Distributed Systems Programming (F21DS1)
SPIN: Formal Analysis II

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.
- 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.
  \( []!(\text{doors==open} \&\& \text{lift==moving}) \)

- **Response:** \( [](p \rightarrow <>q) \)
  
  Every state that satisfies \( p \) is eventually followed by a state that satisfies \( q \), e.g. \( [](\text{call.lift} \rightarrow <>\text{(lift.arrives)}) \)

- **Precedence:** \( [](p \rightarrow (q \cup r)) \)
  
  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.
  \( [](\text{start.lift} \rightarrow (\text{lift.running} \cup \text{stop.running}) \)
Temporal Reasoning in iSPIN

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

\[
\text{ltl} \ [ \ <\text{name}> \ ] \ \{ \ <\text{formula}> \ \}
\]

For example:

\[
\begin{align*}
\text{ltl} \ p1 \ &\{ \ [] \ p \ \} \\
\text{ltl} \ p2 \ &\{ \ [](p \rightarrow <> \ q) \ \} \\
\text{ltl} \ p3 \ &\{ \ [](p \rightarrow (q \ U \ r)) \ \}
\end{align*}
\]

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

\[
\begin{align*}
\text{#define} \ p (x > y) \\
\text{#define} \ q (\text{len(in\_data)} < \text{max}) \\
\text{#define} \ r (x > 0 && x < \text{max})
\end{align*}
\]

▶ Step 3: Enable “use claim” (Never Claims) within the Verification tab and ensure “acceptance cycles” is also enabled, then select “Run” button.

Note: LTL formula can be selectively enabled via the “claim name (opt)” slot.
TrainWare Revisited: Safety Property

\[
\begin{align*}
[] (\text{len}(\text{TunnelAB}) < 2 & \lor \\
\text{len}(\text{TunnelBC}) < 2 & \lor \\
\text{len}(\text{TunnelCD}) < 2 & \lor \\
\text{len}(\text{TunnelDA}) < 2)
\end{align*}
\]

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.
chan TunnelAB = [2] of byte;
chan TunnelBC = [2] of byte;
chan TunnelCD = [2] of byte;
chan TunnelDA = [2] of byte;

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

ltp1[()]

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); /* introduce train 1 before station C */
    run Setup(TunnelDA, 2); /* introduce train 2 before station A */
    run Station(TunnelDA, TunnelAB); /* station A */
    run Station(TunnelAB, TunnelBC); /* station B */
    run Station(TunnelBC, TunnelCD); /* station C */
    run Station(TunnelCD, TunnelDA); /* station D */
}
```c
chan TunnelAB = [2] of { byte };
chan TunnelBC = [2] of { byte };
chan TunnelCD = [2] of { byte };
chan TunnelDA = [2] of { byte };

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

ll p1[] q
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]; /* introduce train 1 before station C */
    run Setup[TunnelDA, 2]; /* introduce train 2 before station A */
    run Station[TunnelDA, TunnelAB]; /* station A */
    run Station[TunnelAB, TunnelBC]; /* station B */
    run Station[TunnelBC, TunnelCD]; /* station C */
    run Station[TunnelCD, TunnelDA]; /* station D */
  }
```
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); /* introduce train 1 before station C */
    run Setup(TunnelDA, 2); /* introduce train 2 before station A */
    run Station(TunnelDA, TunnelBC); /* station A */
    run Station(TunnelBC, TunnelAC); /* station B */
    run Station(TunnelAC, TunnelCD); /* station C */
    run Station(TunnelCD, TunnelDA); /* station D */
}
Modelling Hardware

Integer Division

Modelling Input-Output Relation

::(load==1) \rightarrow a = \text{in1}; b = \text{in2};
   \quad \text{quo} = 0; \text{rem} = a;
   \quad \text{done} = 0;
::(load!=1) \rightarrow \text{if}
   \quad ::(\text{rem}\geq\text{b}) \rightarrow \text{rem} = \text{rem}-\text{b};
   \quad \quad \text{quo} = \text{quo}+1;
   \quad ::(\text{b} > \text{rem}) \rightarrow \text{done} = 1
\text{fi}
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.

```c
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 \( a == ((\text{quo} \times b) + \text{rem}) \).
  - \( \Box ((load == 1 && in1 == a && in2 == b) \rightarrow \langle\rangle (done == 1 && a == ((\text{quo} \times b) + \text{rem}))) \)

- 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:

  \[
  [] (\text{done} == 0 \rightarrow \text{load} == 0) \rightarrow \\
  [] (\text{load} == 1 \land \text{in1} == a \land \text{in2} == b) \rightarrow \\
  <> (\text{done} == 1 \land a == ((\text{quo} \times b) + \text{rem}))
  \]

Note that the extra condition ensures that whenever \text{done} is set to 0 then \text{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!
byte in1, in2;
byte a, b, quo, rem;
bit load = 0, done = 1;

#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))

if p1 { [] (p -> q) -> [] (r & s) -> <> (t & u))

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

proctype env()
{
in1 = 7; in2 = 2; load = 1; /* init inputs */
done = 0; load = 0; done = 1; /* read results */
print("quotient = %d", quo);
print("remainder = %d", rem);

init { atomic{ run quo_rem(); run env(); }

1 atomic steps
hash conflicts: 6 (resolved)

Stats on memory usage (in Megabytes):
0.223 equivalent memory usage for states (stored="State-vector + overhead")
0.387 actual memory usage for states (unsuccessful compression: 173.60%) state vector as stored = 67 byte + 16 byte overhead
2.000 memory used for hash table (w13)
0.343 memory used for DFS stack (w14000)
2.636 total actual memory usage

unreached in proctype quo_rem
reactive:21, state 12, "done = 1"
reactive:24, state 18, "end"
(2 of 18 states)

unreached in proctype env
reactive:28, state 5, "load = 0"
reactive:28, state 6, "((done = 1))"
reactive:29, state 7, "print(quotient = %d, quo)"
reactive:30, state 8, "print(remainder = %d, rem)"
reactive:30, state 9, "end"*
(5 of 9 states)

unreached in init
(0 of 4 states)
unreached in claim p1

spin_inv tmp: 10, state 11, "end"*
(1 of 11 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
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 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.
▶ To have a basic understanding of how SPIN’s temporal reasoning mechanism works.

Recommended reading: