Division: Verification Result

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View Simulate / Replay **Verification** Swarm Run <Help> Save Session Restore Session <Quit>

Safety	Storage Mode	Search Mode
<input type="radio"/> safety	<input checked="" type="radio"/> exhaustive	<input checked="" type="radio"/> depth-first search
<input checked="" type="checkbox"/> + invalid endstates (deadlock)	<input type="checkbox"/> + minimized automata (slow)	<input checked="" type="checkbox"/> + partial order reduction
<input checked="" type="checkbox"/> + assertion violations	<input type="checkbox"/> + collapse compression	<input type="checkbox"/> + bounded context switching
<input type="checkbox"/> + xr/xs assertions	<input type="radio"/> hash-compact <input type="radio"/> bitstate/supertrace	with bound: 0
Liveness	Never Claims	<input type="checkbox"/> + iterative search for short trail
<input type="radio"/> non-progress cycles	<input type="radio"/> do not use a never claim or ltl property	<input type="radio"/> breadth-first search
<input checked="" type="radio"/> acceptance cycles	<input checked="" type="radio"/> use claim	<input checked="" type="checkbox"/> + partial order reduction
<input type="checkbox"/> enforce weak fairness constraint	claim name (opt):	<input checked="" type="checkbox"/> report unreachable code

Run Stop Save Result in: pan.out Show Error Trapping Option

```
1 byte in1, in2;
2 byte a, b, quo, rem;
3 bit load = 0, done = 1;
4
5 #define p (done == 0)
6 #define q (load == 0)
7 #define r (load == 1)
8 #define s ((in1 == a) && (in2 == b))
9 #define t (done == 1)
10 #define u (a == ((quo * b) + rem))
11
12 ltl p1 { [] (p -> q) -> [] ((r && s) -> <> (t && u)) }
13
14 proctype quo_rem()
15 {
16   do
17     :: (load == 1) -> a = in1; b = in2;
18     quo = 0; rem = a; done = 0;
19   :: (load != 1) -> if
20     :: (rem >= b) -> rem = rem-b; quo = quo+1
21     :: (b > rem) -> done = 1
22   fi
23 od
```

128.925 total actual memory usage

```
unreached in proctype quo_rem
  quo_rem_ltl.pml:21, state 12, "done = 1"
  quo_rem_ltl.pml:24, state 18, "-end-"
  (2 of 18 states)
unreached in proctype env
  quo_rem_ltl.pml:28, state 5, "load = 0"
  quo_rem_ltl.pml:28, state 6, "((done==1))"
  quo_rem_ltl.pml:29, state 7, "printf('quotient = %d\n',quo)"
  quo_rem_ltl.pml:30, state 8, "printf('remainder = %d\n',rem)"
  quo_rem_ltl.pml:30, state 9, "-end-"
  (5 of 9 states)
unreached in init
  (0 of 4 states)
unreached in claim p1
  spin_nvr.tmp:10, state 13, "-end-"
  (1 of 13 states)
```

pan.depuration: 0 seconds

No errors found -- did you verify all claims?

Summary

Learning outcomes:

- ▶ To be able to understand & write temporal properties expressed in LTL.
- ▶ To be able to use iSPIN to verify temporal properties of system models.
- ▶ To understand the notion of fairness and how it relates to the behaviour of a system model.

Next lecture: *How a Model Checker Works.*

Recommended reading:

- ▶ “The Model Checker SPIN”, G.J. Holzmann, IEEE Transactions on Software Engineering, Vol 23 (5), 1997. [available via <http://spinroot.com/spin/theory.html>]