Specification of a Distributed System

Kerry Raymond
Tim Mansfield

Key Centre for Software Technology
Department of Computer Science
University of Queensland
St. Lucia QLD 4067
AUSTRALIA

Discussion Document No. 8

21st March 1989*

Abstract

A library of standard Z definitions for work in the Telecom project is presented. The model covers point-to-point message passing with broad- and multi-casting.

1 Messages

Messages are the "bottom line" in a distributed system. They come from one process and are sent to a set of processes. Their data is just a bit string. SID and RID are the generic identity sets for senders and receivers respectively.

$$Bit \mathrel{\widehat{=}} 0|1$$
$$BitString \mathrel{\widehat{=}} \operatorname{seq} Bit$$

$Message[SID, RID]$

$$
\begin{array}{|l}
tag : TAG \\
from : SID \\
to : \mathbb{P}\, RID \\
data : BitString \\
\hline
|to| \geq 1
\end{array}
$$

Messages have at least one destination. The message tag is used to distinguish one message from another.

*printed September 4, 1989

2 Messages Boxes

Message boxes are collections of messages.

$$Box \mathrel{\widehat{=}} \mathbb{P}\, Message$$

Some message boxes are for input and some for output.

MessageSystem[*SID*, *RID*]

$$\begin{array}{l}
InTrays : RID \rightarrow Box \\
OutTrays : SID \rightarrow Box \\
UsedTags : \mathbb{P}\, TAG \\
\hline
\forall i : RID \bullet \\
\quad \forall m : InTrays(i) \bullet m.to = \{i\} \\
\forall i : SID \bullet \\
\quad \forall m : OutTrays(i) \bullet i = m.from \\
\quad \forall m : OutTrays'(i) \bullet \\
\quad\quad OutTrays(i) = OutTrays'(i) - \{m\} \Rightarrow \\
\quad\quad\quad m.tag \notin UsedTags \\
\quad\quad\quad UsedTags' = UsedTags \cup \{m.tag\}
\end{array}$$

After delivery, messages only carry the identity of this receiving entity, not all. If it is desired that a receiver knows the identity of the other receivers, then the sender must enclose this information in the data part of the message.
The tags are used to ensure that each message is unique.

Initially

$$\begin{array}{l}
\forall i : RID \bullet \\
\quad InTrays(i) = \{\,\} \\
\forall i : SID \bullet \\
\quad OutTrays(i) = \{\,\} \\
UsedMsgs = \{\,\}
\end{array}$$

$$
\begin{array}{l}
\textit{i_Transfer} \\
\hline
\exists s : SID \bullet \\
\quad \exists m : OutTray(s) \bullet \\
\quad\quad \exists rs : \mathbb{P}\, RID \bullet \\
\quad\quad\quad rs \subseteq m.to \\
\quad\quad\quad |rs| \geq 1 \\
\quad\quad\quad \forall r : rs \bullet \\
\quad\quad\quad\quad \exists rm : Message \bullet \\
\quad\quad\quad\quad\quad rm \backslash to = m \backslash to \\
\quad\quad\quad\quad\quad rm.to = \{j\} \\
\quad\quad\quad\quad\quad InTrays'(r) = InTrays(r) \cup \{rm\} \\
\quad\quad\quad rs = m.to \Rightarrow \\
\quad\quad\quad\quad OutTrays'(s) = OutTrays'(s) - \{m\} \\
\quad\quad\quad rs \subset m.to \Rightarrow \\
\quad\quad\quad\quad \exists sm : Message \bullet \\
\quad\quad\quad\quad\quad sm \backslash to = m \backslash to \\
\quad\quad\quad\quad\quad sm.to = m.to - rs \\
\quad\quad\quad\quad\quad OutTrays'(s) = (OutTrays(s) - \{m\}) \cup \{sm\} \\
\quad\quad\quad \forall or : RID \bullet \\
\quad\quad\quad\quad or \notin rs \Rightarrow \\
\quad\quad\quad\quad\quad InTrays'(or) = InTrays(or) \\
\quad \forall os : SID \bullet \\
\quad\quad os \neq s \Rightarrow \\
\quad\quad\quad OutTrays'(os) = OutTrays(os) \\
\hline
\end{array}
$$

i_Transfer delivers a message to some of its destinations. Each destination is not aware that other destinations receive this message (unless this information is conveyed in the data). The message held in the OutTray records only the destinations still to be delivered to, and the message is removed from the OutTray when all deliveries have been made.
The prefix "i_" is a naming convention adopted to denote an internal operation. This operation is not intended to be visible to other schemas.

3 Direct Naming Communication

DirectNaming defines the process interface to message passing using a direct naming style of communication. For this, a process needs a name and two mailboxes, for incoming and outgoing messages.

$DirectNaming[PID]$

$InBox, OutBox : Box$
$self : PID$

$\forall m : OutBox \bullet$
$\quad self = m.from$
$\forall m : InBox \bullet$
$\quad m.to = \{self\}$

$Initially$

$InBox = OutBox = \{\}$

$Send$

$to : \mathbb{P}\, PID$
$data : BitString$

$\exists m : Message \bullet$
$\quad m.from = self$
$\quad m.to = to$
$\quad m.data = data$
$\quad OutBox' = OutBox \cup \{m\}$
$unchanged(self, InBox, to, data)$

$Receive$

$from : \mathbb{P}\, PID$
$data : BitString$

$\exists m : InBox \bullet$
$\quad m.from = from$
$\quad self \in m.to$
$\quad data = m.data$
$\quad InBox' = InBox - \{m\}$
$unchanged(self, OutBox, from, data)$

4 Message Types

It seems the best way to handle data of a message is to declare a schema for each message type, and construct AnyMsg as a disjoint union of them. For example, here are 2 message types, a TOKEN with an integer value and a REQUEST (which our example process will not be using).

Token

$counter : \mathbb{N}$

$counter\ mod\ 2 = 0$
{ An example constraint }

Request

{ declarations }

{ predicates }

$$AnyMsg \mathrel{\widehat=} Token | Request$$

We need to define the set of all messages.

5 Process

Process is probably the level at which we wish to develop our algorithms. Here is an example of a process which passes a token around from one neighbour to the next.

BasicProcess[PID, MsgTypes]

$comms : DirectNaming[PID]$
$self : PID$
$pack : MsgTypes \rightarrowtail (ANY \twoheadrightarrow BitString)$

$\forall X : MsgTypes \bullet$
$\quad pack(X) \in X \rightarrowtail BitString$
$\# \mathrm{ran}(\mathrm{ran}(pack)) = \sum_{X:MsgTypes} \# \mathrm{ran}(pack(X))$

Although declared more generally, the functions returned by pack are in fact 1-to-1 mappings from the particular message type. Each function returned by pack has its own individual sets of bit-strings. If not, the count of BitStrings in total (# ran (ran (pack))) would not equal the sum of the individual range sizes.

Initially

$comms.Initially$
$comms.self = self$

$$Send[X] \mathrel{\widehat=} i_Send[X] \setminus (data, from)$$

$i_Send[X]$

$msg : X$

$X \in MsgTypes$
$comms.Send$
$(pack(X))(msg) = data$

$Receive[X] \;\widehat{=}\; i_Receive[X] \setminus (data, to)$

$i_Receive[X]$

$msg : X$

$X \in MsgTypes$
$comms.Receive$
$(pack(X))(msg) = data$

$Process[PID]$

$BasicProcess[PID, AnyMsg]$
$havetoken : Boolean$
$tokenval : \mathbb{N}$

$Initially$

{ BasicProcess's Initially is included here }
$havetoken \Leftrightarrow self = PID_{min}$
$havetoken \Rightarrow tokenval = 0$

$GetToken$

$msg : Token$
$Receive[Token]$
$from = pred_{PID}(self)$
$tokenval' = msg.counter$
$havetoken'$

$PassToken$

$havetoken$
$Send[Token]$
$to = \{succ_{PID}(self)\}$
$msg.counter = tokenval + 2$
$\neg\, havetoken'$

Now this makes Send and Receive look at lot neater in Process (which is good as it is the

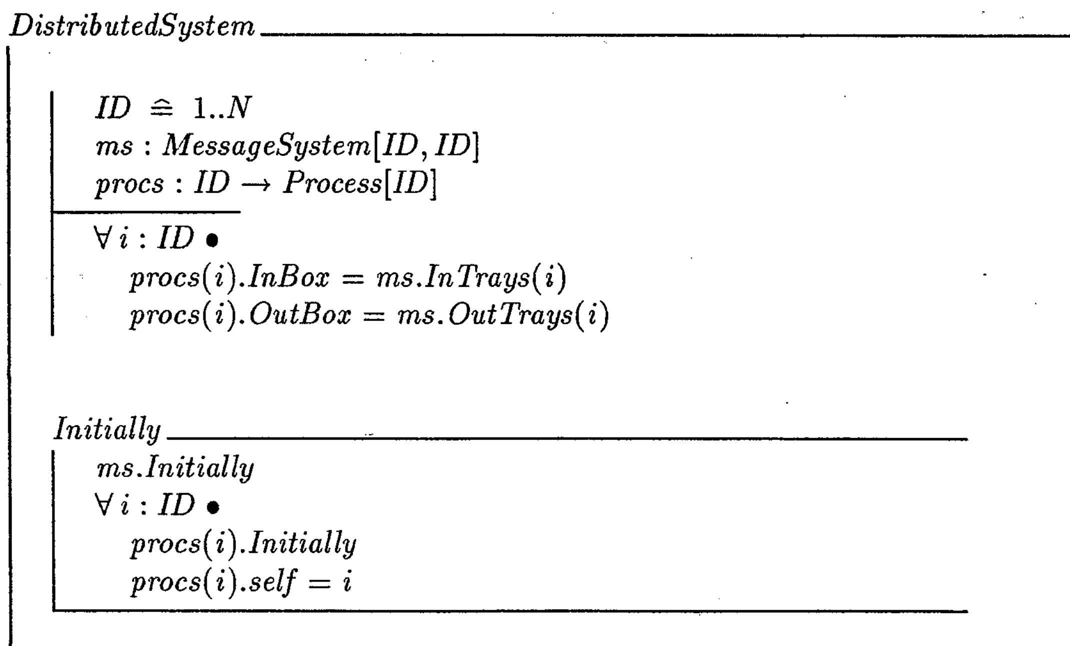
one that it rewritten for each new algorithm). Warning: each Send and Receive generic schema introduces “msg” (appropriately typed). Therefore a schema must not contain Send/Receives of different MsgTypes because the declarations of “msg” would clash. Even where there are Send/Receives of the same type, the “msg” used in each must be the same message. This is not a problem as renaming can be used:

$$Send[Token][tmsg/msg]$$

where tms is introduced in the declaration part or in a quantifier.

6 Distributed System

A distributed system is a collection of processes and a message system, where the inboxes and outboxes of the processes are bound to those of the message system. We will assume that the nodes are identified by the numbers 1 to N, with pred and succ with the obvious circular definitions.



7 Conclusions

The purpose of this exercise is to develop a suite of Z schemas which are reusable in the specification of a variety of distributed algorithm. We believe the following schemas are reusable for a wide variety of distributed algorithms: *DistributedSystem*, *MessageSystem*, *Message*, *BasicProcess* and *DirectNaming*.

The following schemas are intended to be rewritten for each algorithm: *Process* (which specifies the algorithm itself) and *Token* and *Request* (or whatever message types are used by the algorithm). The type *AllMsgs* (or name of your choice) must be the union of all of the required message types (e.g. *Request* and *Token*).