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): `[]p`

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

  `[]!(doors==open && lift==moving)`

- Response: `[](p -> <>q)`

  Every state that satisfies `p` is eventually followed by a state that satisfies `q`, *e.g* `[](call_lift -> <>(lift_arrives))`

- Precedence: `[](p -> (q U 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.*

  `[](start_lift ->(lift_running U 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

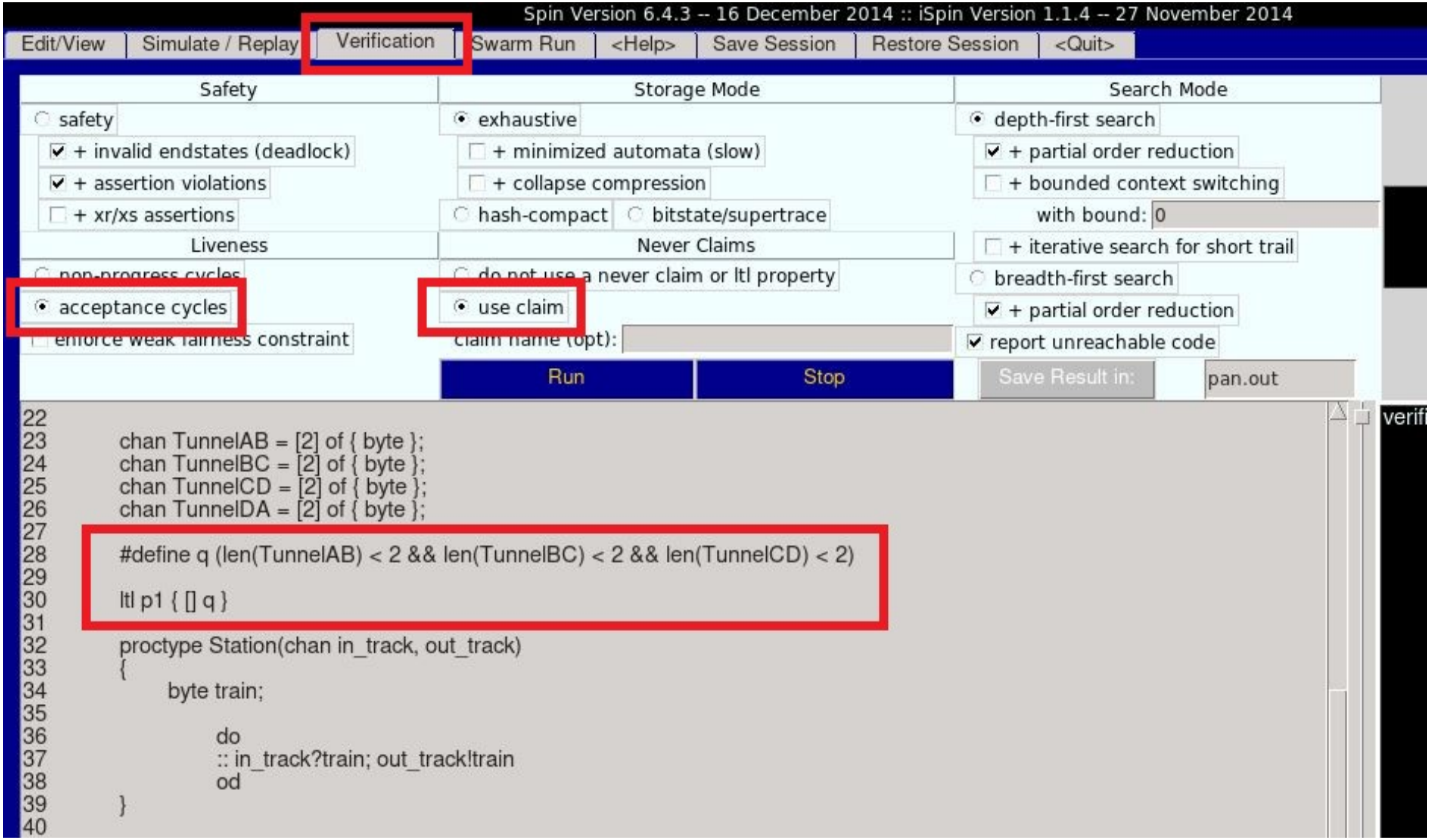
```
[] (len(TunnelAB) < 2 &&
    len(TunnelBC) < 2 &&
    len(TunnelCD) < 2 &&
    len(TunnelDA) < 2)
```

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

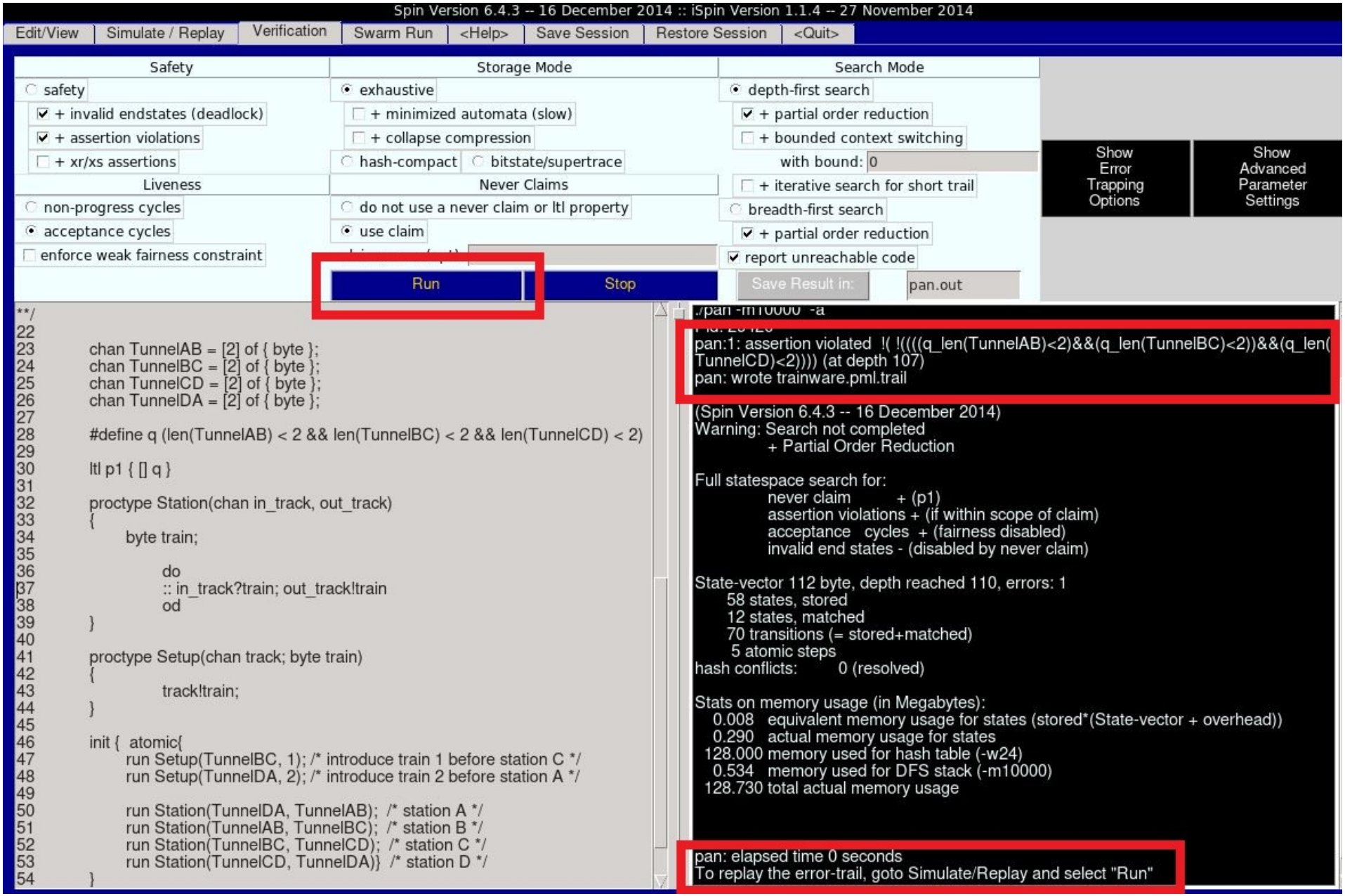
```
#define q  (len(TunnelAB) < 2 &&
            len(TunnelBC) < 2 &&
            len(TunnelCD) < 2 &&
            len(TunnelDA) < 2)

ltl p1 { [] q }
```

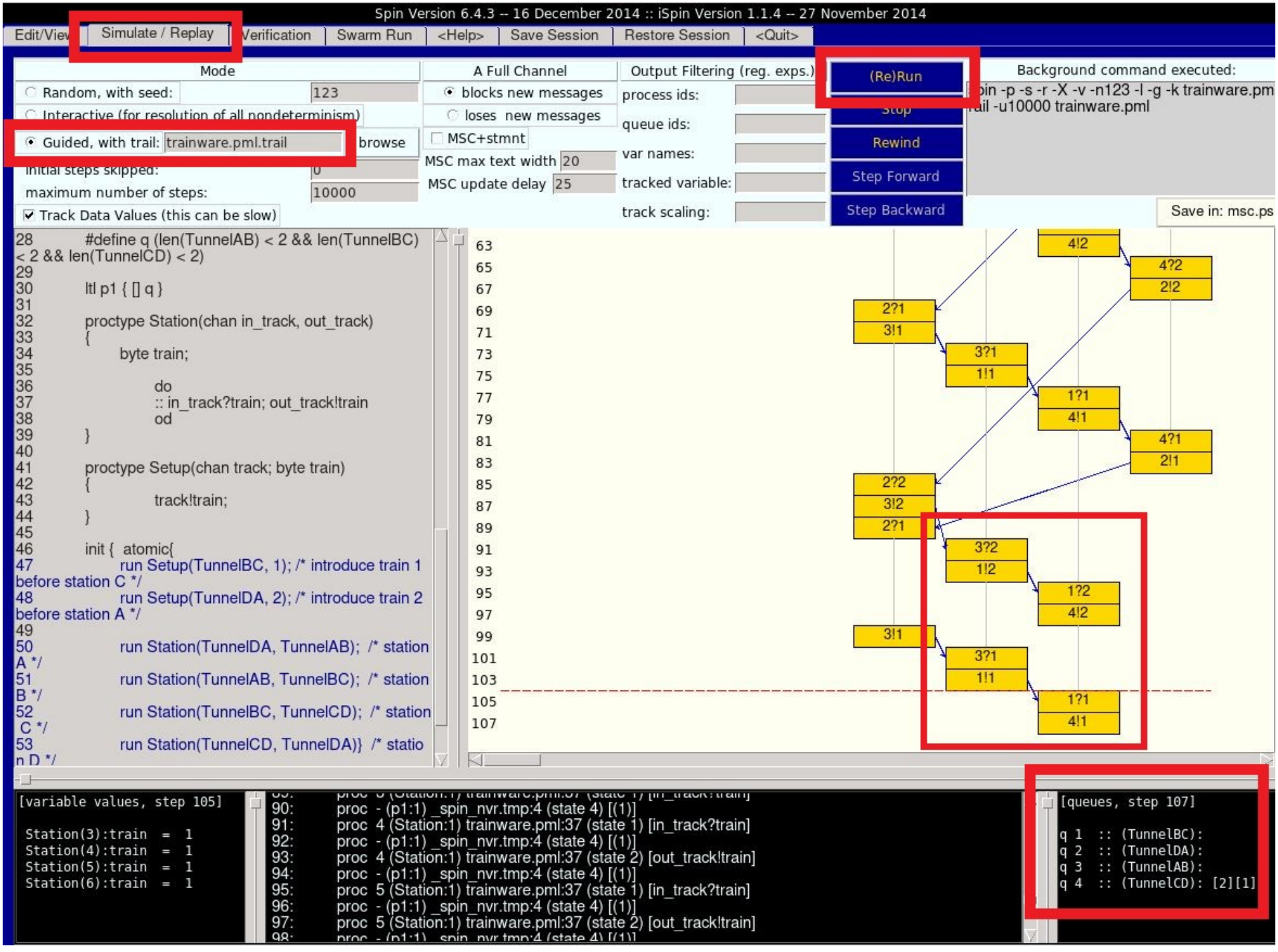
Note that LTL formula cannot make use of `empty`, `nempty`, `full`, `nfull`.

TrainWare Revisited: Verification Set-up

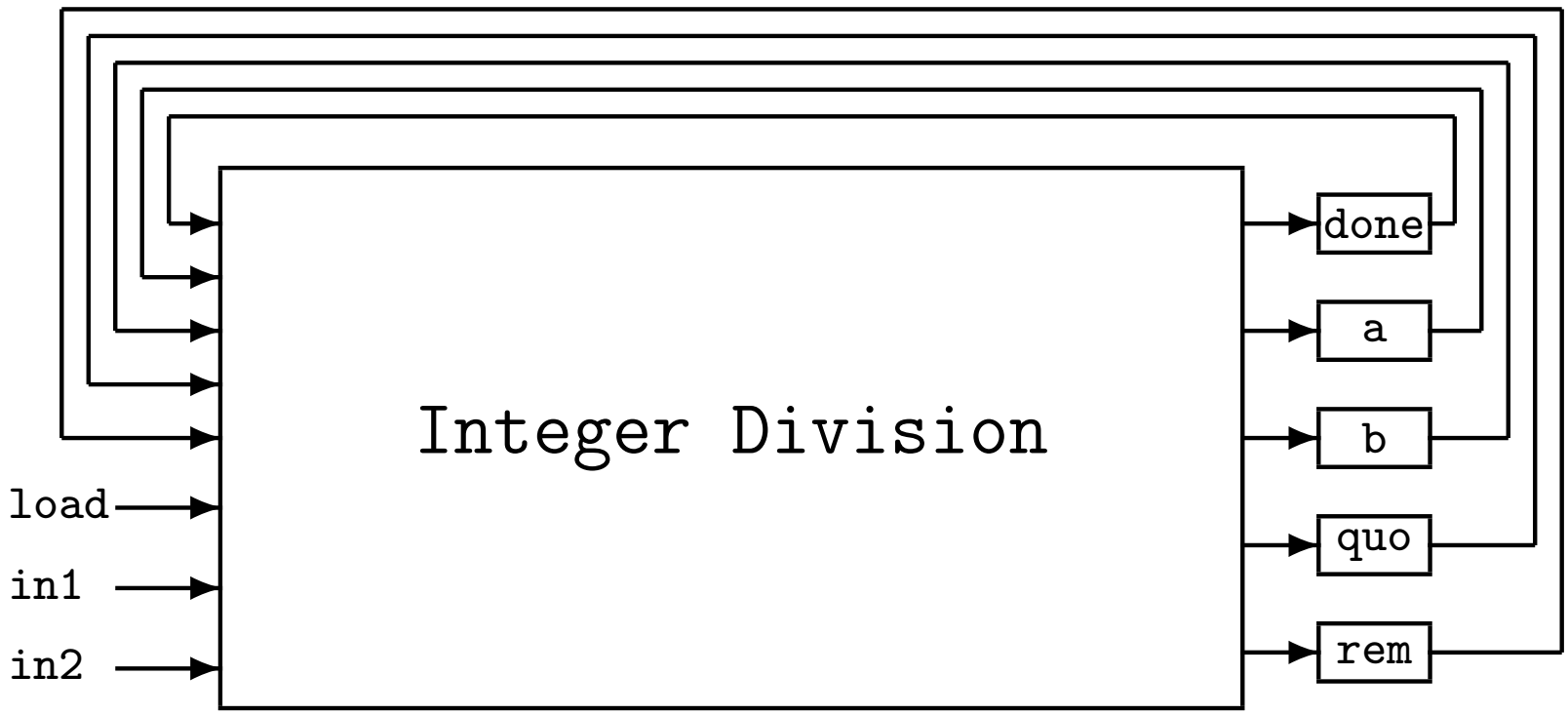


TrainWare Revisited: Verification Run and Result



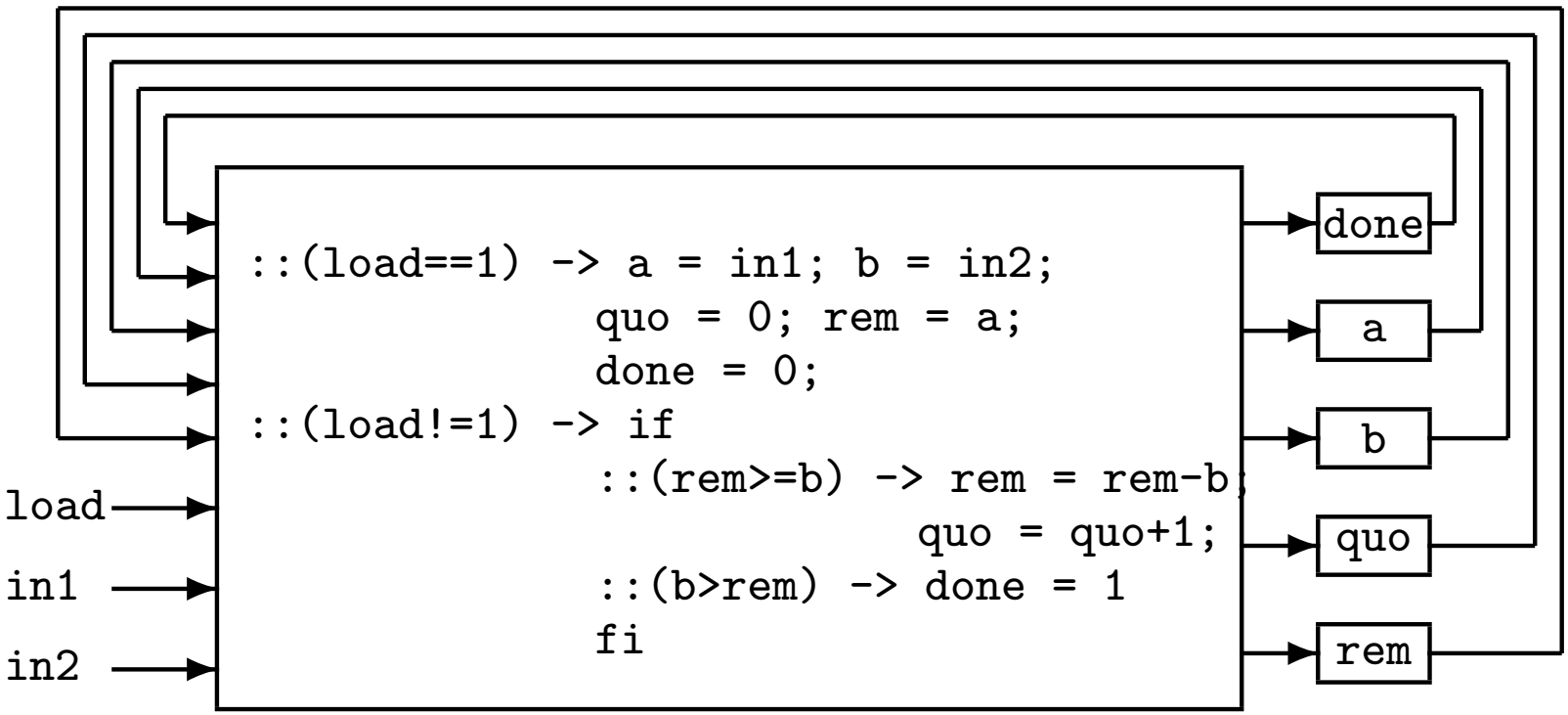

Spin Version 6.4.3 -- 16 December 2014 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Mode
Random, with seed:
123
Interactive (for resolution of all nondeterminism)
Guided, with trail:
trainware.pml.trail
browse
Initial steps skipped:
0
maximum number of steps:
10000
Track Data Values (this can be slow)
A Full Channel
blocks new messages
loses new messages
MSC+stmnt
MSC max text width
20
MSC update delay
25
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:
(Re)Run
Stop
Rewind
Step Forward
Step Backward
Background command executed:
spin -p -s -r -X -v -n123 -l -g -k trainware.pml
trail -u10000 trainware.pml
Save in: msc.ps
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 */
63
65
67
69
71
73
75
77
79
81
83
85
87
89
91
93
95
97
99
101
103
105
107
4!2
4?2
2!2
2?1
3!1
3?1
1!1
1?1
4!1
4?1
2!1
2?2
3!2
2?1
3?2
1!2
1?2
4!2
3!1
3?1
1!1
1?1
4!1
[variable values, step 105]
Station(3):train = 1
Station(4):train = 1
Station(5):train = 1
Station(6):train = 1
90: proc - (p1:1) _spin_nvr.tmp:4 (state 4) [(1)]
91: proc 4 (Station:1) trainware.pml:37 (state 1) [in_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]
[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))`.
  - `[]((load == 1 && in1 == a && in2 == b) ->`
    `<> (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.

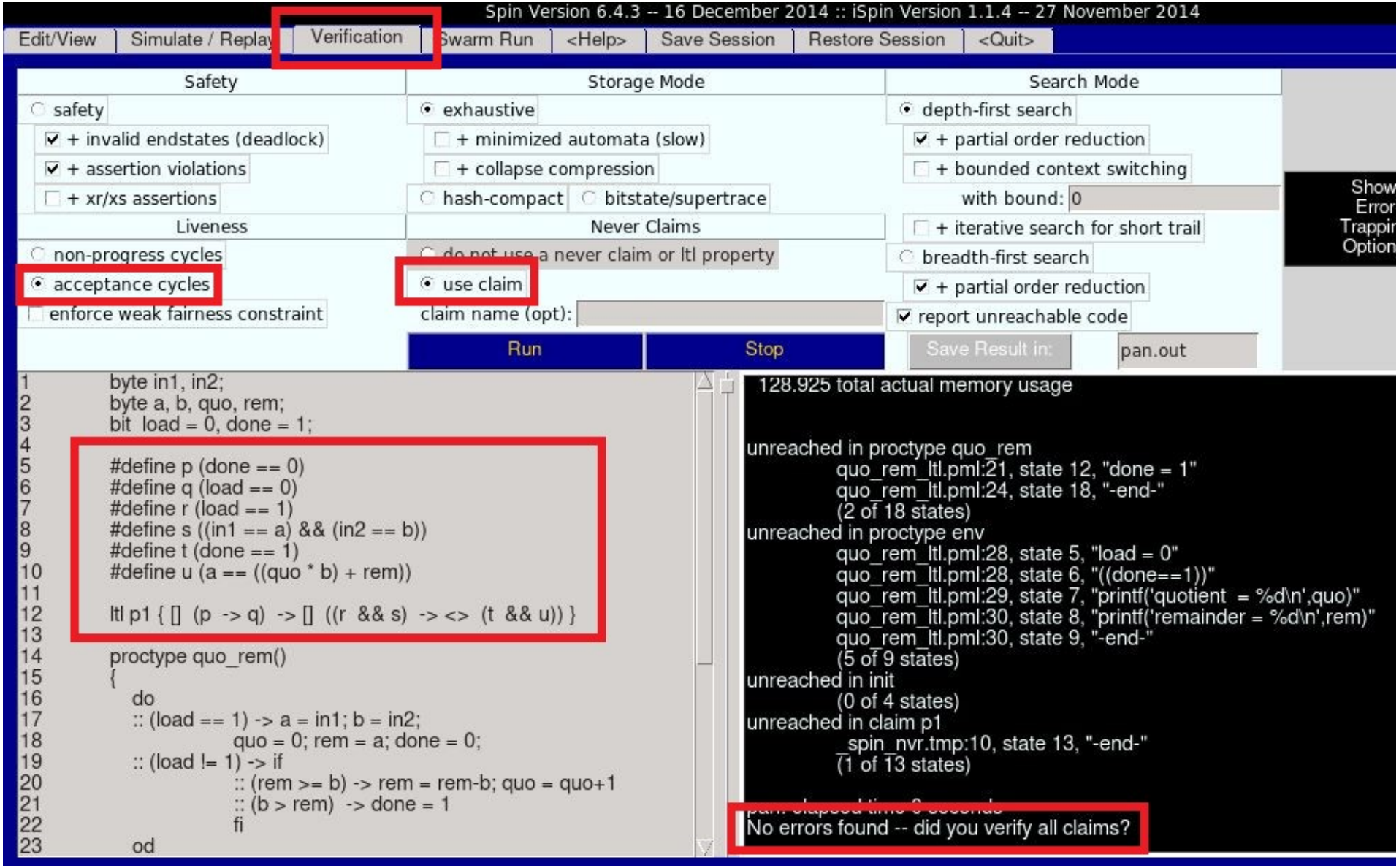
Specifying Weak Fairness in SPIN

- 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



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`]