MODULE *StateTransfer*

This specification describes both the allowable behaviours and liveness
of a system that transfers dynamically-changing state from a *mutator*
to an observer via snapshot and delta messages.

EXTENDS *Naturals*, *Sequences*

Model checking constants

For bounded checking, our models provide the following
constant values.

The set of values from which a *mutator* might pick as its next state.
CONSTANT *PickableValues*

A value, not in *PickableValues*, that signifies an observer has not yet observed
a *mutator*-chosen value.
CONSTANT *Empty*
ASSUME $Empty \notin PickableValues$

A value, not in *PickableValues*, that allows the simulation of a partial function
in *Mutations* below.
CONSTANT *Invalid*
ASSUME $Invalid \notin PickableValues$

The maximum number of messages that the (*FIFO*) network can hold at any time.
CONSTANT *NetworkCapacity*
ASSUME $NetworkCapacity \in Nat$

Model state

The evolving state of the specification comprises the three variables below.

The value last picked by the *mutator*.
VARIABLE *mutator*

The value (if any) last observed by the observer (since connecting).
VARIABLE *observer*

The queue of messages from the *mutator* to the observer.
It is emptied upon a disconnection.
VARIABLE *network*

$vars \triangleq \langle mutator, observer, network \rangle$

Types

Below we describe the structure of the types that the
variables above inhabit.

The set of possible mutations that the *mutator* may perform.

$$Mutations \triangleq [PickableValues \to PickableValues \cup \{Invalid\}]$$

The set of possible snapshot messages the *mutator* may send
to the observer. At the time of sending, it contains
the *mutator*'s latest picked value, *i.e.*, its state.

$$SnapshotMessages \triangleq [type : \{\text{“snapshot"}\},\ value : PickableValues]$$

The set of possible delta messages the *mutator* may send
to the observer. At the time of sending, it contains
a function that relates the previous state to the current
state. In a real implementation, the *mutator* might send
a well-known event rather than a “mapping”.

$$DeltaMessages \triangleq [type : \{\text{“delta"}\},\ event : Mutations]$$

The set of messages the *mutator* may send to the observer.

$$Messages \triangleq SnapshotMessages \cup DeltaMessages$$

TypeOK is an invariant that *TLC* checks. It ensures that, in all states,
our variables are assigned values of the types we expect.

$$\begin{aligned} TypeOK \triangleq\ & \\ & \wedge\ mutator \in PickableValues && \text{The } mutator \text{ state is a value it has picked.} \\ & \wedge\ observer \in (PickableValues \cup \{Empty\}) && \text{The observer state is a value that it has observed or empty.} \\ & \wedge\ network \in Seq(Messages) && \text{The network is a queue of messages from the } mutator \text{ to the observer.} \end{aligned}$$

Behaviour

In this section, we describe the behaviours of our system.

The system starts in an *Init* state, where the network and observer are empty,
but the *mutator* has already picked a value.

$$\begin{aligned} Init \triangleq\ & \\ & \wedge\ mutator \in PickableValues \\ & \wedge\ observer = Empty \\ & \wedge\ network = \langle\rangle \end{aligned}$$

An action the *mutator* might take, if there is capacity on the network, is to
send a snapshot of the value it last picked. In a real system, the observer

may request a snapshot, as an optimisation.

$$
\begin{array}{l}
\mathit{TransferSnapshot} \triangleq \\
\quad \land \mathit{Len}(\mathit{network}) < \mathit{NetworkCapacity} \\
\quad \land \mathit{network}' = \mathit{Append}(\mathit{network}, [\mathit{type} \mapsto \text{“snapshot”}, \mathit{value} \mapsto \mathit{mutator}]) \\
\quad \land \text{UNCHANGED } \langle \mathit{mutator}, \mathit{observer} \rangle
\end{array}
$$

Another action the *mutator* might take, if there is capacity on the network, is to pick a new value and send an event to the observer that describes the transition.

$$
\begin{array}{l}
\mathit{TransferDelta} \triangleq \\
\quad \land \mathit{Len}(\mathit{network}) < \mathit{NetworkCapacity} \\
\quad \land \exists\, \mathit{next} \in \mathit{PickableValues} : \\
\qquad \text{LET} \\
\qquad\quad \mathit{event} \triangleq [v \in \mathit{PickableValues} \mapsto \text{IF } v = \mathit{mutator} \text{ THEN } \mathit{next} \text{ ELSE } \mathit{Invalid}] \\
\qquad \text{IN} \\
\qquad\quad \land \mathit{mutator}' = \mathit{next} \\
\qquad\quad \land \mathit{network}' = \mathit{Append}(\mathit{network}, [\mathit{type} \mapsto \text{“delta”}, \mathit{event} \mapsto \mathit{event}]) \\
\qquad\quad \land \text{UNCHANGED } \langle \mathit{observer} \rangle
\end{array}
$$

Note:
I wonder the specification would be more general/applicable if the *mutator*'s actions of "picking a new value" and "sending a delta" were not tied together. For example, that might allow the specification to describe systems employing delta conflation strategies.

The only action to observer can take, when it receives a message over the network, is to accumulate state.

$$
\begin{array}{l}
\mathit{Accumulate} \triangleq \\
\quad \land \mathit{Len}(\mathit{network}) > 0 \\
\quad \land \mathit{network}' = \mathit{Tail}(\mathit{network}) \\
\quad \land \lor \land \mathit{Head}(\mathit{network}).\mathit{type} = \text{“snapshot”} \quad \text{A new value} \\
\qquad\quad \land \mathit{observer}' = \mathit{Head}(\mathit{network}).\mathit{value} \\
\qquad \lor \land \mathit{Head}(\mathit{network}).\mathit{type} = \text{“delta”} \qquad \text{An applicable change in value} \\
\qquad\quad \land \mathit{observer} \in \text{DOMAIN } \mathit{Head}(\mathit{network}).\mathit{event} \\
\qquad\quad \land \mathit{observer}' = \mathit{Head}(\mathit{network}).\mathit{event}[\mathit{observer}] \\
\qquad \lor \land \mathit{Head}(\mathit{network}).\mathit{type} = \text{“delta”} \qquad \text{An inapplicable change in value} \\
\qquad\quad \land \mathit{observer} = \mathit{Empty} \\
\qquad\quad \land \text{UNCHANGED } \langle \mathit{observer} \rangle \\
\quad \land \text{UNCHANGED } \langle \mathit{mutator} \rangle
\end{array}
$$

Occasionally, the network might disconnect the *mutator* from the observer and lose all its in-flight messages. The observer detects this drop in connection and resets its state to *Empty* to avoid processing changes after a gap.

$$
\begin{array}{l}
\mathit{Disconnect} \triangleq \\
\quad \land \mathit{network}' = \langle\rangle \\
\quad \land \mathit{observer}' = \mathit{Empty}
\end{array}
$$

$\land$ UNCHANGED $\langle mutator \rangle$

After an *Init* state, the system can transition to another state via any of these actions.

$Next \triangleq$
- $\lor$ *TransferSnapshot*
- $\lor$ *TransferDelta*
- $\lor$ *Accumulate*
- $\lor$ *Disconnect*

Together the *Init* states and *Next* action define the allowed behaviours of our system.

$Spec \triangleq$
- $\land$ *Init*
- $\land \Box[Next]_{vars}$

FairSpec adds assumptions about the fairness of certain actions to the behaviours of *Spec*, thus limiting the behaviours to those that satisfy the *StateTransfers* property.

$FairSpec \triangleq$

$\land$ *Spec*	Must be a behaviour that satisfies *Spec*.
$\land \mathrm{WF}_{vars}(TransferDelta)$	Any behaviour where *TransferDelta* is constantly enabled must eventually perform it.
$\land \mathrm{WF}_{vars}(TransferSnapshot)$	Any behaviour where *TransferSnapshot* is constantly enabled must eventually perform it.
$\land \mathrm{SF}_{vars}(Accumulate)$	Any behaviour where *Accumulate* is repeatedly (but not necessarily constantly) enabled must eventually perform it.

Properties

This property says that the observer evenutally witnesses any value that the *mutator* picks.

$$StateTransfers \triangleq \Box(\forall\, v \in PickableValues : mutator = v \rightsquigarrow \lor observer = v \lor observer = Empty)$$