Embedded Systems Specification and Design
Model-based Design and Verification

David Kendall

Outline

Bounded Channel

```
---- MODULE BoundedChannel ----
EXTENDS Integers, Sequences

ISeq(S) == [(Nat \ {0}) -> S]
IHead(iseq) == iseq[1]
ITail(iseq) == [i \in (Nat \ {0}) |-> iseq[i + 1]]

CONSTANT
  N,
  Msg,
  Input

ASSUME
  /\ N \in (Nat \ {0})
  /\ Input \in ISeq(Msg)
```

Bounded Channel (ctd)

- Pluscal algorithm – variables and sending process

```
(*
--algorithm BChan
  variable
    in = Input;
    ch = << >>;
    out = << >>;

  process Send = 1
  begin
s:  while (TRUE) do
      await Len(ch) < N;
      ch := Append(ch, IHead(in));
      in := ITail(in);
    end while
  end process
```

Bounded Channel (ctd)

- Pluscal algorithm – receiving process

```
  process Receive = 2
  begin
r:  while TRUE do
      await Len(ch) > 0;
      out := Append(out, Head(ch));
      ch := Tail(ch);
    end while
  end process
end algorithm
*)
```

Bounded Channel - TLA$^+$ translation

- TLA$^+$ translation – variables and `Init` operation

```
\* BEGIN TRANSLATION
VARIABLES in, ch, out, pc

vars == << in, ch, out, pc >>

ProcSet == {1} \cup {2}

Init == (* Global variables *)
  /\ in = Input
  /\ ch = << >>
  /\ out = << >>
  /\ pc = [self \in ProcSet |-> CASE self = 1 -> "s"
                                [] self = 2 -> "r"]
```

Bounded Channel - TLA$^+$ translation

- TLA$^+$ translation – sending process

```
s == /\ pc[1] = "s"
     /\ Len(ch) < N
     /\ ch' = Append(ch, IHead(in))
     /\ in' = ITail(in)
     /\ pc' = [pc EXCEPT ![1] = "s"]
     /\ out' = out

Send == s
```

Bounded Channel - TLA$^+$ translation

- TLA$^+$ translation
  - receiving process
  - `Next` and `Spec` operations

```
r == /\ pc[2] = "r"
     /\ Len(ch) > 0
     /\ out' = Append(out, Head(ch))
     /\ ch' = Tail(ch)
     /\ pc' = [pc EXCEPT ![2] = "r"]
     /\ in' = in

Receive == r

Next == Send \/ Receive

Spec == Init /\ [][Next]_vars
\* END TRANSLATION
```

Bounded Channel Properties - TypeOk

```
TypeOk ==
  /\ in \in ISeq(Msg)
  /\ ch \in Seq(Msg)
  /\ out \in Seq(Msg)
```

- The input, `in`, is an infinite sequence of messages
- The *message channel*, `ch`, is a finite sequence of messages (bounded by `N`)
- The output, `out`, is a finite sequence of messages (the messages received so far from `in` via `ch`)

Bounded Channel Properties - Safety

```
seq ** iseq == [i \in (Nat \ {0}) |->
                  IF i <= Len(seq) THEN seq[i]
                                   ELSE iseq[i - Len(seq)]]
```

- `**` is a 'helper' operator used in specifying the safety invariant, `Inv`, below
- it is a version of the concatenation operator that allows us to add an infinite sequence on to the end of a finite sequence

```
Inv ==
  /\ TypeOk
  /\ Len(ch) <= N
  /\ Input = (out \o ch) ** in
```

- Key idea - the messages in the output, plus the messages in the channel, plus the remaining input messages give us the original messages in `Input`, i.e. no lost messages and no duplicates

Model constraints

There are some aspects of this specification that prevent us from using TLC to check it, without modifying the model

What is the model?

- Need to give values for `N`, `Msg` and `Input`
- Notice that TLC cannot check a model when `Input` is an infinite sequence

Definition overrides

- `ISeq`, `ITail`, and `**` need to be overridden by operators on finite sequences

State constraints

- Having made `Input` finite, we need to impose a constraint so that we don't try to explore the state space further, once the `Input` has been used up

Bounded Channel Properties - Liveness

```
Liveness ==
  \A i \in DOMAIN Input, j \in 1..N :
    (Len(out) = i) /\ (Len(ch) = j) ~> (Len(out) = i + j)
```

We want to check that any message that is sent is eventually received

One way of doing this is to check that if we have received *i* messages so far (`Len(out) = i`) and there are *j* messages in the channel (`Len(ch) = j`), then eventually we will have received $i + j$ messages (`Len(out) = i + j`)

We'll need to ensure that the receive process is weakly fair

Note that we don't require that the sender keeps sending messages; only that they are received if they are sent

Notice some typos in Lamport's hyperbook treatment of these issues