Distributed Systems Programming F29NM1

SPIN: Formal Analysis I

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- Introduce SPIN's formal analysis capabilities via XSPIN.
- Focus upon XSPIN'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

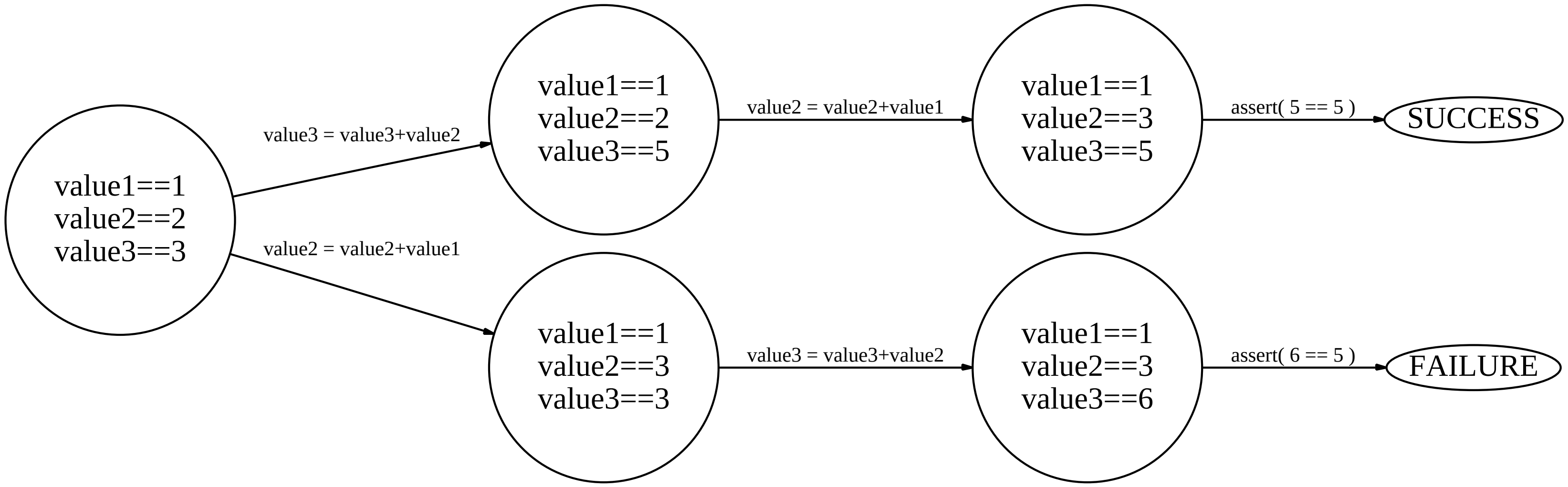
- Assertion verification:
  - local process assertions
  - global system assertions
- Validation labels:
  - success without termination
  - productive progress
- Temporal verification:
  - Linear Temporal Logic (LTL)
    (also referred to as linear-time temporal logic)

Local Assertions Revisited

```
byte value1 = 1, value2 = 2, value3 = 3;
proctype A() { value3 = value3 + value2;
               assert( value3 == 5 )
}
proctype B() { value2 = value2 + value1;
               assert( value3 == 5 )
}
init { atomic{ run A(); run B()}}
```

Using SPIN's simulator, will assertion checking succeed or fail?

Simulation vs Verification



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

Next: doing verification within XSPIN ...

Setting Verification Parameters

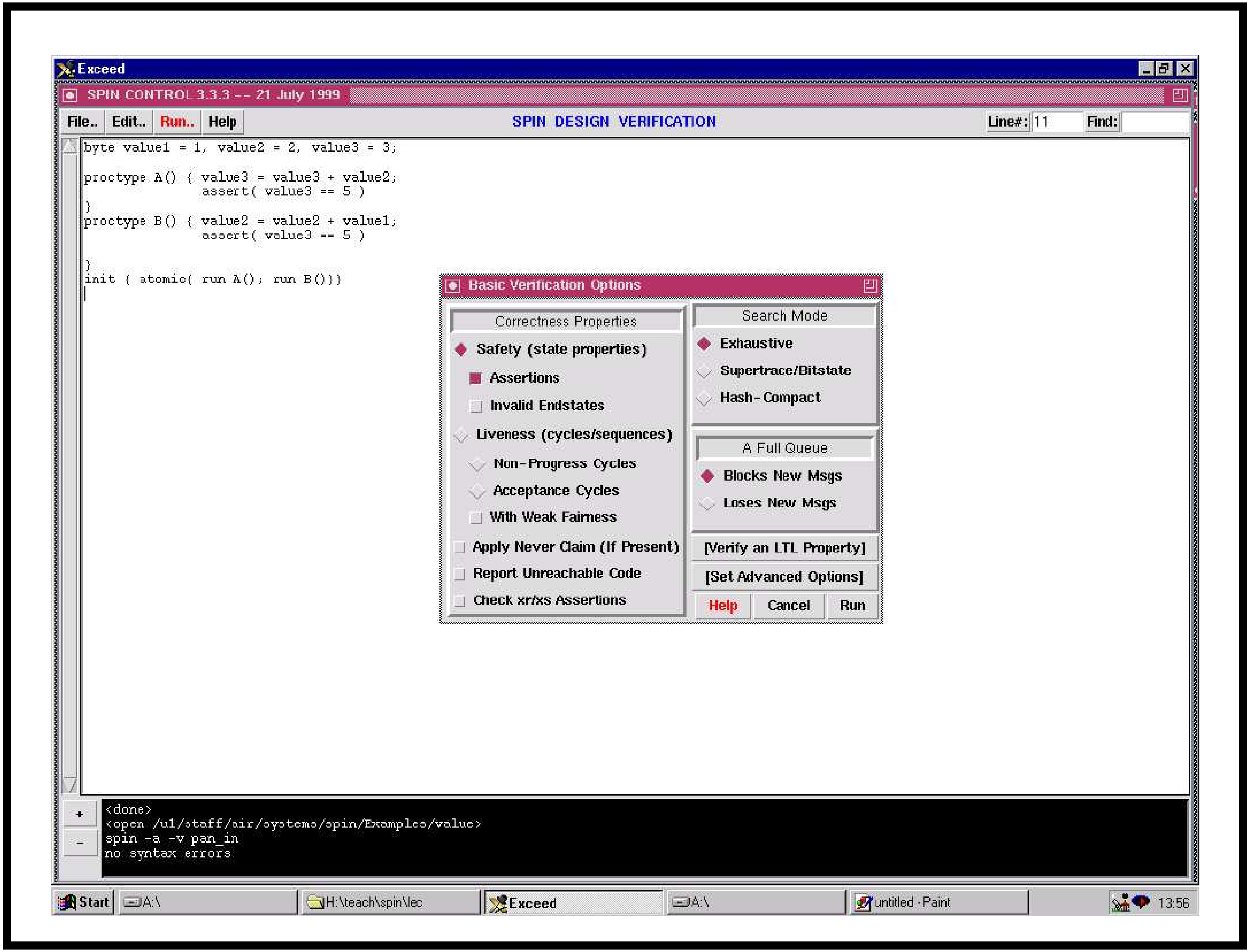
Correctness properties: safety & liveness, ...

Search mode: exhaustive, supertrace/bitstate, hash-compact.

A full queue: blocks or loses new messages when queue is full.

Verify an LTL property: management of LTL formulae (also available via `"LTL Property manager.."` option of the `"Run.."` menu.

Set advance options: includes search options such as `"Find Shortest Trail (costly!)"` (counter-example).

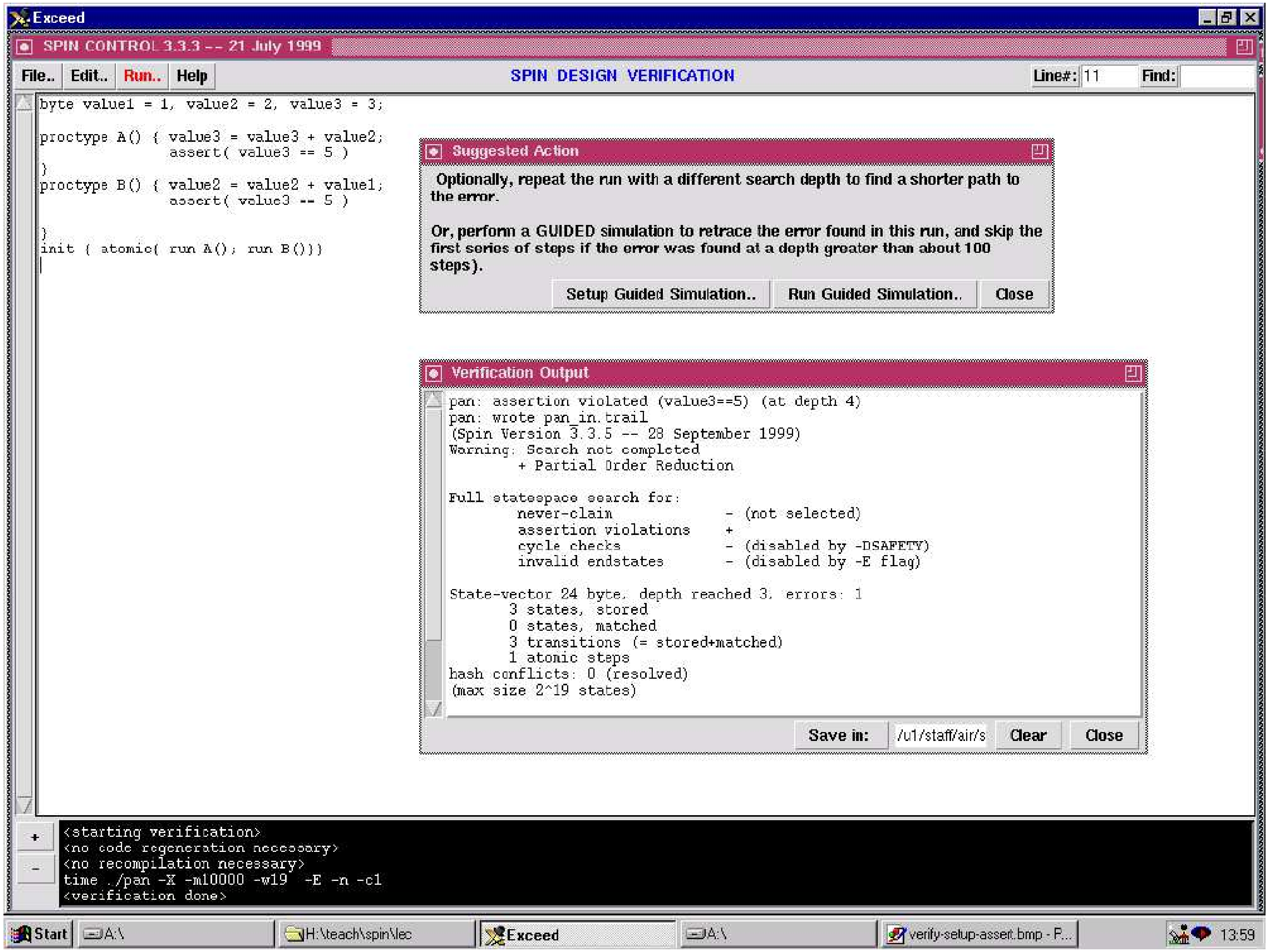

Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 11 Find:
byte value1 = 1, value2 = 2, value3 = 3;
proctype A() { value3 = value3 + value2;
assert(value3 == 5)
}
proctype B() { value2 = value2 + value1;
assert(value3 == 5)
}
init { atomic(run A(); run B()) }
Basic Verification Options
Correctness Properties
Safety (state properties)
Assertions
Invalid Endstates
Liveness (cycles/sequences)
Non-Progress Cycles
Acceptance Cycles
With Weak Fairness
Apply Never Claim (If Present)
Report Unreachable Code
Check xr/xs Assertions
Search Mode
Exhaustive
Supertrace/Bitstate
Hash-Compact
A Full Queue
Blocks New Msgs
Loses New Msgs
[Verify an LTL Property]
[Set Advanced Options]
Help
Cancel
Run
<done>
<open /u1/staff/air/systems/spin/Examples/value>
spin -a -v pan_in
no syntax errors
Start
A:\
H:\teach\spin\lec
Exceed
A:\
untitled - Paint
13:56

Running a Verification (Assertion Correctness)

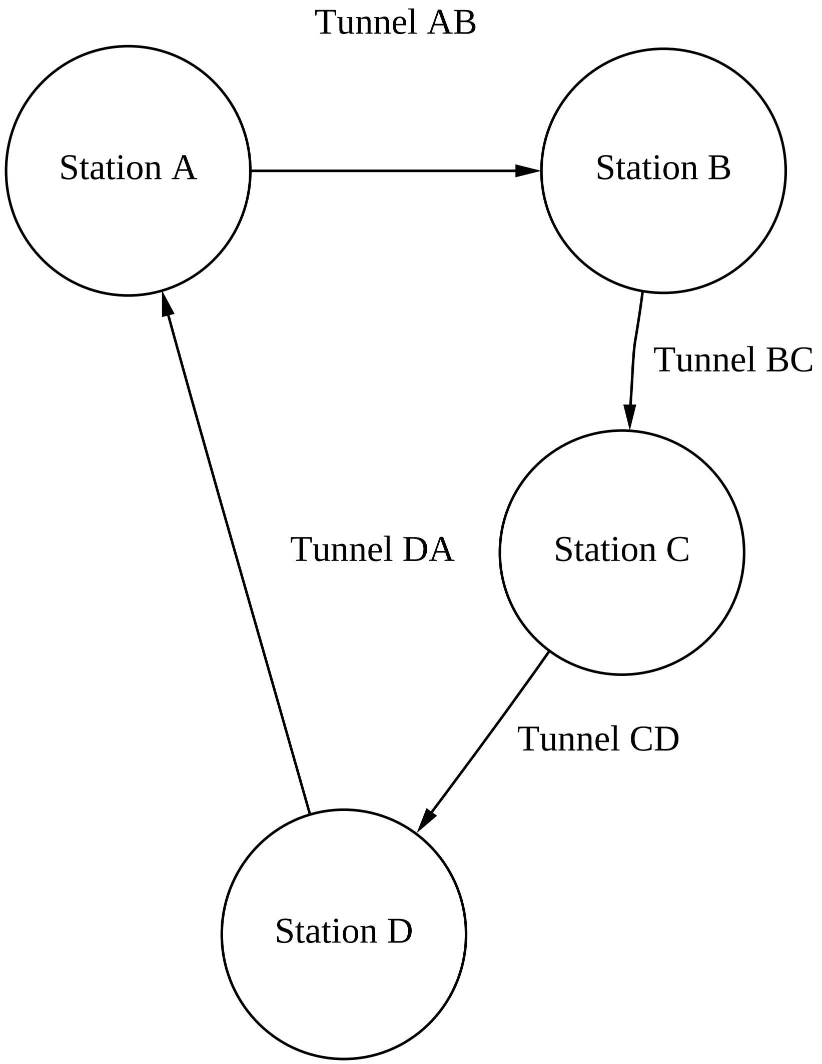
1. **Set verification parameters:** within the `"Correctness Properties"` panel select the `"Assertions"` option which appears under `"Safety (state properties)"` (see slide 8).

2. **(Re)run verification:** a `"Verification Output"` window will be generated. Either a successful verification (no violations) or an unsuccessful verification (violations) will be reported. In the case of an unsuccessful verification a `"Suggested Action"` window will be generated (see slide 10):

   - **Setup Guided Simulation**: enables you to check and/or modify the simulation parameters.
   - **Run Guided Simulation**: runs the failure trace (counter-example) generated by the verification run.


Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 11 Find:
byte value1 = 1, value2 = 2, value3 = 3;
proctype A() { value3 = value3 + value2;
assert(value3 == 5)
}
proctype B() { value2 = value2 + value1;
assert(value3 == 5)
}
init { atomic(run A(); run B()}}
Suggested Action
Optionally, repeat the run with a different search depth to find a shorter path to the error.
Or, perform a GUIDED simulation to retrace the error found in this run, and skip the first series of steps if the error was found at a depth greater than about 100 steps).
Setup Guided Simulation..
Run Guided Simulation..
Close
Verification Output
pan: assertion violated (value3==5) (at depth 4)
pan: wrote pan_in.trail
(Spin Version 3.3.5 -- 28 September 1999)
Warning: Search not completed
+ Partial Order Reduction
Full statespace search for:
never-claim - (not selected)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid endstates - (disabled by -E flag)
State-vector 24 byte, depth reached 3, errors: 1
3 states, stored
0 states, matched
3 transitions (= stored+matched)
1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
Save in: /u1/staff/air/s Clear Close
<starting verification>
<no code regeneration necessary>
<no recompilation necessary>
time ./pan -X -m10000 -w19 -E -n -c1
<verification done>
Start A:\ H:\teach\spin\lec Exceed A:\ verify-setup-assert.bmp - P... 13:59

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

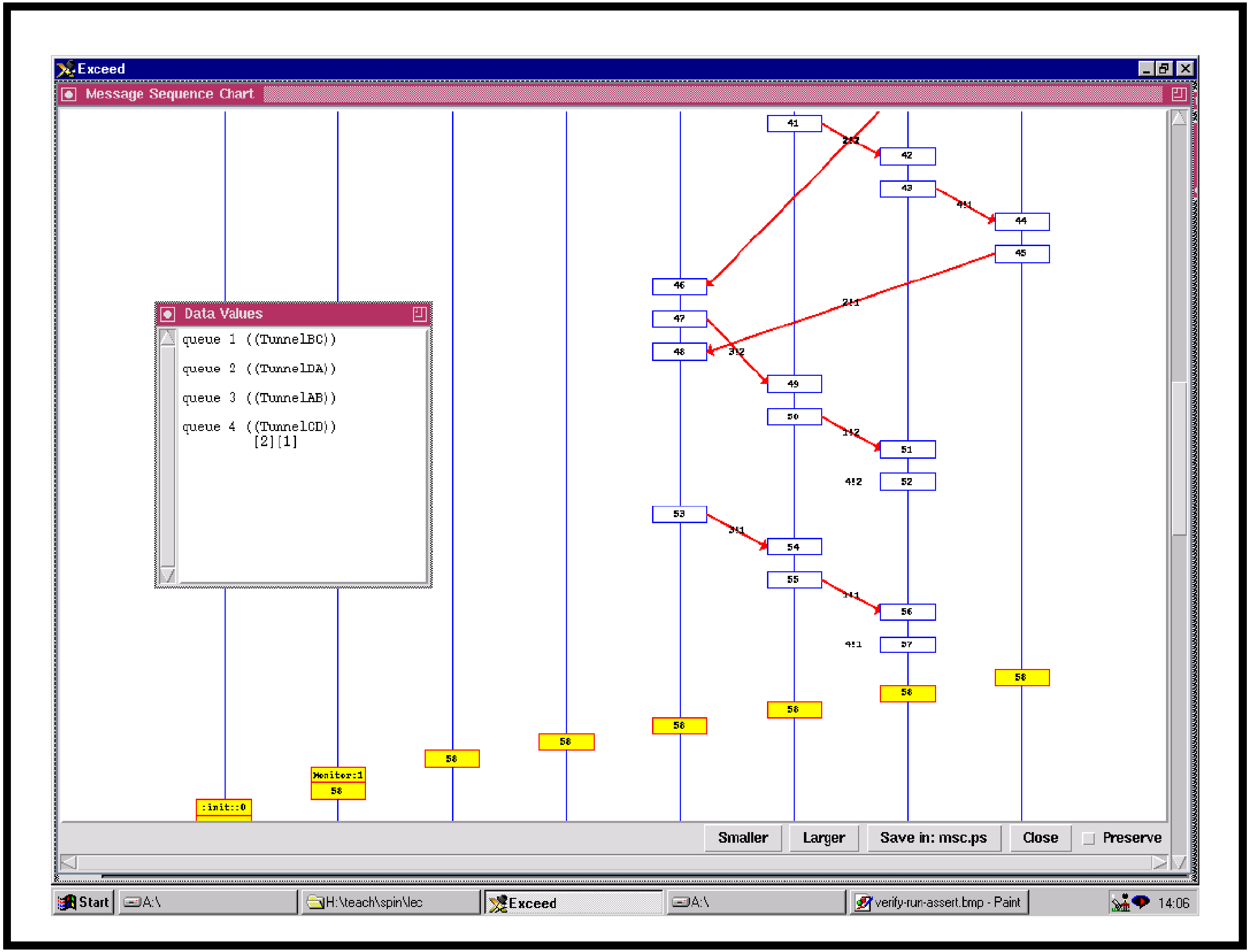
- Safety Assertion:

  "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 slide for SPIN's counter example ...


Exceed
Message Sequence Chart
Data Values
queue 1 ((TunnelBC))
queue 2 ((TunnelDA))
queue 3 ((TunnelAB))
queue 4 ((TunnelCD))
[2][1]
Smaller
Larger
Save in: msc.ps
Close
Preserve
Start
A:\
H:\teach\spin\lec
Exceed
A:\
verify-run-assert.bmp - Paint
14:06

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 lecture 4), in particular the definition of the `semaphore` process:

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

  with the `"Invalid Endstates"` correctness 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:

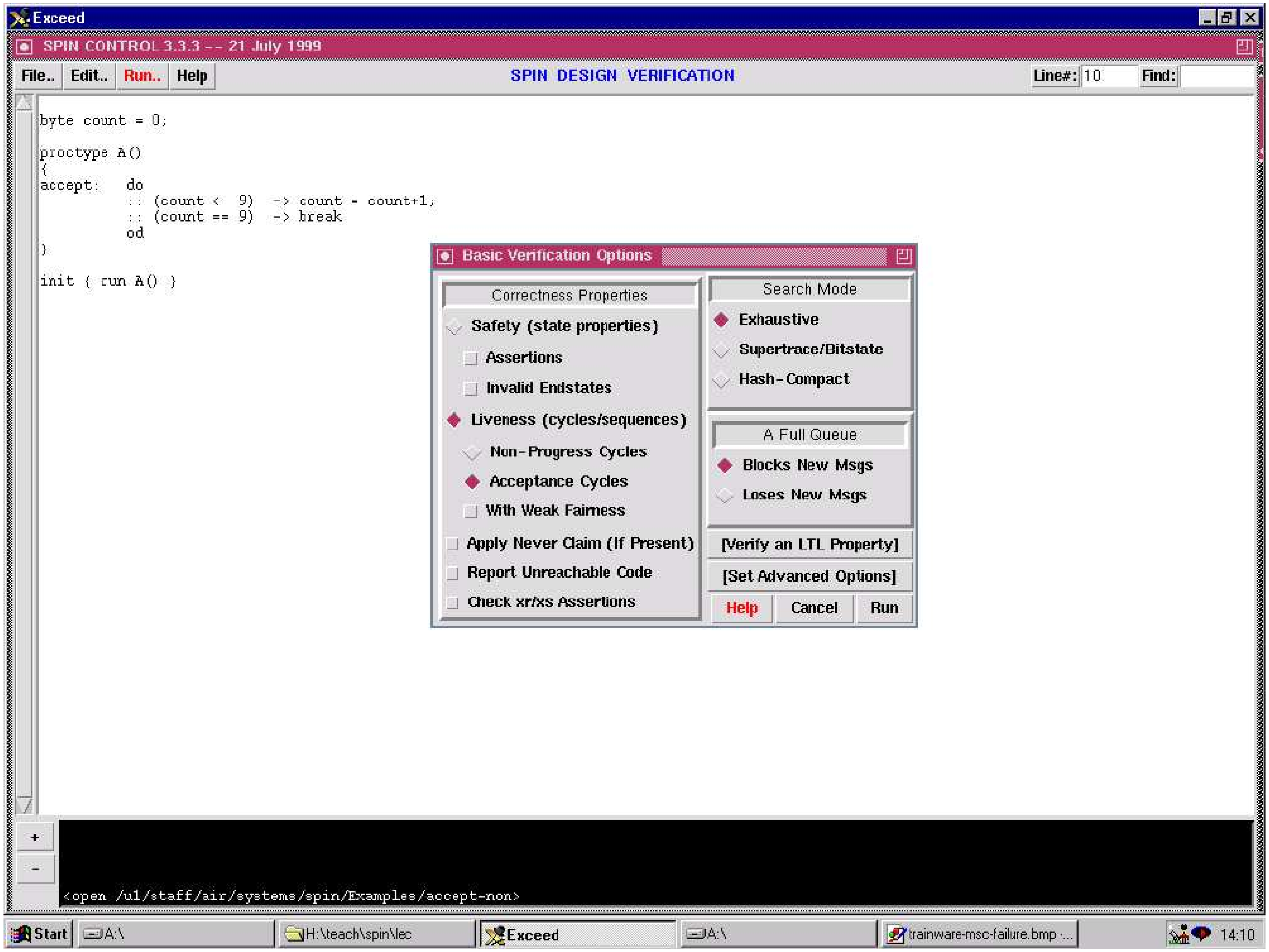
```
accept: do
          :: (count < 9)  -> count = count+1;
          :: (count == 9) -> break;
          od
```

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 progress, *i.e.* no acceptance cycles implies no non-progress cycles.

- Using validation labels requires that the appropriate options are enabled via the `"Correctness Properties"` panel (see next slide).


Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File..
Edit..
Run..
Help
SPIN DESIGN VERIFICATION
Line#: 10
Find:
byte count = 0;
proctype A()
{
accept: do
:: (count < 9) -> count = count+1;
:: (count == 9) -> break
od
}
init { run A() }
Basic Verification Options
Correctness Properties
Safety (state properties)
Assertions
Invalid Endstates
Liveness (cycles/sequences)
Non-Progress Cycles
Acceptance Cycles
With Weak Fairness
Apply Never Claim (If Present)
Report Unreachable Code
Check xr/xs Assertions
Search Mode
Exhaustive
Supertrace/Bitstate
Hash-Compact
A Full Queue
Blocks New Msgs
Loses New Msgs
[Verify an LTL Property]
[Set Advanced Options]
Help
Cancel
Run
<open /u1/staff/air/systems/spin/Examples/accept-non>
Start
A:\
H:\teach\spin\lec
Exceed
A:\
trainware-msc-failure.bmp -...
14:10

Summary

Learning outcomes:

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

Recommended reading:

- "Basic Spin Manual" — see course homepage