Distributed Systems Programming F29NM1

SPIN: Formal Analysis II

Andrew Ireland

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.
- Sketch the foundational issues that underpin 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 & progress cycles) and undesirable (accept 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 SPIN

- Step 1: Run the `"LTL Property Manager"`.
- Step 2: Enter the temporal property you wish to verify. Note that you must use lowercase names for propositional constants.
- Step 3: Indicate whether the temporal property should hold on:
  - all executions (**desired behaviour**) or
  - no executions (**error behaviour**)
- Step 4: Enter a macro-definition for each propositional constant within the `"Symbol Definitions"` sub-window.
- Step 5: Press the `"Run Verification"` button and then `"Run"` button within the `"LTL Verification"` window.

Note that LTL properties can be saved for future use (see the `"Save As"` and `"Load"` buttons).

TrainWare Revisited

- **Desired behaviour:**

```
[]!(full(TunnelAB) || full(TunnelBC) ||
    full(TunnelCD) || full(TunnelDA))
```

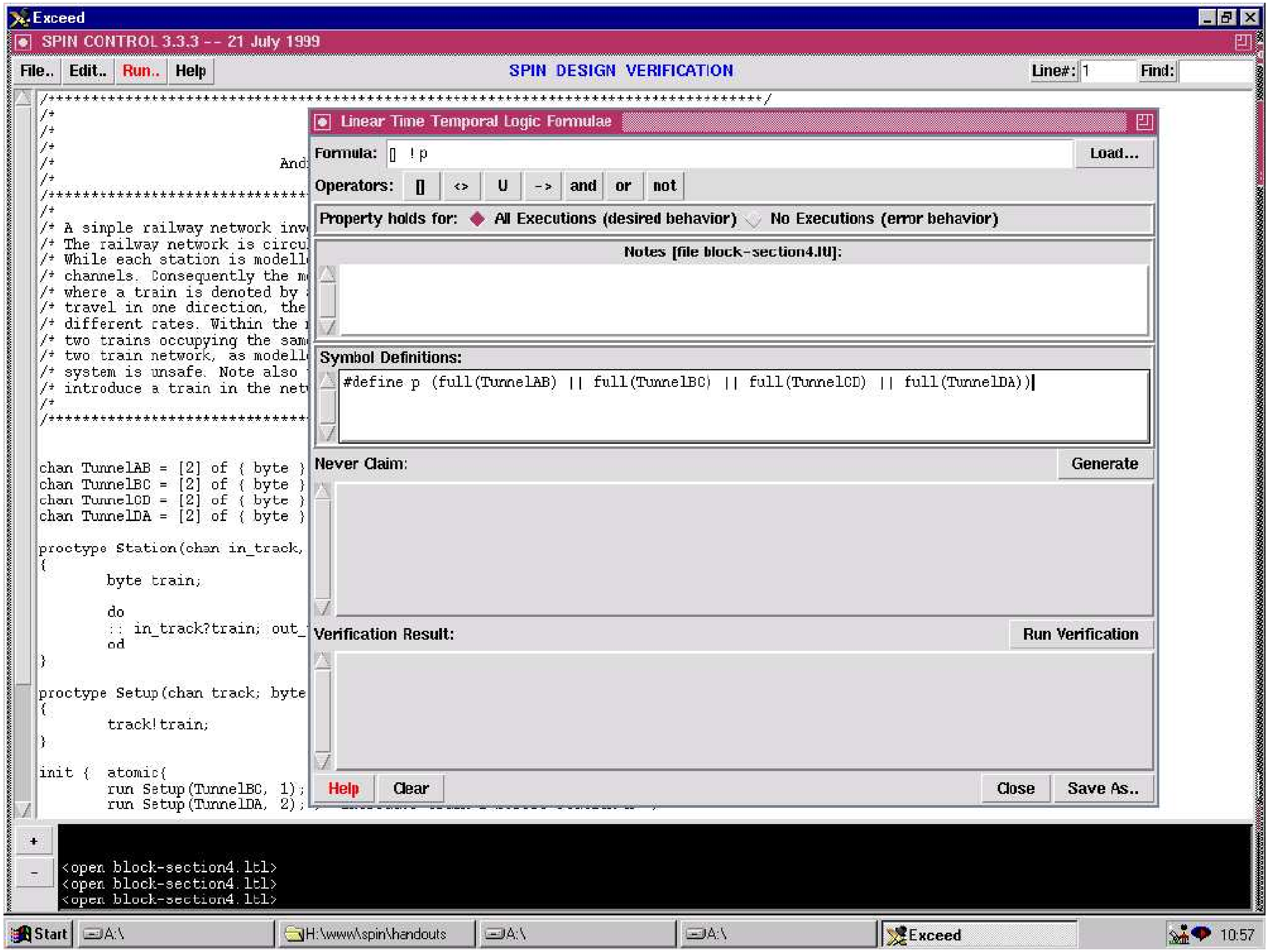
  This property should hold on **all executions**, *i.e.* always the case that none of the tunnels is occupied by more than one train.

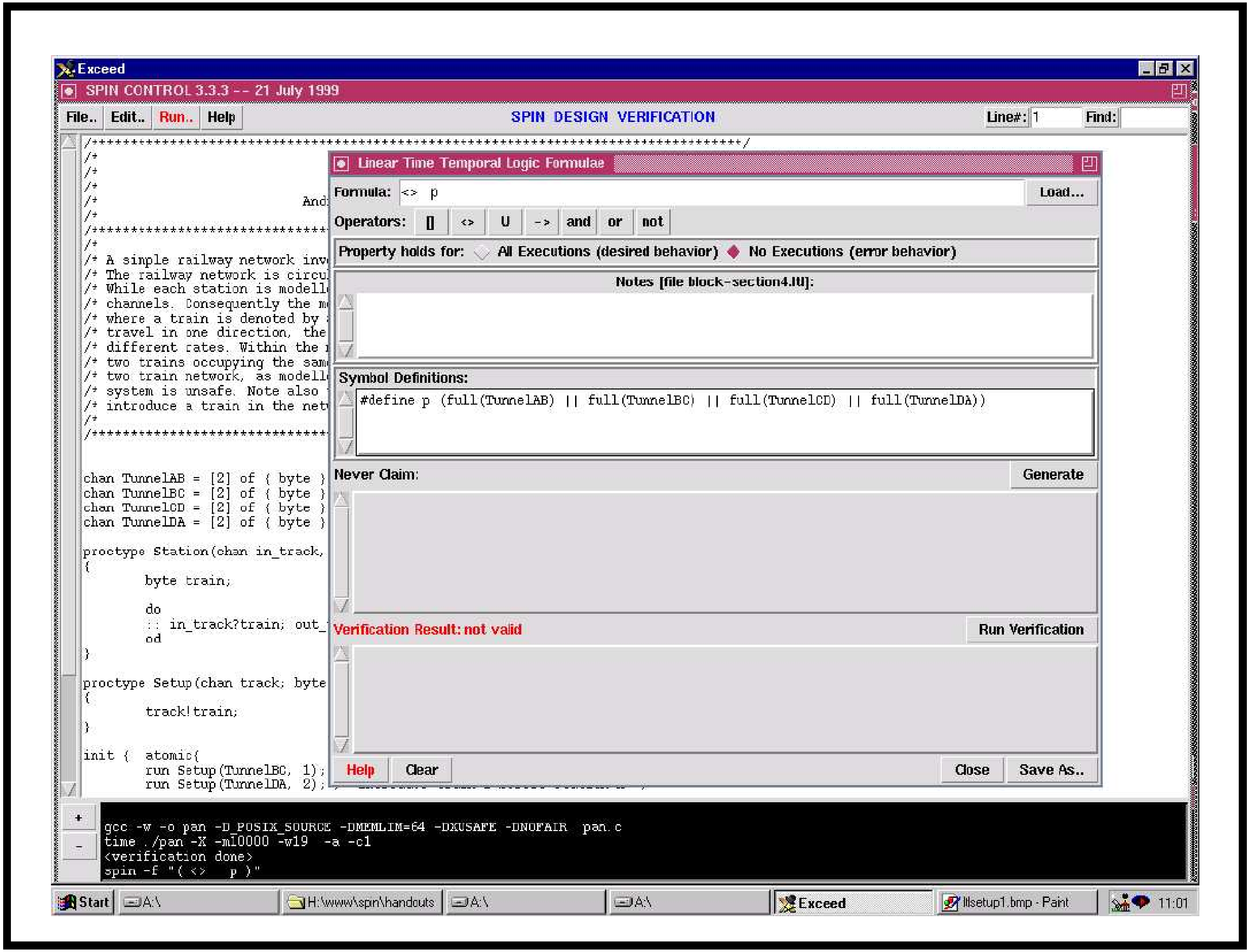
- **Error behaviour:**

```
<>(full(TunnelAB) || full(TunnelBC) ||
   full(TunnelCD) || full(TunnelDA))
```

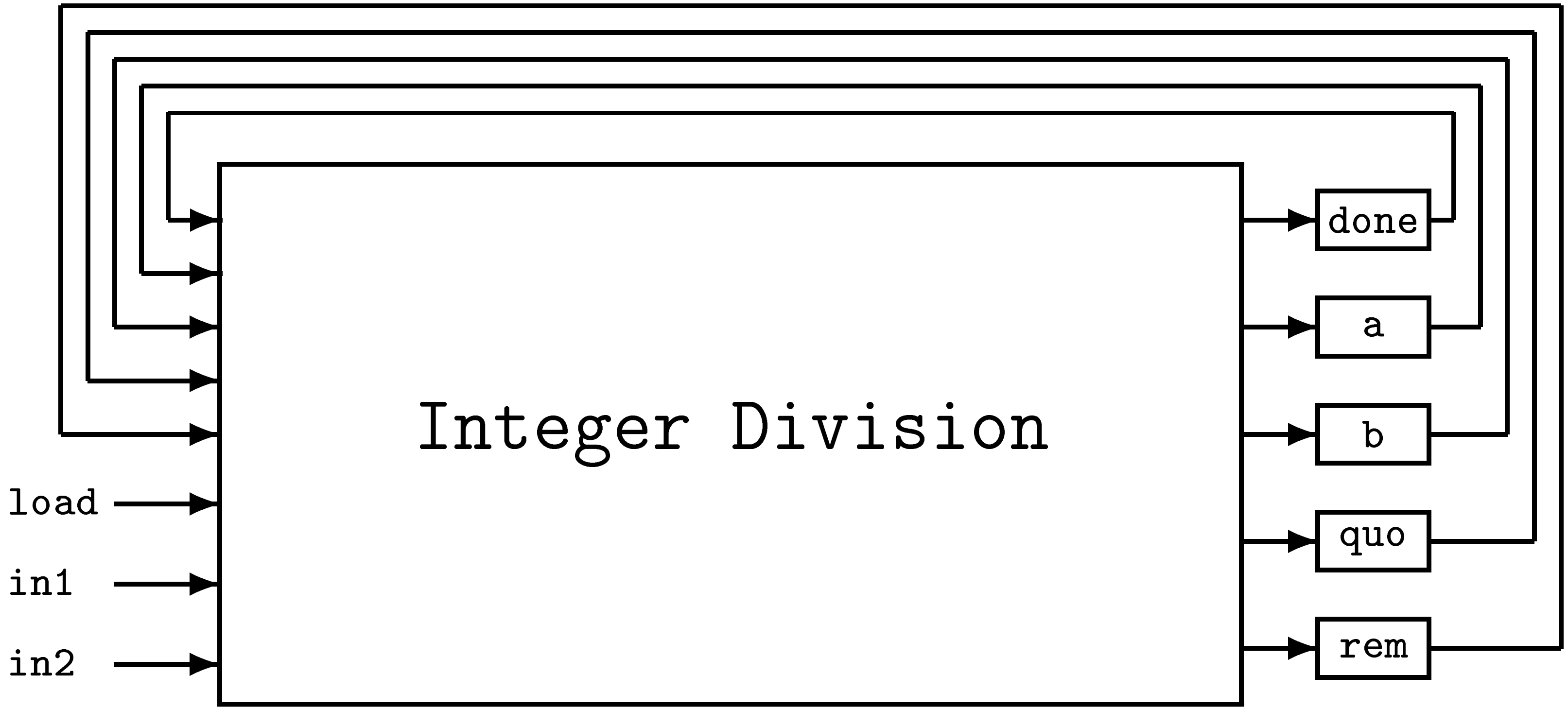
  This property should hold on **no executions**, *i.e.* if eventually a tunnel is occupied by more than one train then our design is flawed.

The setup windows for the examples given above are presented on the following two slides ...


Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 1 Find:
Linear Time Temporal Logic Formulae
Formula: [] !p
Load...
Operators: [] <> U -> and or not
Property holds for: All Executions (desired behavior) No Executions (error behavior)
Notes [file block-section4.ltl]:
Symbol Definitions:
#define p (full(TunnelAB) || full(TunnelBC) || full(TunnelCD) || full(TunnelDA))
Never Claim:
Generate
Verification Result:
Run Verification
Help Clear Close Save As..
/* A simple railway network inv
/* The railway network is circu
/* While each station is modell
/* channels. Consequently the m
/* where a train is denoted by
/* travel in one direction, the
/* different rates. Within the
/* two trains occupying the sam
/* two train network, as modell
/* system is unsafe. Note also
/* introduce a train in the net
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,
byte train;
do
:: in_track?train; out_
od
proctype Setup(chan track; byte
track!train;
init { atomic{
run Setup(TunnelBC, 1);
run Setup(TunnelDA, 2);
<open block-section4.ltl>
<open block-section4.ltl>
<open block-section4.ltl>
Start A:\ H:\www\spin\handouts A:\ A:\ Exceed 10:57

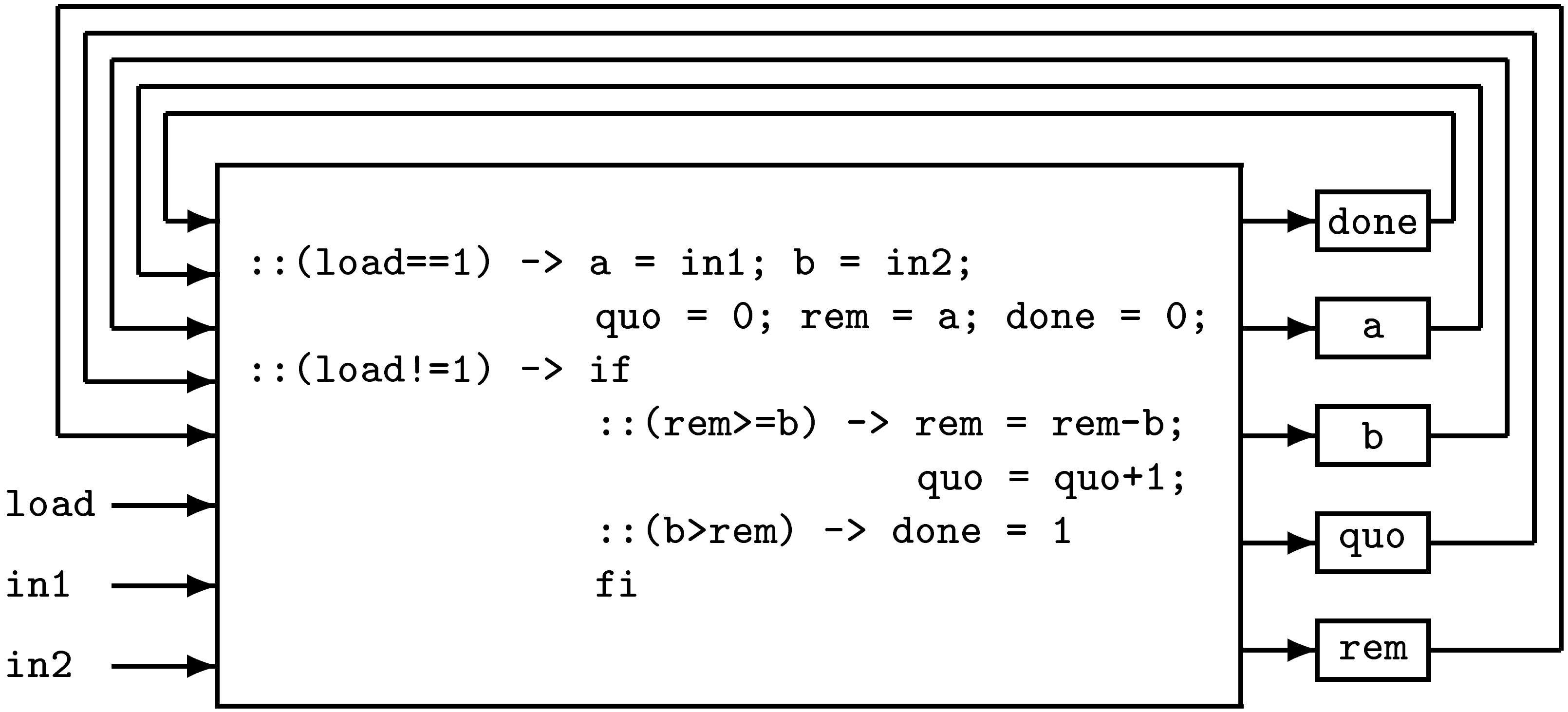

Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 1 Find:
Linear Time Temporal Logic Formulae
Formula: <> p
Load...
Operators: [] <> U -> and or not
Property holds for: All Executions (desired behavior) No Executions (error behavior)
Notes [file block-section4.ltl]:
Symbol Definitions:
#define p (full(TunnelAB) || full(TunnelBC) || full(TunnelCD) || full(TunnelDA))
Never Claim:
Generate
Verification Result: not valid
Run Verification
Help Clear Close Save As..
/* A simple railway network inv
/* The railway network is circu
/* While each station is modell
/* channels. Consequently the m
/* where a train is denoted by
/* travel in one direction, the
/* different rates. Within the
/* two trains occupying the sam
/* two train network, as modell
/* system is unsafe. Note also
/* introduce a train in the net
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,
byte train;
do
:: in_track?train; out_
od
proctype Setup(chan track; byte
track!train;
init { atomic{
run Setup(TunnelBC, 1);
run Setup(TunnelDA, 2);
gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=64 -DXUSAFE -DNOFAIR pan.c
time ./pan -X -m10000 -w19 -a -c1
<verification done>
spin -f "(<> p)"
Start A:\ H:\www\spin\handouts A:\ A:\ Exceed llsetup1.bmp - Paint 11:01

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

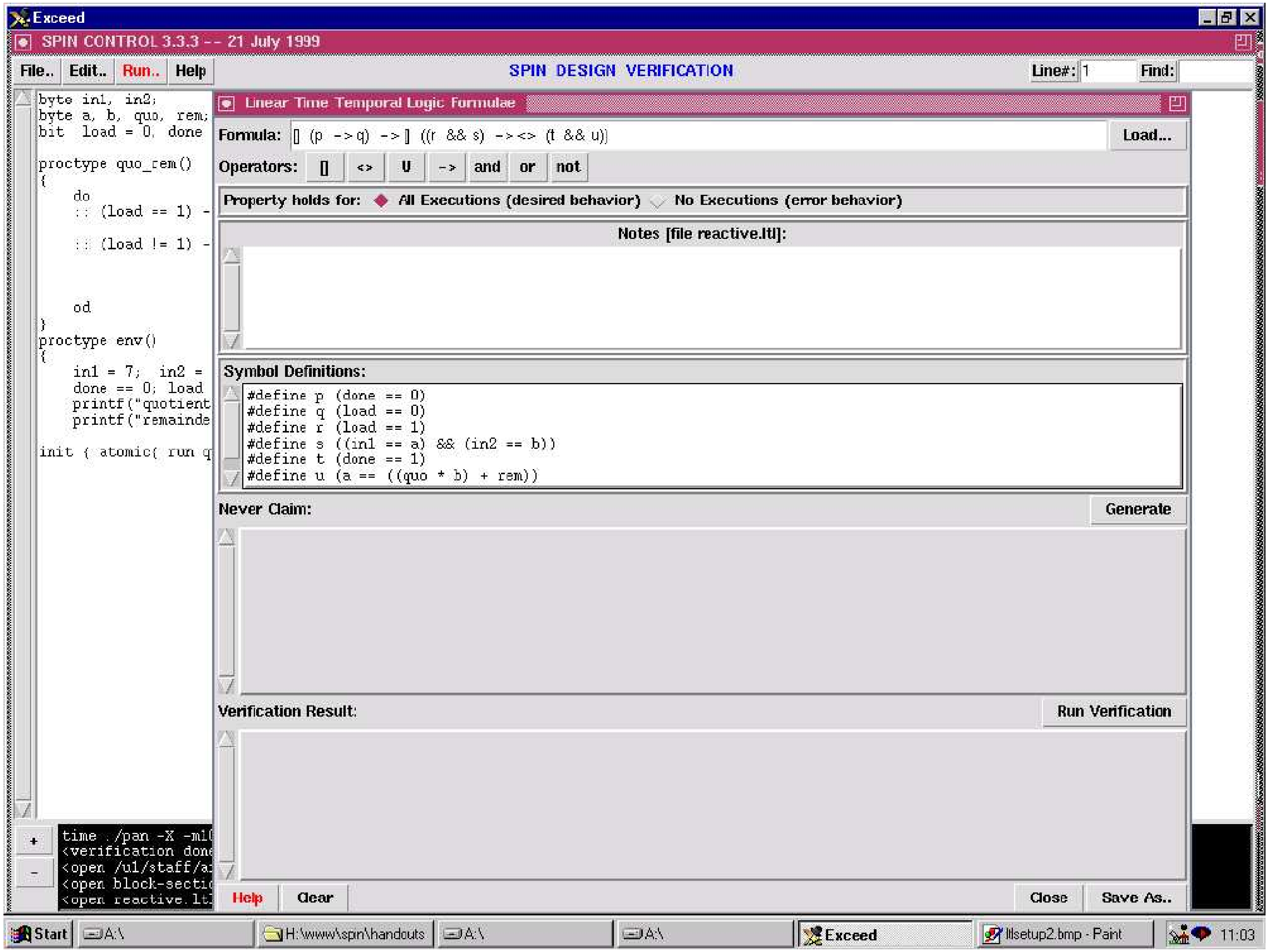
- Fairness is a special case of liveness and relates to the how the underlying process scheduler deals with contention, *i.e.* clients competing for the same computational resource.
- Notions of fairness:
  - Weak-fairness (just): a process that continuously makes a request will eventually be serviced.
  - 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 `"LTL Verification"` window (a sub-window of `"Linear Time Logic Formulae"` window).
- 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!


Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 1 Find:
byte in1, in2;
byte a, b, quo, rem;
bit load = 0, done
proctype quo_rem()
{
do
:: (load == 1) -
:: (load != 1) -
od
}
proctype env()
{
in1 = 7; in2 =
done == 0; load
printf("quotient
printf("remainde
init { atomic(run q
Linear Time Temporal Logic Formulae
Formula: [] (p -> q) -> [] ((r && s) -> <> (t && u))
Load...
Operators: [] <> U -> and or not
Property holds for: All Executions (desired behavior) No Executions (error behavior)
Notes [file reactive.ltl]:
Symbol Definitions:
#define p (done == 0)
#define q (load == 0)
#define r (load == 1)
#define s ((in1 == a) && (in2 == b))
#define t (done == 1)
#define u (a == ((quo * b) + rem))
Never Claim:
Generate
Verification Result:
Run Verification
Help Clear Close Save As..
time ./pan -X -m1
<verification done
<open /u1/staff/a
<open block-secti
<open reactive.ltl
Start A:\ H:\www\spin\handouts A:\ A:\ Exceed ltlsetup2.bmp - Paint 11:03

Temporal Reasoning: How Does It Work?

- A Promela model of a system is defined in terms of process templates.
- System level verification requires a representation that expresses all possible behaviours that the system can exhibit.
- To achieve this, SPIN translates each process template into a finite automaton.
- A single automaton that models the behaviour of the whole system can be calculated by taking the **asynchronous** interleaving product of all the processes.
- This product automaton defines the system model, *i.e.* the state space of the system, and is represented as a graph called the **reachability graph**.

Temporal Reasoning: How Does It Work?

- SPIN attempts to verify a **temporal property** by proving that its negation is not satisfied by the system model.
- SPIN translates the negated temporal property into a special Promela process called a **never claim**.
- A never claim defines behaviours that are undesirable or erroneous. Note that if verifying an error behaviour, then the original property is not negated.
- Verification corresponds to checking that no execution sequence within the system model matches (synchronizes) with the never claim, *i.e.* no acceptance cycles are detected.
- Matching is implemented at the level of the reachability graph and a graph representation of the never claim.

Summary

Learning outcomes:

- To be able to understand & write temporal properties expressed in LTL.
- To be able to use XSPIN to verify temporal properties of system models.
- To understand the notion of fairness and how it relates to the behaviour of a system model.
- To have a basic understanding of how SPIN's temporal reasoning mechanism works.

Recommended reading:

- "The Model Checker SPIN", G.J. Holzmann, IEEE Transactions on Software Engineering, Vol 23 (5), 1997.
- "Concise Promela Reference" — see course homepage