

Using Z for Distributed Systems Specification*

Kerry Raymond
Tim Mansfield

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

Proceedings of ACSC-13
Vol 12, Number 1
pp 314-321

30th June 1989[†]

Abstract

This is a draft specification of a distributed system in Z [1] which is intended to form a basis for the subsequent specification of distributed algorithms. The idea is to create a reusable library which defines a distributed system as a set of processes communicating via an underlying message passing system. This library forms an envelope which can encompass the algorithmic specification of the individual processes.

With such a reusable library, we can minimize the effort involved in the specification of particular algorithms, increasing clarity and reducing errors. By attempting to make the library modular, it will be possible to "plug in" alternate components, e.g. replacing a direct-naming model of communication with a mailbox model of communication (global naming).

1 Introduction

With a little experience in specifying algorithms for distributed systems, it becomes apparent that much of the specification is replicated every time. Nearly every distributed algorithm describes a set of communicating processes which have no shared variables. The processes typically communicate by passing messages through some medium and the message-passing is (predictably) non-instantaneous.

The constraints on message-passing may differ between algorithms, but the basic requirements are the same. It seems useful, then, to specify these common components in a reusable manner, as "modules", so that sections of the specification need not be recreated time and again.

Creating such a library of specification modules means that future specifications needn't be cluttered with details of well-understood components. In addition, the number of errors in the specification should be reduced simply by reducing the sheer volume of text that needs to be written.

The most obvious requirement for the library is that it has a good modular interface to the process definition, since the latter is the component which will change from algorithm to algorithm. However, it is useful if the library has an internally modular structure so that some aspects of the system (e.g. the style of channel naming, the determinism of message delivery, etc.) can be altered.

The specifications which follow form the foundations of just such a library. They are written in Z [1] since we believe it is a suitably abstract language for specification and we are familiar with it. An abstract specification of a message and a medium with some unpredictable transmission delay are given as well

[†] printed November 20, 1989

*This work has been supported by Telecom Research Laboratories

as a specification of a direct-naming style of communication. A simple example process is then defined as well as a reusable definition of a distributed system of communicating processes.

Following the specification of the modules is a discussion of their interfaces.

2 Messages

Messages are the "bottom line" in a distributed system. They are sent by a sending entity and received by a receiving entity. *SID* and *RID* are the generic sets of identifiers for senders and receivers respectively. These will be discussed in a later section.

The information content of a message is of type *DATA* (defined in a later section).

$[SID, RID, DATA]$

Message

<i>from</i> : <i>SID</i> <i>to</i> : <i>RID</i> <i>data</i> : <i>DATA</i>

3 Message Delivery Service

We need to capture the concept of the delays incurred in message passing. As we are not interested in the details of networks and communications, we can divide unreceived messages into two groups, those that have been delivered to their destinations, and those that have not.

Since there may be many messages with the same content sent from the same source to the same destination, these two groups must be represented as bags rather than sets.

$MailBag \cong \text{bag } Message$

MessageSystem

<i>delivered</i> : <i>MailBag</i> <i>enroute</i> : <i>MailBag</i>
--

Initially the message system has no messages at all.

MessageSystemInitially

<i>MessageSystem</i>

$delivered = enroute = []$

Messages are added to the undelivered mailbag by *PutMessage*, and extracted from the delivered mailbag by *GetMessage*.

PutMessage

$\Delta MessageSystem$ <i>m</i> : <i>Message</i>

$enroute' = enroute \uplus [m]$ unchanged (<i>delivered</i>)

<i>GetMessage</i>
Δ <i>MessageSystem</i>
$m : \textit{Message}$
$m \in \text{dom } \textit{delivered}$
$\textit{delivered}' \uplus \llbracket m \rrbracket = \textit{delivered}$
unchanged (<i>enroute</i>)

MessageTransfer shifts a message from the *enroute* MailBag to the *delivered* MailBag, specifying the actions of the network.

<i>MessageTransfer</i>
Δ <i>MessageSystem</i>
$\exists m : \text{dom } \textit{enroute} \bullet$
$\textit{enroute}' = \textit{enroute} \uplus \llbracket m \rrbracket$
$\textit{delivered}' \uplus \llbracket m \rrbracket = \textit{delivered}$

4 Direct Naming Communication

DirectNaming defines the process interface to message passing using a direct naming style of communication, i.e. each message is addressed to a particular process. Therefore each process must have a unique identification which we call a *PID* which we introduce as a generic type.

$\llbracket \textit{PID} \rrbracket$

Each process also contains a message system. Later we will introduce the constraint that all processes contain the same message system (or else communication would not occur!).

In direct naming, the sending and receiving entities are processes, and hence we can define that the *SIDs* and *RIDs* from section 3 are just *PIDs*.

<i>DirectNaming</i>
<i>MessageSystem</i>
$\textit{self} : \textit{PID}$
$\textit{PID} = \textit{RID} = \textit{SID}$

The initialization of *DirectNaming* is trivial, but we include it for reasons of modularity (it shields *Process* from *MessageSystem*).

<i>DirectNamingInitially</i>
<i>DirectNaming</i>
<i>MessageSystemInitially</i>

Sending and receiving are defined at this level by composing/decomposing the components of the message, and then using *PutMessage* and *GetMessage* to associate this message with the message system.

Any message sent comes from "self", and any message received must be addressed to "self".

<i>Send</i>
Δ <i>DirectNaming</i>
<i>to</i> : <i>PID</i>
<i>data</i> : <i>DATA</i>
$\exists m$: <i>Message</i> •
<i>m.to</i> = <i>to</i>
<i>m.from</i> = <i>self</i>
<i>m.data</i> = <i>data</i>
<i>PutMessage</i>
<i>unchanged(self)</i>

<i>Receive</i>
Δ <i>DirectNaming</i>
<i>from</i> : <i>PID</i>
<i>data</i> : <i>DATA</i>
$\exists m$: <i>Message</i> •
<i>GetMessage</i>
<i>m.to</i> = <i>self</i>
<i>m.from</i> = <i>from</i>
<i>m.data</i> = <i>data</i>
<i>unchanged(self)</i>

The last operation we define for *DirectNaming* is perhaps surprising, as it defines a no-op on *DirectNaming*. It is needed to complete the set of operations, so that Processes may specify that one of their operations does not affect the message system, i.e. its effects are completely internal to the process.

<i>InternalOperation</i>
\equiv <i>DirectNaming</i>

5 Specifying the Algorithm

In this section, we deal with the specification of a particular algorithm. These specifications do not form part of the distributed system library but illustrate how an algorithm may be “plugged” into the library.

5.1 Message Types

We declare a schema for each message type. In this simple example, we need only one message type *Token* which contains a counter which (for no good reason at all) must hold an even value. We will also introduce a second message type *Tick* (which is not used in our example) to illustrate how a number of different message types are handled.

<i>Token</i>
<i>counter</i> : \mathbb{N}
<i>counter mod 2</i> = 0

<i>Tick</i>
<i>declarations</i>
<i>predicate</i>

Now that we know the contents of messages, the free type *DATA*, from section 3, can be defined as the disjoint union of these message schemas.

$$DATA \cong token \ll Token \gg \mid tick \ll Tick \gg$$

For those unfamiliar with \mathbb{Z} 's disjoint union, "token" is a one-to-one function from *Token* to *DATA*, and "tick" is similarly a one-to-one function from *Tick* to *DATA*. Furthermore, the ranges of the *token* and *tick* functions are disjoint. Therefore a *Token* converted into the type *DATA* can be unambiguously converted back to a *Token*, but never to a *Tick*, and vice versa.

5.2 Process

Process is the level at which we specify the particular distributed algorithm of interest. In this example, the algorithm involves processes passing a token around from one neighbour to the next in a circular fashion.

Since this algorithm is based on direct naming, it must include the *DirectNaming* schema. Each process records if it presently holds the token, and if so, what is the token's counter value.

<i>Process</i> <i>DirectNaming</i> <i>havetoken</i> : Boolean <i>t</i> : Token

The process that starts off with the token must assign to it some suitable initial value. Later we will see that only one such process exists, and hence the system contains only a single token.

<i>ProcessInitially</i> <i>Process</i> <i>DirectNamingInitially</i> <i>havetoken</i> \Rightarrow <i>T.counter</i> = 0

In our example, a process can do only three things. When the tokenholder, a process can send the token to its successor, or use the possession of the token to permit some mutually exclusive action (increment the counter by 2). When not the tokenholder, a process may receive the token from its predecessor.

We assume here that the functions "pred" and "succ" are suitably defined as circular decrement and increment on *PID*.

<i>PassToken</i> Δ <i>Process</i> <i>havetoken</i> Send[succPID(self)/to, token(t)/data] \neg <i>havetoken'</i>
--

The value of *t'* is deliberately omitted in *PassToken*, as its value is only important when the process holds the token.

<i>UseToken</i> Δ <i>Process</i> <i>havetoken</i> <i>t'.counter</i> = <i>t.counter</i> + 2 unchanged(<i>havetoken</i>) <i>InternalOperation</i>

GetToken $\Delta Process$ $\neg havetoken$ *Receive*[*prevPID*(*self*)/*from*, *token*(*t'*)/*data*]*havetoken'*

It is worth noting that all of the operations that a process can perform can be described by the disjunction of all the operations.

$$ProcessOperations \hat{=} PassToken \vee UseToken \vee GetToken$$

For those less familiar with \mathbb{Z} , this is equivalent to:

ProcessOperations $\Delta Process$ $(havetoken$ *Send*[*succPID*(*self*)/*to*.*token*(*t*)/*data*] $\neg havetoken')$ \vee $(havetoken$ *t'.counter* = *t.counter* + 2unchanged(*havetoken*)*InternalOperation*) \vee $(\neg havetoken$ *Receive*[*prevPID*(*self*)/*from*, *token*(*t'*)/*data*]*havetoken'*)

It is actually necessary to define *ProcessOperations*, the reason will become apparent in section 6.

5.3 Initializing the Algorithm

Although we have described the initialization of each individual process, we may need to describe the overall initialization of the algorithm or, more properly, the initialization of the set of processes which are using the algorithm.

In our example algorithm, we require that there be only one initial token holder (to ensure only one token in the system).

We use a framing schema to define the initial state of the processes. The following section defines the distributed system of processes which will use the algorithm and so $\Phi_{AlgorithmInitially}$ is written to be included as part of the systems initial state.

The system is defined to have only one initial token holder.

 $\Phi_{AlgorithmInitially}$ $\exists_1 p : procs \bullet$ *p.havetoken*

6 The Distributed System

The Distributed System encapsulates both the processes and the message system.

Here then is the distributed system defined for direct naming. It consists of a set of processes, each of which has a unique "self". It also ensures that all processes share the same message system.

DistributedSystem $procs : P \text{ Process}$

$$\begin{aligned} \#procs &= \#\{p : procs \bullet p.self\} \\ \forall p1, p2 : procs \bullet \\ & p1.enroute = p2.enroute \\ & p1.delivered = p2.delivered \end{aligned}$$

Note that the predicate depends on the specification of the Message Delivery System (section 3), so that if this is changed, the specification of the distributed system must be altered to match.

The initial state of the distributed system is simply the initial state of the algorithm ($\Phi \text{AlgorithmInitially}$). Additionally, every process in the system must have a correct initial state (ProcessInitially).

*DistributedSystemInitially**DistributedSystem*

$$\begin{aligned} & \Phi \text{AlgorithmInitially} \\ \forall p : procs \bullet \\ & p \in \text{ProcessInitially} \end{aligned}$$

The next step is to specify that the distributed system changes according to the operations defined on the processes and the act of message transfer from within the message system.

An operation performed by a process affects only that process. We use *ProcessOperations* (defined in section 5.2) to constrain the changes in a process to the defined operations on processes. Note that an operation in Z can be considered as a relation between a “before” and an “after” state. We use this view of an operation to express *DistributedSystemProcessOperation* in terms of *ProcessOperations*.

DistributedSystemProcessOperation $\Delta \text{DistributedSystem}$

$$\begin{aligned} \exists p, p' : \text{Process} \bullet \\ & p \in procs \\ & p \text{ProcessOperations} p' \\ & procs' = (procs - p) \cup \{p'\} \end{aligned}$$

A message transfer affects all processes indirectly (as they all “contain” the message system), but doesn’t affect any of the algorithmic state of the process.

DistributedSystem.MessageTransfer $\Delta \text{DistributedSystem}$

$$\begin{aligned} \exists delivered, delivered', enroute, enroute' : \text{MailBag} \bullet \\ & \text{MessageTransfer} \\ \forall p : procs \bullet \\ & p.delivered = routeelivered \\ & p.enroute = enr \\ \exists p' : procs' \bullet \\ & p \setminus \{delivered, enroute\} = p' \setminus \{delivered, enroute\} \\ & p'.delivered = delivered' \\ & p'.enroute = enroutel' \end{aligned}$$

7 The Module “Interfaces”

There are essentially two interfaces between the specification of an algorithm and the system that surrounds it:

1. The underlying specifications (e.g. the Message Delivery System and Direct Naming) form an interface which defines the facilities available to the algorithm.
2. The specification of the algorithm forms an interface which is used by the distributed system. The specification of the system requires a number of definitions to be made in the specification of a process which implements the algorithm.

In the example given, the process must inherit *DirectNaming* and include *DirectNamingInitially* in the specification of its initial state. The operations *Send*, *Receive* and *InternalOperation* are provided to define interactions with the message delivery system.

In turn, the specification of the process must define the state of the process in the schema *Process*, its initial state with *ProcessInitially* and construct *ProcessOperations* as the disjunction of its operations. Additionally, it must define the type *DATA* to be equal to the disjoint union of the message types. Any constraints on the set of processes are specified in the framing schema Φ *AlgorithmInitially*, bearing in mind that it is intended for inclusion in *DistributedSystemInitially*.

8 Conclusions

The reusable specification we have presented defines a distributed system of processes which communicate using a one-to-one direct-naming style of communication through some non-instantaneous medium. The specification is deliberately modular in order to reduce rewriting in future specifications.

A simple example algorithm is presented in order to demonstrate that the modules are sufficient to specify a distributed system of processes.

We intend that this document should form the beginning of a growing library of specification modules. Eventually, when specifying a new distributed algorithm, it should be possible to focus almost entirely on the details of the algorithm instead of on the surrounding system, as is currently the case.

References

- [1] J.-R. Abrial, S. Schnuman, and B. Meyer. Specification language. In R. McKeog and A. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343-410. Cambridge University Press, 1980.