Rigorous Methods for Software Engineering (F21RS-F20RS) Spin – Formal Analysis (Part 1)

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

Overview

- Introduce **Spin**'s formal analysis capabilities via **iSpin**.
- Focus upon **Spin**'s support for assertion verification and deadlock/livelock detection.

Going Beyond Simulation

- So far we have looked at **Spin**'s simulation capabilities, *i.e.* the random (or interactive) exploration of the state space.
- While simulation is very useful at giving earlier feedback on a design, it can **never prove** that a design is bug free, *i.e.* correct with respect to its specification.
- **Spin**'s formal analysis capabilities provide such a correctness guarantee, it has been said:

  "... formal analysis reaches the parts simulation can not reach."
- **Spin**'s formal analysis corresponds to the fast and exhaustive search of the state space.

Formal Analysis within Spin

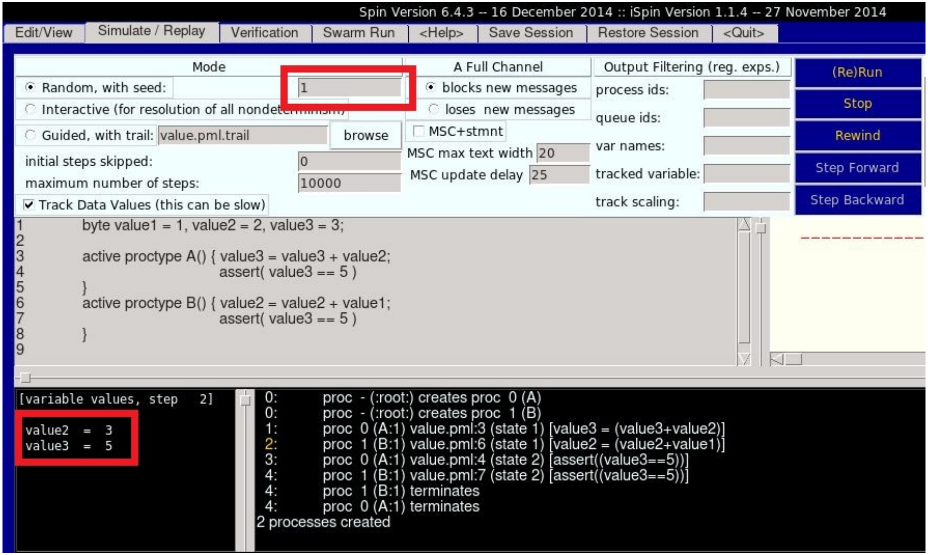
- Assertion verification:
  - local process assertions
  - global system assertions
- Validation labels:
  - success without termination
  - productive progress
- Temporal verification:
  - Linear Temporal Logic (LTL)
    (also referred to as linear-time temporal logic)

Local Assertions Revisited

```
byte value1 = 1, value2 = 2, value3 = 3;

active proctype A() { value3 = value3 + value2;
                                 assert( value3 == 5 )
}
active proctype B() { value2 = value2 + value1;
                                 assert( value3 == 5 )
}
```

- Using **Spin**'s simulator, will assertion checking succeed or fail?
- Let's take a look at a couple of simulation runs ...

First Simulation Run



Random, with seed: 1 [✓]
Interactive (for resolution of all nondeterminism)
Guided, with trail: value.pml.trail
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

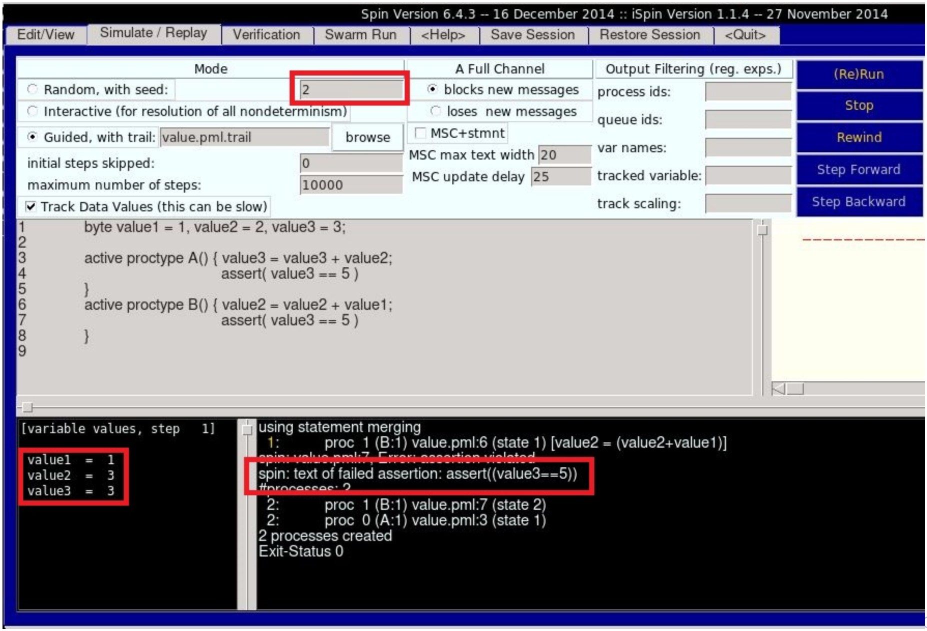
process ids:
queue ids:
var names:
tracked variable:
track scaling:

```
        byte value1 = 1, value2 = 2, value3 = 3;

        active proctype A() { value3 = value3 + value2;
                              assert( value3 == 5 )
        }
        active proctype B() { value2 = value2 + value1;
                              assert( value3 == 5 )
        }
```

```
[variable values, step   2]

value2  =  3
value3  =  5
```

```
0:      proc  - (:root:) creates proc  0 (A)
0:      proc  - (:root:) creates proc  1 (B)
1:      proc  0 (A:1) value.pml:3 (state 1) [value3 = (value3+value2)]
2:      proc  1 (B:1) value.pml:6 (state 1) [value2 = (value2+value1)]
3:      proc  0 (A:1) value.pml:4 (state 2) [assert((value3==5))]
4:      proc  1 (B:1) value.pml:7 (state 2) [assert((value3==5))]
4:      proc  1 (B:1) terminates
4:      proc  0 (A:1) terminates
2 processes created
```

Second Simulation Run



Mode
- ○ Random, with seed: 2
- ○ Interactive (for resolution of all nondeterminism)
- ● Guided, with trail: value.pml.trail

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:

```
byte value1 = 1, value2 = 2, value3 = 3;

active proctype A() { value3 = value3 + value2;
                      assert( value3 == 5 )
}
active proctype B() { value2 = value2 + value1;
                      assert( value3 == 5 )
}

```

```
[variable values, step    1]

value1  =  1
value2  =  3
value3  =  3
```

```
using statement merging
  1:        proc  1 (B:1) value.pml:6 (state 1) [value2 = (value2+value1)]
spin: value.pml:7, Error: assertion violated
spin: text of failed assertion: assert((value3==5))
#processes: 2
  2:        proc  1 (B:1) value.pml:7 (state 2)
  2:        proc  0 (A:1) value.pml:3 (state 1)
2 processes created
Exit-Status 0
```

Simulation vs Verification

Initialization:

```
value1 = 1, value2 = 2, value3 = 3;
```

Simulation 1:

```
A: value3 = value3 + value2;
B: value2 = value2 + value1;
A: assert( value3 == 5 );
B: assert( value3 == 5 );
```

5 == 5

Simulation 2:

```
B: value2 = value2 + value1;
B: assert( value3 == 5 );
```

3 == 5

- A simulation run will only explore one execution trace.
- A verification run will explore all execution traces.

Next: performing verification within **iSpin** ...

Setting Verification Parameters

Safety: safety - invalid endstates (deadlocks); assertion violations.

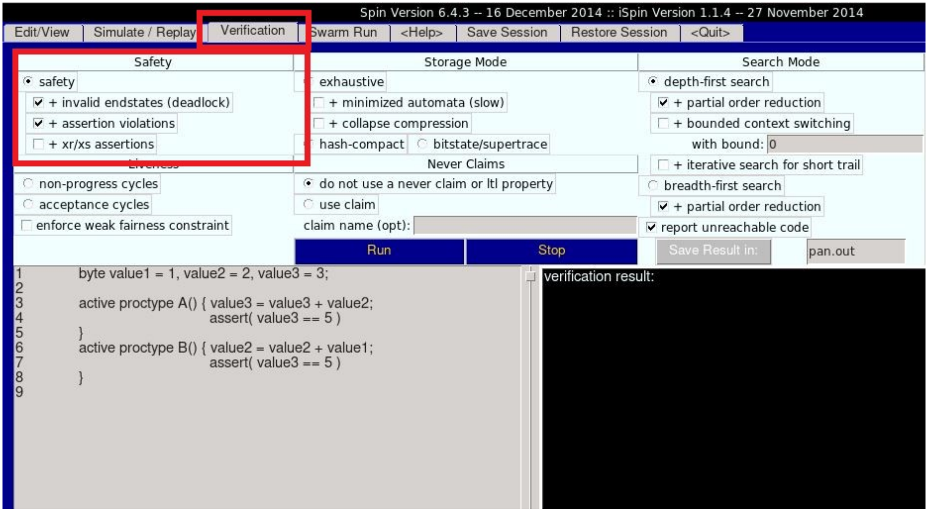
Liveness: non-progress cycles; acceptance cycles; enforce weak fairness.

Never Claims: relates to LTL reasoning (more details later).

Storage Mode: exhaustive; hash-compact; bitstate/supertrace.

Search Mode: depth-first - partial order reduction; iterative search (shortest trail); breadth first; report unreachable code.

Running a Verification (Assertion Correctness)



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

Safety
- [✓ safety]
  - [✓] + invalid endstates (deadlock)
  - [✓] + assertion violations
  - [ ] + xr/xs assertions

Liveness
- ( ) non-progress cycles
- ( ) acceptance cycles
- [ ] enforce weak fairness constraint

Storage Mode
- ( ) exhaustive
  - [ ] + minimized automata (slow)
  - [ ] + collapse compression
- ( ) hash-compact
- ( ) bitstate/supertrace

Never Claims
- [✓ do not use a never claim or ltl property]
- ( ) use claim
- claim name (opt):

Search Mode
- [✓ depth-first search]
  - [✓] + partial order reduction
  - [ ] + bounded context switching
  - with bound: 0
  - [ ] + iterative search for short trail
- ( ) breadth-first search
  - [✓] + partial order reduction
- [✓] report unreachable code

pan.out

```
        byte value1 = 1, value2 = 2, value3 = 3;

        active proctype A() { value3 = value3 + value2;
                              assert( value3 == 5 )
        }
        active proctype B() { value2 = value2 + value1;
                              assert( value3 == 5 )
        }
```

verification result:

Running a Verification (Assertion Correctness)

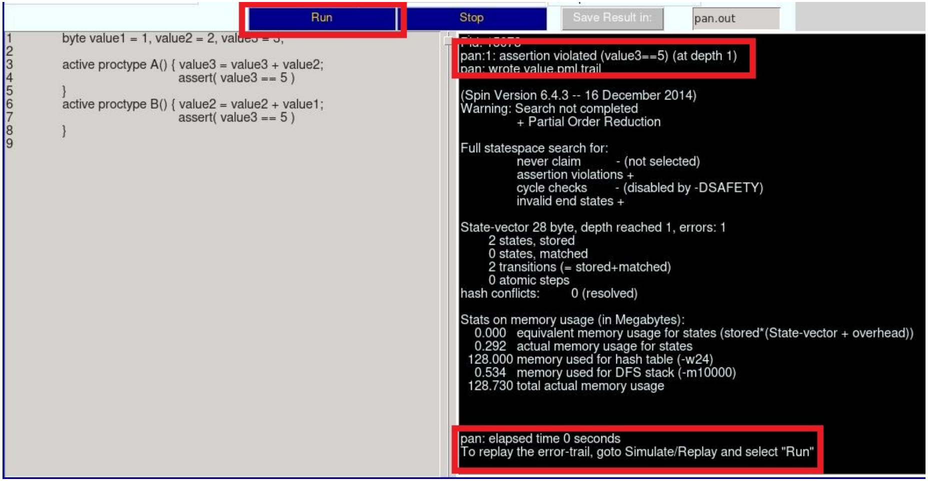
1. **Set verification parameters:** within the `"Safety"` panel select the `"assertion violations"` second on the list (see slide 10).
2. **Select "Run" button:** Output will be generated within the `"verification result"` panel (see slide 12). Either
   - a successful verification (no violations) or
   - an unsuccessful verification (violations) will be reported.

   In the case of an unsuccessful verification you will be invited to

   'replay the error-trail, goto Simulate/Replay and select "Run"'.

   The **Guided (simulation), with trail** mode runs the failure trace (counter-example) generated by the verification run.

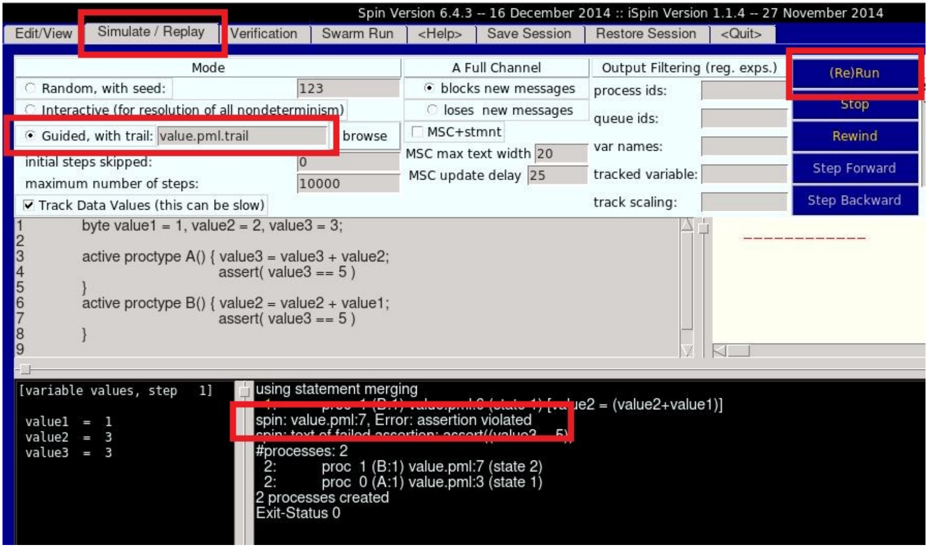
Running a Verification (Assertion Correctness)



```
byte value1 = 1, value2 = 2, value3 = 3;

active proctype A() { value3 = value3 + value2;
                      assert( value3 == 5 )
}
active proctype B() { value2 = value2 + value1;
                      assert( value3 == 5 )
}
```

```
Pid: 15075
pan:1: assertion violated (value3==5) (at depth 1)
pan: wrote value.pml.trail

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

Full statespace search for:
        never claim         - (not selected)
        assertion violations +
        cycle checks        - (disabled by -DSAFETY)
        invalid end states +

State-vector 28 byte, depth reached 1, errors: 1
    2 states, stored
    0 states, matched
    2 transitions (= stored+matched)
    0 atomic steps
hash conflicts:        0 (resolved)

Stats on memory usage (in Megabytes):
  0.000   equivalent memory usage for states (stored*(State-vector + overhead))
  0.292   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"
```

Running a Guided Simulation



Mode
- Random, with seed: 123
- Interactive (for resolution of all nondeterminism)
- [✓ Guided, with trail: value.pml.trail]

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

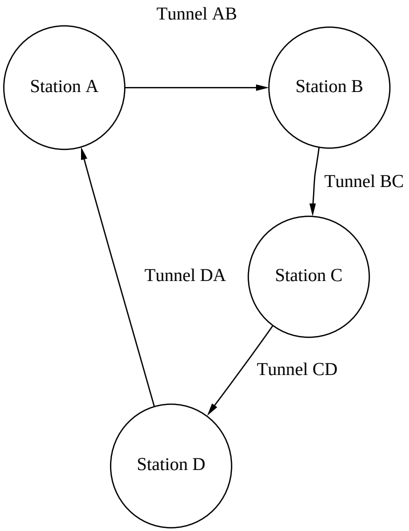
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:

```
byte value1 = 1, value2 = 2, value3 = 3;

active proctype A() { value3 = value3 + value2;
                      assert( value3 == 5 )
}
active proctype B() { value2 = value2 + value1;
                      assert( value3 == 5 )
}
```

```
[variable values, step   1]

 value1  =  1
 value2  =  3
 value3  =  3
```

```
using statement merging
  1:    proc  1 (B:1) value.pml:6 (state 1) [value2 = (value2+value1)]
spin: value.pml:7, Error: assertion violated
spin: text of failed assertion: assert((value3==5))
#processes: 2
  2:    proc  1 (B:1) value.pml:7 (state 2)
  2:    proc  0 (A:1) value.pml:3 (state 1)
2 processes created
Exit-Status 0
```

TrainWare: An Unsafe Railway Network



A Promela Model of TrainWare

```
chan TunnelAB = [2] of { byte };
chan TunnelBC = [2] of { byte };
chan TunnelCD = [2] of { byte };
chan TunnelDA = [2] of { byte };

proctype Station(chan in_track, out_track) {
        byte train;
        do :: in_track?train; out_track!train od }

proctype Setup(chan track; byte train) { track!train }

init {  atomic{ run Setup(TunnelBC, 1);
                run Setup(TunnelDA, 2);
                run Station(TunnelDA, TunnelAB);
                run Station(TunnelAB, TunnelBC);
                run Station(TunnelBC, TunnelCD);
                run Station(TunnelCD, TunnelDA)} }
```

A Global Safety Assertion for TrainWare

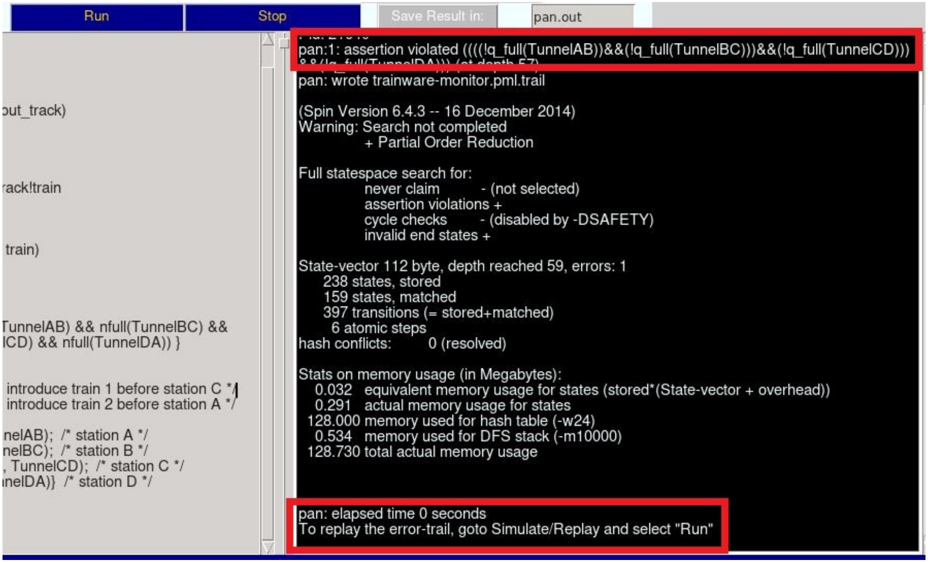
- Safety Assertion:

  "A tunnel can only be occupied by one train at a time."

```
chan TunnelAB = [2] of { byte };
chan TunnelBC = [2] of { byte };
chan TunnelCD = [2] of { byte };
chan TunnelDA = [2] of { byte };
...
proctype Monitor() { assert(nfull(TunnelAB) &&
                            nfull(TunnelBC) &&
                            nfull(TunnelCD) &&
                            nfull(TunnelDA))}

init {  atomic{ run Monitor(); ... }}
```

- See next 2 frames for **Spin**'s verification failure ...

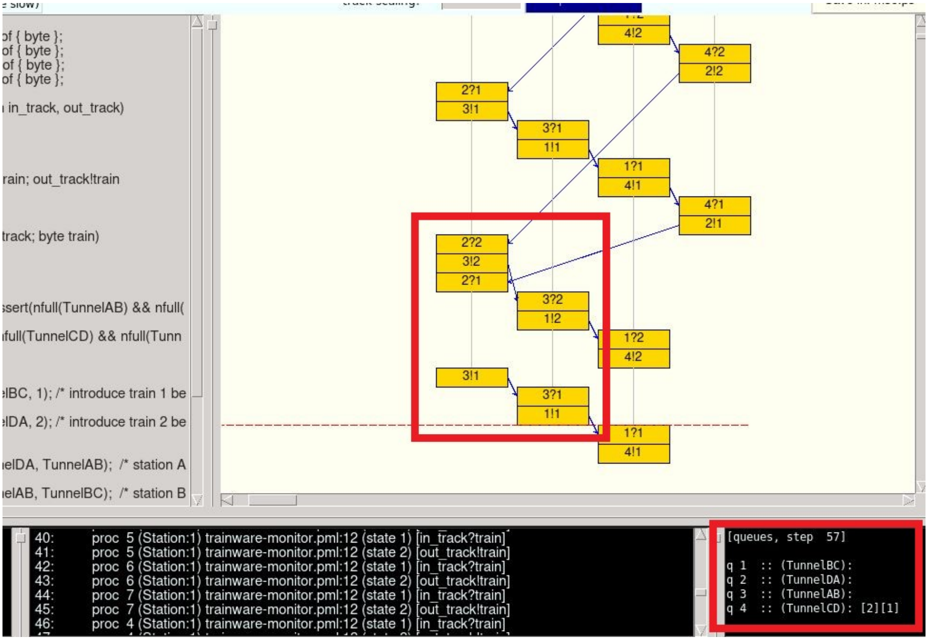
TrainWare: Safety Assertion Violation



```
out_track)

rack!train

 train)

TunnelAB) && nfull(TunnelBC) &&
ICD) && nfull(TunnelDA)) }

 introduce train 1 before station C */
 introduce train 2 before station A */

nelAB);  /* station A */
nelBC);  /* station B */
, TunnelCD);  /* station C */
nelDA)}  /* station D */
```

```
[illegible]
pan:1: assertion violated ((((!q_full(TunnelAB))&&(!q_full(TunnelBC)))&&(!q_full(TunnelCD)))
&&(!q_full(TunnelDA))) (at depth 57)
pan: wrote trainware-monitor.pml.trail

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

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 112 byte, depth reached 59, errors: 1
      238 states, stored
      159 states, matched
      397 transitions (= stored+matched)
        6 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.032   equivalent memory usage for states (stored*(State-vector + overhead))
    0.291   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"
```

TrainWare: Guided Simulation



```
of { byte };
of { byte };
of { byte };
of { byte };

in_track, out_track)

rain; out_track!train

track; byte train)

ssert(nfull(TunnelAB) && nfull(
nfull(TunnelCD) && nfull(Tunn

elBC, 1); /* introduce train 1 be
elDA, 2); /* introduce train 2 be

elDA, TunnelAB);  /* station A
elAB, TunnelBC);  /* station B
```

```
40:    proc  5 (Station:1) trainware-monitor.pml:12 (state 1) [in_track?train]
41:    proc  5 (Station:1) trainware-monitor.pml:12 (state 2) [out_track!train]
42:    proc  6 (Station:1) trainware-monitor.pml:12 (state 1) [in_track?train]
43:    proc  6 (Station:1) trainware-monitor.pml:12 (state 2) [out_track!train]
44:    proc  7 (Station:1) trainware-monitor.pml:12 (state 1) [in_track?train]
45:    proc  7 (Station:1) trainware-monitor.pml:12 (state 2) [out_track!train]
46:    proc  4 (Station:1) trainware-monitor.pml:12 (state 1) [in_track?train]
```

```
[queues, step  57]

q 1  :: (TunnelBC):
q 2  :: (TunnelDA):
q 3  :: (TunnelAB):
q 4  :: (TunnelCD): [2][1]
```

Validation Labels: End State Labels

- When modelling non-terminating systems how can we judge whether a process is in a **deadlock** state or an acceptable **waiting** state?
- End-state labels provide a solution, they allow the designer to explicitly indicate valid end-states.
- Definition of a valid end-state:
  - Every instantiated process has either terminated or is blocked at a statement that is labelled as an end-state.
  - All message channels are empty.
- An end-state label is any label with the prefix `"end"`, *e.g.* `end, end_1,...`

Semaphores Revisited

- Consider again Dijkstra's semaphore solution to the problem of ensuring mutual exclusion (see *Promela Part 2*), in particular the definition of the `semaphore` process:

  ```
  proctype semaphore(){do ::sema!p -> sema?v od}
  ```

  with the `"Invalid endstates (deadlock)"` verification option selected, this definition will result in a failed verification.
- The failure is because the `semaphore` process will block until another `user` process starts-up. Such desirable blocking is indicated to the verifier via an end-state label, *e.g.*

  ```
  proctype semaphore(){end: do ::sema!p -> sema?v od}
  ```

  this revised definition will lead to a successful verification.

Validation Labels: Progress-State Labels

- Progress-state labels allow a designer to explicitly define a notion of progress and thus verify that certain events do actually occur (a progress-state label has a "progress" prefix).
- A verification fails if there exists an infinite execution cycle that does not pass a progress-state label (**non-progress cycle**), *e.g.* assuming that count is initially 0 then the following fails:

```
do
:: (count >= 0) -> count = count+1;
:: (count == 9) -> progress: count = 0;
od
```

  On the other hand, the following will succeed:

```
do
:: (count < 9)  -> count = count+1;
:: (count == 9) -> progress: count = 0;
od
```

Validation Labels: Acceptance-State Labels

- Accept-state labels allow a designer to be able to verify that certain events do not happen infinitely often (an accept-state label has an `"accept"` prefix).
- A verification will fail if there exists an execution that visits an accept-state label infinitely often (**acceptance cycle**), *e.g.* assuming that `count` is initially 0 then the following fails:

```
        do
        :: (count >= 0) -> count = count+1;
        :: (count == 9) -> accept: count = 0;
        od
```

  On the other hand, the following will succeed:

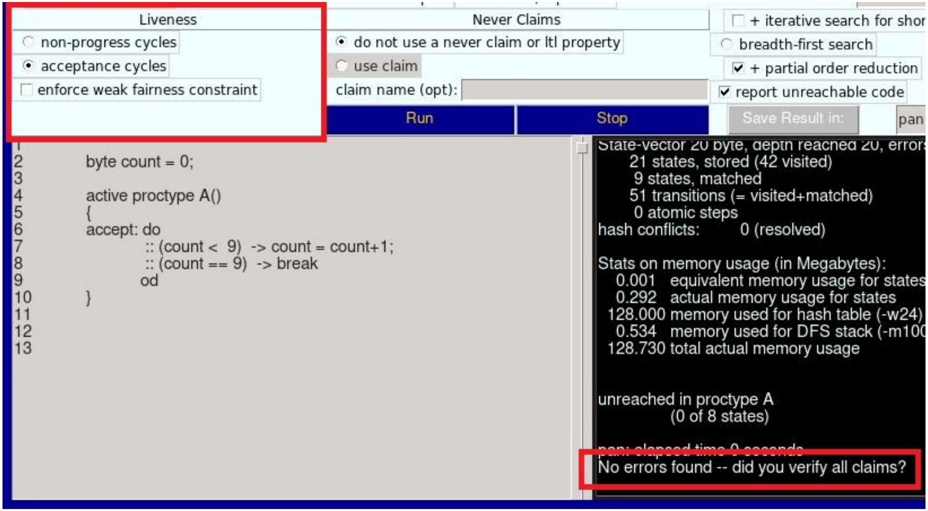
```
accept: do
        :: (count < 9)  -> count = count+1;
        :: (count == 9) -> break;
        od
```

Summary of Validation Labels

- Checking for:
  - invalid end-states is a safety property.
  - non-progress cycles is a liveness property.
  - acceptance cycles is a liveness property.

  Note: **acceptance** is the converse of **non-progress**, *i.e.* an **acceptance cycle** denotes a computation that visits an **accept label** infinitely often whereas a **non-progress cycle** denotes a computation that does not visit a **progress label** infinitely often.
- Using validation labels requires that the appropriate options are enabled via the `"Liveness"` panel of the verification tab (see next slide).

Liveness Panel in iSpin



Liveness
- ( ) non-progress cycles
- [✓ acceptance cycles]
- ☐ enforce weak fairness constraint

Never Claims
- [✓ do not use a never claim or ltl property]
- ( ) use claim
- claim name (opt):

- ☐ + iterative search for shor
- ( ) breadth-first search
- ☑ + partial order reduction
- ☑ report unreachable code

```
byte count = 0;

active proctype A()
{
accept: do
        :: (count <  9)  -> count = count+1;
        :: (count == 9)  -> break
        od
}
```

```
State-vector 20 byte, depth reached 20, errors
       21 states, stored (42 visited)
        9 states, matched
       51 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.001   equivalent memory usage for states
    0.292   actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m100
  128.730 total actual memory usage


unreached in proctype A
        (0 of 8 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Summary

Learning outcomes:

- To understand the difference between simulation and verification.
- To be able to use **iSpin** to verify assertions, both local (process) and global (system).
- To be able to use **iSpin** to detect deadlocks (invalid end-states) and livelocks (non-progress & acceptance cycles).

Recommended reading:

- **Spin** homepage: `http://spinroot.com`
- **Spin** on-line material: `http://spinroot.com/spin/Man/`