Message passing case: Principles of Protocol Design
Plan

- General design
- Lynch’s protocol
- Stop-and-wait protocol
- Alternating bit protocol
Typical application areas for FSP/SPIN/Uppaal

- Network protocols
- Embedded systems
- Control systems
What is a protocol?

- In general: A set of rules for (good) behaviour.
- In computer systems: A set of rules for component *interaction* in a *distributed* system.
The three parts of a protocol description

- **Protocol semantics:**
  - Service description
  - Environment assumptions (Typical problem: incompleteness. Formalisation helps.)

- **Protocol syntax:**
  - Message vocabulary (protocol commands)
  - Message format and coding

- **Protocol rules for message exchange** (Typical problem: inconsistency and incompleteness. Models and model analysis helps.)
General Design Guidelines

- General design principles.....
- Build prototype model and verify
- Validate through testing:
  - Apply model to identify interesting cases
  - Draw external stimuli from MSC’s
Example: Lynch’s protocol

Semantics (service & environment):
- A user may request a file to be transferred and wait for completion
- The transmission channel is full duplex and error prone
- No data may be lost. Data may not be reordered.

Syntax (vocabulary & coding):
- $V = \{\text{ack}, \text{nack}, \text{err}\}$
- Frame = struct message{V;data}:
  
<table>
<thead>
<tr>
<th>V</th>
<th>data</th>
</tr>
</thead>
</table>

Lynch’s protocol (rules)

1. If data are correctly received, then send ‘ack’ with next message.
2. If error in data, then send ‘nack’ with next message.
3. If ‘nack’ or ‘err’ is received, then resend old data (with ‘ack’ or ‘nack’).
4. Timeout is treated as ‘err’ (convenient addition).
Lynch protocol (diagram)
Lynch protocol (model)

mtype = \{ack, nak, err, next, accept\}
#define message byte

proctype transfer(chan in, out, chin, chout)
{
message o, i;
in?next(o);
do
:: chin?nak(i) -> out!accept(i); chout!ack(o)
:: chin?ack(i) -> out!accept(i); in?next(o); chout!ack(o)
:: chin?err(i) -> chout!nak(o)
:: timeout -> chout!nak(o)
od
}

proctype buggy_chan(chan chin,chout)
{
mtype mt;
message m;
do
:: chin?mt(m) -> chout!mt(m)
:: chin?mt(m) -> chout!err(m)
:: chin?mt(m) -> skip
od
}
Lynch protocol (model continued)

init
{

chan Ain   = [2] of {mtype, message};
chan Aout  = [2] of {mtype, message};
chan Bin   = [2] of {mtype, message};
chan Bout  = [2] of {mtype, message};
chan AtoB1 = [1] of {mtype, message};
chan BtoA1 = [1] of {mtype, message};
chan BtoA2 = [1] of {mtype, message};

atomic {

run send(Ain);
run receive(Aout);
run send(Bin);
run receive(Bout);
run buggy_chan(AtoB1,AtoB2);
run buggy_chan(BtoA1,BtoA2);
run transfer(Ain, Aout, BtoA2, AtoB1);
run transfer(Bin, Bout, AtoB2, BtoA1)
}

proctype send(chan out)
{
    do
        :: out!next(0)
    od
}

proctype receive(chan in)
{
    message i;
    do
        :: in?accept(i)
    od
}

Correct?
Lynch protocol (model continued)

```plaintext
init
{
  chan Ain   = [2] of {mtype, message};
  chan Aout  = [2] of {mtype, message};
  chan Bin   = [2] of {mtype, message};
  chan Bout  = [2] of {mtype, message};
  chan AtoB1 = [1] of {mtype, message};
  chan BtoA1 = [1] of {mtype, message};
  chan BtoA2 = [1] of {mtype, message};

  atomic {
    run send(Ain);
    run receive(Aout);

    run send(Bin);
    run receive(Bout);

    run buggy_chan(AtoB1, AtoB2);
    run buggy_chan(BtoA1, BtoA2);

    run transfer(Ain, Aout, BtoA2, AtoB1);
    run transfer(Bin, Bout, AtoB2, BtoA1)
  }
}
```

proctype send(chan out)
{
  do
    :: out!next(0) -> out!next(1)
  od
}

proctype receive(chan in)
{
  message i,j;
  do
    :: in?accept(i);
      in?accept(j);
      assert(i != j)
  od
}

Validate lynchnew (depth=50)
Example: Stop-and-wait protocol

Semantics (service & environment):
- The protocol must offer an acknowledged connection-oriented service.
- The connection is simplex (one-way).
- The transmission channel is full duplex and error prone
- No data may be lost. Data may not be reordered.

Syntax (vocabulary & coding):
- $V = \{\text{ack, data}\}$
- Frame = struct message{$V; data$}:
Stop-and-wait protocol rules

1. Receiver protocol unit: If data are correctly received, then send ‘ack’ to sender (and deliver data to the receiving user).

2. Sender protocol unit: If ‘ack’ is received, then get the next frame from the sending user, and transmit it to the receiver via the channel.

3. Sender protocol unit: Retransmit the current frame at timeout.
Stop-and-and-wait protocol (diagram)
Stop-and-wait model

proctype buggy_chan(chan chin, chout)
{
    mtype mt;
    message m;
    do
    :: chin?mt(m) -> chout!mt(m)
    :: chin?mt(m) -> skip
    od
}

proctype sender2(chan in, chin, chout)
{
    message o, i;
    do
    :: in?next(o); chout!data(o);
    do
    :: chin?ack(i); break
    :: timeout -> chout!data(o)
    od
    od
}

proctype reader2(chan out, chin, chout)
{
    message o, i;
    do
    :: chin?data(i) -> out!accept(i); chout!ack(o)
    od
}

mtype = {ack, data, next, accept}
#define message byte
Stop-and-wait: correctness check

proctype send(chan out)
{
    do
    :: out!next(0) -> out!next(1)
    od
}

proctype receive(chan in)
{
    message i,j;
    do
    :: in?accept(i);
    in?accept(j);
    assert(i != j)
    od
}

init
{
    chan Ain = [2] of {mtype, message};
    chan Bout = [2] of {mtype, message};
    chan AtoB1 = [1] of {mtype, message};
    chan BtoA1 = [1] of {mtype, message};
    chan BtoA2 = [1] of {mtype, message};
}

atomic {
    run send(Ain);
    run receive(Bout);
    run buggy_chan(AtoB1,AtoB2);
    run buggy_chan(BtoA1,BtoA2);
    run sender2(Ain, BtoA2, AtoB1);
    run reader2(Bout, AtoB2, BtoA1)
}
Verifying data-independent protocols

- Question: How many different values must be sent via a data independent protocol in order to prove absence of loss and reordering?

- Answer (Wolper’85): 3 different values suffice! (More precisely: one more than the number of different values in the correctness formulation – in our case two).
Revised stop-and-wait model

mtype = {ack, data, next, accept, white, red, blue}
#define message byte

proctype sender2(chan in, chin, chout)
{
    message o, i;

    do
    :: in?next(o); chout!data(o);
    do
    :: chin?ack(i); break
    :: timeout -> chout!data(o)
    od
    od
}

proctype reader2(chan out, chin, chout)
{
    message o, i;

    do
    :: chin?data(i) -> out!accept(i); chout!ack(o)
    od
}

proctype buggy_chan(chan chin, chout)
{
    mtype mt;
    message m;
    do
    :: chin?mt(m) -> chout!mt(m)
    :: chin?mt(m) -> skip
    od
}
Revised sop-and-wait correctness check

proctype send(chan out)
{ mtype sent=white;
do ::
do:: sent = white; out!next(sent)
:: sent = red; out!next(sent); break
od;
do:: sent = white; out!next(sent)
:: sent = blue; out!next(sent); break
od
}

proctype receive(chan in)
{ mtype rcvd = white;
do ::
do:: in?accept(rcvd);
if:: (rcvd==white) -> skip:: (rcvd==red) -> break
:: (rcvd==blue) -> assert(0)
fi
od;
do:: in?accept(rcvd);
if:: (rcvd==white) -> skip:: (rcvd==red) -> assert(0)
:: (rcvd==blue) -> break
fi
od
}

Validate tan2 (depth=50)
Example: Alternating bit protocol

Semantics (service & environment):
- The protocol must offer an acknowledgement service.
- The connection is duplex (two-way).
- The transmission channel is full duplex and error prone.
- No data may be lost. Data may not be reordered.

Syntax (vocabulary & coding):
- V = \{ack\}
- Frame = struct message{V; s_seq; ack_no; data}:
Alternating bit protocol rules

1. Each entity maintains two 0/1 counters \( s_{\text{seq}} \) (next sequence number) and \( r_{\text{seq}} \) (expected sequence number from peer entity).

2. At timeout: resend current output as \( (\text{ack}, s_{\text{seq}}, 1-r_{\text{seq}}, \text{data}) \).

3. At \( (\text{ack}, s, a, \text{data}) \) from network:
   1. Accept data if \( s=r_{\text{seq}} \) and update \( r_{\text{seq}} \).
   2. Get next data item, if \( a=s_{\text{seq}} \) and update \( s_{\text{seq}} \).

4. Send current output as \( (\text{ack}, s_{\text{seq}}, 1-r_{\text{seq}}, \text{data}) \).
Alternating bit protocol: diagram
Alternating bit protocol model

\[
mtype = \{\text{ack, next, accept, white, red, blue}\}
\]

```c
#define message byte

proctype transfer(chan in, out, chin, chout)
{ byte s_seq=0, r_seq=0, s, r;
  message o, i;
  in?next(o);
  chout!ack(s_seq,1-r_seq,o);
  do
  :: chin?ack(s,r,i) ->
    if
      :: s==r_seq -> out!accept(i); r_seq = 1-r_seq
      :: else skip
    fi;
    if
    :: r==s_seq -> in?next(o); s_seq=1-s_seq
    :: else skip
    fi
  :: timeout -> chout!ack(s_seq,1-r_seq,o)
  :: chout!ack(s_seq,1-r_seq,o)
  od
}
```
Alternating bit protocol: correctness check

```
init
{
  chan Ain  = [0] of {mtype, message};
  chan Aout = [0] of {mtype, message};
  chan Bin  = [0] of {mtype, message};
  chan Bout = [0] of {mtype, message};
  chan AtoB1 = [0] of {mtype, byte,byte,byte};
  chan AtoB2 = [0] of {mtype, byte,byte,byte};
  chan BtoA1 = [0] of {mtype, byte,byte,byte};
  chan BtoA2 = [0] of {mtype, byte,byte,byte};

  atomic {
    run send(Ain);
    run receive(Aout);
    run send(Bin);
    run receive(Bout);
    run buggy_chan(AtoB1,AtoB2);
    run buggy_chan(BtoA1,BtoA2);
    run transfer(Ain, Aout, BtoA2, AtoB1);
    run transfer(Bin, Bout, AtoB2, BtoA1);
  }
}
```

Validate smallnew
Depth 10000, no endstates checked
LTL verification of Alternating bit protocol

- We want to verify the order preservation property of the protocol.

- Illegal behaviour: No red frame is received before a blue frame.

- In LTL $\neg r \neg r \ U \ rb$.

- We check that no execution contains this behaviour (verify small-LTL)
Supertrace algorithm

The problem: Given M bytes of memory, how can we search the state space using precisely M bytes?

Basic idea:
- Use the M bytes as a hash-table of investigated states.
- Allocate 1 bit per entry (actually 2 bits are used).
- Cut-off the search, when a hash conflict is met.
Supertrace algorithm
Supertrace coverage

- Hash factor: ratio between table size and the number of stored values (number of set bits).
- Empirical evidence:
  - Hash factor close to 100 gives more than 99.9% coverage of state space.
  - Hash factor > 10 gives more than 90% coverage of state space.
- Conclusions:
  - All model errors can be found.
  - Correct models can *almost* be verified.
Example: buffered alternating bit protocol

chan Ain  = [0] of {mtype, message};
chan Aout = [0] of {mtype, message};
chan Bin  = [0] of {mtype, message};
chan Bout = [0] of {mtype, message};
chan AtoB1 = [1] of {mtype, byte,byte,byte};
chan AtoB2 = [1] of {mtype, byte,byte,byte};
chan BtoA1 = [1] of {mtype, byte,byte,byte};
chan BtoA2 = [1] of {mtype, byte,byte,byte};

Validate medium4 (depth 100000, state space = 1000)
Result of supertrace analysis

(Spin Version 3.4.12 -- 18 December 2001)
+ Partial Order Reduction

Bit statespace search for:
never-claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid endstates - (disabled by -E flag)

State-vector 156 byte, depth reached 93393, errors: 0
1.22816e+07 states, stored
1.08697e+07 states, matched
2.31513e+07 transitions (= stored+matched)
7 atomic steps
hash factor: 10.9284 (expected coverage: >= 98% on avg.)
(max size 2^27 states)

Stats on memory usage (in Megabytes):
1965.050 equivalent memory usage for states (stored*(State-vector + overhead))
33.554 memory used for hash-array (-w27)
2.800 memory used for DFS stack (-m100000)
36.969 total actual memory usage
Conclusions

- Protocol design is based on message passing
- SPIN as a modelling/validation tool
- Normally three different values suffices for verification of data independant protocols
- Supertracing is often needed in realistic cases