Chapter 22

Case study: A mail system

This final case study contains little in the way of intricate algorithm development, and not a single iterative invariant: it is a case study in specification itself.

In our earlier examples we did not have to worry about the construction of the specification: it was given at the outset, usually of a program fragment or a single procedure. In this chapter however we concern ourselves with the specification of a whole 'system', which in our terms is a module encapsulating a state and all operations on it. The increased size and possible complexity means that we cannot take even our starting point for granted.

Electronic mail systems, our subject, have their main features in common but vary a lot in the detail. Thus our initial specification, in Figure 22.1, is more or less just the bare minimum that electronic mail could be said to comprise. But the immediately following two sections nevertheless discuss deficiencies apparent even at that level of abstraction, and propose design changes to avoid them.

Design changes are seldom without hazard, and several 'opportunities' arise on our way to Figure 22.10, the 'final' specification, for changes that if implemented would sooner or later prove disastrous. The dangers are revealed by exhaustive — and sometimes exhausting — application of the rigorous techniques now at our disposal: assumptions, coercions, refinement and above all data refinement.

The message of those sections is twofold: that even the apparently simple Figure 22.10 is too complex a specification to accept without some degree of rigorous analysis; and that one way of producing even a simple specification is to develop it in small steps from one that is simpler still.

The second part of the chapter makes the first few development steps, from Figure 22.10 towards a conventional implementation of electronic mail in terms of asynchronous delivery of message packets. The concurrency involved is limited, but still enough to capture the notion of system actions carried out 'in the background' without the participation of the users.

The result of the development, delivered in Figure 22.15, is laden with the details of headers, unique identifiers, packets and transmission delay — a very long way from the simple beginnings to which shortly we turn.

A warning is appropriate before we start: much of the low-level working is detailed enough to skip at first reading, and just the beginning and end of 'refinement chains' may be sufficient in many cases to maintain the continuity of the narrative.

22.1 A first specification

The system will provide just three procedures for passing messages:

- *Send*: A message and set of intended recipients are supplied; a 'unique' identifier is returned for that transmission. The identifier is used to refer to the message while it remains in the system.
- *Receive*: The set of identifiers is returned for mail that has been received by a given user, but not yet read.
- *Read*: The message text corresponding to a given identifier is presented to its recipient, and the message is removed from the system.

Generally the system works as follows. A user *me* sends a message *msg* to a group *tos* of other users using Procedure *Send*; its result parameter *id* provides a unique reference to that transmission, which could be used by the sender, for example, to enquire after the status of a message or even to cancel it. (See Exercises 22.1 and 22.2.)

Procedure *Receive* is called by potential recipients who wish to know whether mail has been sent to them. Its result is a set *ids* of transmission identifiers that refer to messages sent to them that they have not yet read.

Supplying an identifier to Procedure *Read* will return the message text associated with the identifier, and delete that message from the system.

Naturally the above brief description leaves a lot unsaid. Can a user send a message to himself? What happens for example if an invalid identifier is supplied to *Read*? Can a message be read by a user to whom it was *not* sent? Does the deleting of a message, once read in *Read*, remove copies of it sent to other recipients?

It is extremely unlikely that an informal specification like the one above, no matter how extensive, could ever answer all such questions. Even rigorous specifications, like the one we shall see shortly, are limited. But they do have a significant advantage: their terms of reference are unambiguous, and within their declared terms their answers are unambiguous[1] as well.

To construct an abstract module consistent with the informal description above — our version of 'rigorous specification' here — we begin by choosing types. Let users come from a type *Usr*, messages from *Msg* and identifiers from *Id*. The state of the system could then be given by the two variables

[1] We regard nondeterminism in a specification as an unambiguous indication of freedom in the design.

$$msgs : Id \nrightarrow Msg \quad \text{and} \quad sent : Usr \leftrightarrow Id \ ,$$

in which the partial function *msgs* associates message identifiers with the corresponding message text, and the relation *sent* records the (identifiers of) messages sent but not yet read. By making *sent* a proper relation we allow many users (many possible recipients) to be associated with a single identifier: that is how we shall deal with 'broadcasts', in which the same message is sent to many different users.

For the procedure *Send* we have the following parameters: which user is sending the message; the message itself; to which users it is being sent; and (as a result parameter) the identifier that subsequently can be used to refer to it. Here is the procedure heading:

$$\begin{array}{l} \textbf{procedure}\ Send\ (\textbf{value}\ me : Usr;\ \ msg : Msg;\ \ tos : \textbf{set}\ Usr; \\ \qquad\qquad\qquad\quad \textbf{result}\ id : Id)\ . \end{array}$$

Within the procedure we first have a new identifier selected, which then will be associated both with the message and with the set of recipients. We use simply $i\colon [i \notin \mathsf{dom}\, msgs]$ for the selection, on the understanding that Id must be an infinite set so that the supply of unused identifiers can never 'run out'. That does raise several problems — but its simplicity is so appealing that we shall do it nevertheless.

The first problem is that the above specification is infeasible: if executed in a state in which $\mathsf{dom}\, msgs = Id$ (meaning 'all identifiers in use'), its postcondition is false. Informally we note however that if Id is infinite the module can never reach such a state (provided *msgs* is initially empty); more rigorously, we could argue as in Exercise 17.14, showing that $\mathsf{dom}\, msgs \in \textbf{finset}\, Id$ is invariant for the module, and by data refinement therefore being able to introduce $\mathsf{dom}\, msgs \subset Id$ into the precondition.

The second problem, however, is that we cannot after all provide an infinite type Id: the identifiers may have to be recovered.[2] But we will see much about that later, and so for now leave things as they are.

Once the identifier is selected it is a straightforward matter to construct the neccessary links to the recipients and the message text, and the body of the procedure is as a whole

$$\begin{array}{l} id\colon [id \notin \mathsf{dom}\, msgs]\,; \\ msgs[id] := msg; \\ sent := sent \cup (tos \times \{id\})\ . \end{array}$$

Note that the assignment $msgs[id] := msg$ is just an abbreviation for the lengthier $msg := msg[id := msg]$, and in particular that id need not be in the domain of

[2] It is not altogether certain that they must, since for example using 64-bit unique identifiers would be sufficient to support a traffic of one message per millisecond for more than 500 million years. But other resource implications may make recovery of identifiers appropriate nevertheless.

msgs for that to be meaningful. (See Exercise 9.14; in fact the previous command ensures that *id* is *not* in the domain of *msgs*.)

Note also that in this simple system we are not bothering to record the sender of the message.

For the procedure *Receive* we need only return for user *me* the set of identifiers of messages waiting in *sent* to be read, and for that we use $ids := sent[me]$. (Ignoring for now the warning in Section 9.6.2, that unlikely-looking application $sent[me]$ of a relation to an element we take as an abbreviation for $\{i : Id \mid (me, i) \in sent\}$.)

Finally, procedure *Read* must retrieve a message, given its identifier, and here we deal with some of the questions raised earlier. Suppose user *me* supplies identifier *id* legitimately — that is, that (me, id) is an element of *sent*, meaning that *me* is one of its intended recipients: then the message to be returned is found in *msgs*, and $msg := msgs[id]$ will retrieve it. But if the identifier *id* is not legitimate for *me*, what then? Making 'legitimacy' a precondition of the procedure (we need only include the assumption $\{(me, id) \in sent\}$ as its first command) would relieve the implementor of the obligation to deal with such matters: the procedure would simply abort if the request were not legitimate.

A more forgiving design would insist on termination in any case (not aborting, therefore); but it would allow any message whatsoever to be returned for illegitimate requests. That ranges from the helpful 'Identifier does not refer to a message that has been received.' through the cryptic '`MSGERR BAD ID`', finally to the mischievous option of returning likely-looking but wholly invented messages that were never sent. (That last could be useful if one user is suspected of trying to read messages intended for others.) Thus we use the specification

$$msg\colon [(me, id) \in sent \Rightarrow msg = msgs[id]] \ ,$$

leaving as the last detail the removal of the message from the system. The command $sent := sent - \{(me, id)\}$ does that, with the set subtraction as usual having no effect on *sent* if the pair $\{(me, id)\}$ is not there.

Straightforward initialisation to 'the system is empty' gives us finally the module of Figure 22.1.

22.2 Reuse of identifiers

We considered briefly above the possible implementation problems caused by our use of $id\colon [id \notin \mathsf{dom}\, msgs]$ in *Send*: that an unending supply of identifiers is required. Looking at *Read* in Figure 22.1, however, we can see that once a message has been read by all of its intended recipients, the antecedent $(me, id) \in sent \Rightarrow \cdots$ of the first command will never again be true, and so subsequent calls for the same identifier need not refer to *msgs* — the message texts can simply be invented. Thus, provided that at the end of *Read* we have $id \notin \mathsf{ran}\, sent$ (because the last (me, id) pair has just been removed), we can remove (id, msg) from *msgs*. That is

module *MailSys*
var $msgs : Id \nrightarrow Msg;$
$sent : Usr \leftrightarrow Id\bullet$

procedure *Send* (**value** $me : Usr;\ msg : Msg;\ tos :$ **set** $Usr;$
result $id : Id$)
$\widehat{=}\ id\colon [id \notin \mathsf{dom}\, msgs]\,;$
$msgs[id] := msg;$
$sent := sent \cup (tos \times \{id\});$

procedure *Receive* (**value** $me : Usr;$ **result** $ids :$ **set** Id)
$\widehat{=}\ ids := sent[me];$

procedure *Read* (**value** $me : Usr;\ id : Id;$ **result** $msg : Msg$)
$\widehat{=}\ msg\colon [(me, id) \in sent \Rightarrow msg = msgs[id]]\,;$
$sent := sent - \{(me, id)\};$

initially $msgs = sent = \{\}$
end

Figure 22.1 Initial specification of mail system

procedure *Read* (**value** $me : Usr;\ id : Id;$ **result** $msg : Msg$)
$\widehat{=}\ msg\colon [(me, id) \in sent \Rightarrow msg = msgs[id]]\,;$
$sent := sent - \{(me, id)\};$
$msgs := (\mathsf{ran}\, sent) \lhd msgs$ $\quad\lhd$

Figure 22.2 Attempted recovery of *Id*'s

done with the $\lhd$-marked command in the revised *Read* shown in Figure 22.2, which removes all such pairs at once.

But on what basis have we been saying 'should' and 'can'? Are we changing the specification, or are we merely refining the original module of Figure 22.1? How do we find out?

The effect of the change is to make the state component *msgs* a smaller function than before, taking care however never to delete identifiers still in $\mathsf{ran}\, sent$. We are therefore led to consider a data refinement in which the abstract variable is *msgs*, the concrete is $msgs'$, say, and the coupling invariant is

$$msgs' = (\mathsf{ran}\, sent) \lhd msgs \ . \tag{22.1}$$

(In all of the data refinements of this chapter we shall use 'primed' names for concrete variables; when the result of the data refinement is presented (and thus the abstract variables are gone), we simply remove the primes. That will help prevent a proliferation of names.)

We begin our data refinement with the last command of *Read*:

$$
\begin{array}{ll}
& sent := sent - \{(me, id)\} \\
\multicolumn{2}{l}{\textit{becomes}\ \text{“}\textit{augment assignment}\ 17.8\text{”}} \\
& \{msgs' = (\mathsf{ran}\, sent) \lhd msgs\}; \\
& sent, msgs' := sent - \{(me, id)\}, ?; \\
& [msgs' = (\mathsf{ran}\, sent) \lhd msgs] \\
\sqsubseteq & \text{“see below”} \\
& \{msgs' = (\mathsf{ran}\, sent) \lhd msgs\}; \\
& sent := sent - \{(me, id)\}; \\
& [(\mathsf{ran}\, sent) \lhd msgs' = (\mathsf{ran}\, sent) \lhd msgs]\ ; \\
& msgs' := (\mathsf{ran}\, sent) \lhd msgs' \\
\sqsubseteq & \text{“see below”} \\
& \{msgs' = (\mathsf{ran}\, sent) \lhd msgs\}; \\
& \left[\begin{array}{ll} & (\mathsf{ran}(sent - \{(me, id)\})) \lhd msgs' \\ = & (\mathsf{ran}(sent - \{(me, id)\})) \lhd msgs \end{array} \right]; \\
& sent := sent - \{(me, id)\}; \\
& msgs' := (\mathsf{ran}\, sent) \lhd msgs' \\
\sqsubseteq & \text{“}\textit{remove coercion}\ 17.20\ \text{mainly”} \\
& sent := sent - \{(me, id)\}; \\
& msgs' := (\mathsf{ran}\, sent) \lhd msgs' \ .
\end{array}
$$

Thus $msgs' := (\mathsf{ran}\, sent) \lhd msgs'$ has appeared, as desired.

The comments 'see below' refer to the first of the following two laws, used for moving coercions 'forward' through a program:

Law 22.1 <u>advance coercion</u>

$$w := E\ [post] \quad = \quad [post[w \backslash E]]\ w := E \ .$$

□

Law 22.2 <u>advance assumption</u>

$$w := E\ \{pre\} \quad = \quad \{pre[w \backslash E]\}\ w := E \ .$$

□

Law 22.1 is often used to move a coercion forward through a program until it is 'cancelled' by an earlier, and weaker, assumption (as in *remove coercion* 17.20, given that it is a refinement to weaken an assumption).

For the rest of the module, things are straightforward until finally we reach the first command of *Send*. There, we have

$$
\begin{array}{ll}
& id\colon [id \notin \mathsf{dom}\, msgs] \\
\textit{becomes} & \text{“\textit{augment specification} 17.6”} \\
& \left[\begin{array}{c} id, msgs' \\ \hline msgs' = (\mathsf{ran}\, sent) \lhd msgs \\ \hline msgs' = (\mathsf{ran}\, sent) \lhd msgs \\ id \notin \mathsf{dom}\, msgs \end{array}\right] \\
= & \text{“}msgs,\ sent \text{ not in frame, thus } msgs' \text{ cannot change”} \\
& id\colon [msgs' = (\mathsf{ran}\, sent) \lhd msgs\ ,\ id \notin \mathsf{dom}\, msgs] \\
\textit{becomes} & \text{“\textit{diminish specification} 17.12”} \\
& \left[\begin{array}{c} id \\ \hline (\exists\, msgs \cdot msgs' = (\mathsf{ran}\, sent) \lhd msgs) \\ \hline (\forall\, msgs_0 \cdot msgs' = (\mathsf{ran}\, sent) \lhd msgs_0 \Rightarrow id \notin \mathsf{dom}\, msgs_0) \end{array}\right] \\
= & \text{“simplify precondition”} \\
& \left[\begin{array}{c} id \\ \hline \mathsf{dom}\, msgs' \subseteq \mathsf{ran}\, sent \\ \hline (\forall\, msgs_0 \cdot msgs' = (\mathsf{ran}\, sent) \lhd msgs_0 \Rightarrow id \notin \mathsf{dom}\, msgs_0) \end{array}\right] \\
= & \text{“simplify postcondition”}^{3} \\
& id\colon [\mathsf{dom}\, msgs' \subseteq \mathsf{ran}\, sent\ ,\ id \in \mathsf{ran}\, sent - \mathsf{dom}\, msgs']\ .
\end{array}
$$

We have been reasoning with equality rather than refinement $\sqsubseteq$, because we want to be sure of finding a concrete command if there is one. (Using $\sqsubseteq$ we might accidentally introduce infeasible behaviour and thus miss a data refinement that would actually have been acceptable.)

In the last step, refinement $\sqsubseteq$ is not difficult to show. (The antecedent gives $\mathsf{dom}\, msgs' = \mathsf{ran}\, sent \cap \mathsf{dom}\, msgs_0$, and thus that $\mathsf{ran}\, sent - \mathsf{dom}\, msgs'$ and $\mathsf{dom}\, msgs_0$ are disjoint.) But for the equality we need also the reverse refinement $\sqsupseteq$; for that we choose an arbitrary $m : Msg$ and define $large \mathrel{\widehat{=}} Id \times \{m\}$, whose domain is all of Id. Taking $msgs_0$ to be $msgs' \cup (\mathsf{ran}\, sent) \lhd\!\!\!- large$ satisfies the antecedent, and the consequent is then equivalent to $id \in \mathsf{ran}\, sent - \mathsf{dom}\, msgs'$.

But alas it has all in any case been for nothing, since our conclusion is *not* feasible: the precondition allows $\mathsf{dom}\, msgs' = \mathsf{ran}\, sent$, making the postcondition false. Not having shown refinement with this particular coupling invariant does not of course mean that no other would work; but it does encourage us to look more closely at whether we *are* after all performing refinement.

[3]Some of these comments conceal quite a lot of non-trivial predicate calculation, in this case discussed below. Similarly, 'routine' steps in engineering design sometimes generate quite tough integrals to be calculated. But the *principles* remain simple.

Persistent problems with electronic funds transfer led to chaos recently in the financial markets. The cause was traced to code in which the ordinary electronic mail system had been used to generate the unique identifiers needed for funds transfers.

Noticing that the specification of the mail system guaranteed never to repeat an identifier, a programmer had obtained them as needed by broadcasting null messages to no-one. (Such 'empty broadcasts' were, not surprisingly, particularly efficient and generated no network traffic.)

The mail system originally was implemented with such a large set of possible identifiers it was thought they would never run out. Use grew so rapidly, however, that recently the system was upgraded to recover old identifiers — yet it was not verified that the new system was a refinement of the original, and in fact it was not. Had the absence of refinement been noticed, the well-established principles of the Institute of Systems and Software Engineering would then have required a routine check to be made for dependencies on the original behaviour.

The institutions affected are suing for damages; meanwhile the financial community waits anxiously for other effects to come to light.

Figure 22.3

In fact we are not proposing a refinement: the concrete module can return the same *id* from *Send* on separate occasions, which is something the abstract module cannot do. But does it really matter? See Figure 22.3 (and Exercise 22.3).

$$
\begin{array}{l}
\textbf{module } \mathit{MailSys} \\
\quad \textbf{var } \mathit{msgs} : \mathit{Id} \nrightarrow \mathit{Msg}; \\
\quad\quad \mathit{sent} : \mathit{Usr} \leftrightarrow \mathit{Id} \bullet \\
\\
\quad \textbf{procedure } \mathit{Send}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \mathit{msg} : \mathit{Msg};\ \mathit{tos} : \textbf{set } \mathit{Usr}; \\
\quad\quad\quad\quad\quad\quad \textbf{result } \mathit{id} : \mathit{Id}) \\
\quad \widehat{=}\ \ \mathit{id}\colon [\mathit{id} \notin \mathsf{ran}\, \mathit{sent}]\ ; \qquad\qquad\qquad\qquad\qquad\qquad \triangleleft \\
\quad\quad \mathit{msgs}[\mathit{id}] := \mathit{msg}; \\
\quad\quad \mathit{sent} := \mathit{sent} \cup (\mathit{tos} \times \{\mathit{id}\}); \\
\\
\quad \textbf{procedure } \mathit{Receive}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \textbf{result } \mathit{ids} : \textbf{set } \mathit{Id}) \\
\quad \widehat{=}\ \ \mathit{ids} := \mathit{sent}[\mathit{me}]; \\
\\
\quad \textbf{procedure } \mathit{Read}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \mathit{id} : \mathit{Id};\ \textbf{result } \mathit{msg} : \mathit{Msg}) \\
\quad \widehat{=}\ \ \mathit{msg}\colon [(\mathit{me}, \mathit{id}) \in \mathit{sent} \Rightarrow \mathit{msg} = \mathit{msgs}[\mathit{id}]]\ ; \\
\quad\quad \mathit{sent} := \mathit{sent} - \{(\mathit{me}, \mathit{id})\}; \\
\\
\quad \textbf{initially } \mathit{msgs} = \mathit{sent} = \{\} \\
\textbf{end}
\end{array}
$$

Figure 22.4 Reuse of identifiers

22.3 A second specification: reuse

We are forced to admit that reusing identifiers requires a *change* in the specification that is *not* a refinement of it. Having to accept therefore that we are still in the 'design stage', we consider a simpler change with the same effect: we leave *Read* in its original state, changing *Send* instead so that 'new' identifiers are chosen in fact simply outside the range of *sent* (since it is precisely the identifiers in *sent* that refer to messages not yet read by all recipients). The result is shown in Figure 22.4, with the altered command marked.

A slightly unhelpful aspect of this new specification, and of the earlier attempt, now comes to light: it is that the reuse of identifiers is enabled by *Read* as soon as (me, id) is removed from *sent*. In an eventual implementation that would probably require communication, in some form, from the receiver back to the sender. Our first specification did not require that, since the generation of identifiers was managed locally in *Send*, an essentially self-contained activity.

A second unrealistic aspect of this specification is that messages arrive instantly at the destination: a *Receive* no matter how quickly after a *Send* will return the identifier of the newly sent message, and this too is unlikely to be implementable in practice.

procedure *Receive* (**value** $me : Usr$; **result** ids : **set** Id)
$\mathrel{\widehat{=}}$ ids: $[ids \subseteq sent[me]]$ $\lhd$

Figure 22.5 Attempt at specifying delayed receipt

Thus we are led to consider a third version of our specification.

22.4 A third specification: delay

In order to allow delay between sending a message and receiving it, one might think of altering *Receive* as shown in Figure 22.5: only some, not necessarily all, of the identifiers of sent messages are returned by *Receive*. The rest are 'in transit'.

But if the subset returned is chosen afresh on each occasion, then messages could be received only later to be 'unreceived' again. In order to specify that once a message is received it stays received, we must introduce an extra variable *recd* that records which messages have been received already. That would be necessary in any case to make *Read* sensitive to whether a message has been received or not.

Thus while *Send* will use *sent* as before, in *Read* we find the new variable *recd* instead. The transfer of messages between *sent* and *recd* occurs in *Receive*, as shown in Figure 22.6. Note that in *Read* both *sent* and *recd* must have (me, id) removed: if left in *recd* the message could be read again; if removed from *recd* but left in *sent* it could be received again; and if left in both its identifier would never be recovered.

The effect of the marked command in *Receive* is to allow *recd* to increase arbitrarily up to and including *sent*. (That includes not increasing at all.) As written, however, the command is infeasible: one cannot achieve $recd_0 \subseteq recd \subseteq sent$ while changing only *recd*, unless $recd \subseteq sent$ already. Fortunately $recd \subseteq sent$ is an invariant of the module: as before, a data refinement could introduce it explicitly, allowing the command to be replaced by

$$recd\colon [recd \subseteq sent\ ,\ recd_0 \subseteq recd \subseteq sent]\ .$$

In the interests of brevity, however, we leave it as it is.

We will not attempt to show that Figure 22.6 refines our earlier Figure 22.4: indeed it cannot, because with our new module the program fragment

$$Send\ (me, msg, \{you\}, id);\ \ Receive\ (you, ids)$$

can terminate with $id \notin ids$ (because id is still in transit), and in our earlier module that is not possible. Nevertheless we should investigate carefully what we have done: is delay the *only* change we have made?

module *MailSys*
  var $msgs : Id \nrightarrow Msg$;
    $sent, recd : Usr \leftrightarrow Id \bullet$

  procedure *Send* (**value** $me : Usr$; $msg : Msg$; $tos :$ **set** Usr;
      result $id : Id$)
  $\widehat{=}$ id: $[id \notin \mathsf{ran}\, sent]$;
    $msgs[id] := msg$;
    $sent := sent \cup (tos \times \{id\})$;

  procedure *Receive* (**value** $me : Usr$; **result** $ids :$ **set** Id)
  $\widehat{=}$ $recd$: $[recd_0 \subseteq recd \subseteq sent]$; $\triangleleft$
    $ids := recd[me]$;

  procedure *Read* (**value** $me : Usr$; $id : Id$; **result** $msg : Msg$)
  $\widehat{=}$ msg: $[(me, id) \in recd \Rightarrow msg = msgs[id]]$;
    $sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}$;

  initially $msgs = sent = recd = \{\}$
end

Figure 22.6 Delayed receipt of messages

For our investigation, we go back and alter (but do not necessarily *refine*) our 'prompt' module of Figure 22.4 to express our minimum expectations of introducing delay. First, we must accept that *Receive* will not return *all* identifiers of messages sent, and so we use in this 'mock-up' the alternative procedure in Figure 22.5. Second, we split *Read* into two procedures: one for reading received messages, and the other for reading (or attempting to read) not-yet-received ones. The former should behave as *Read* does in Figure 22.4; the latter should return a randomly chosen message, but change nothing else. The result is shown in Figure 22.7.

We should be quite clear about the role of Figure 22.7: it is not a refinement of Figure 22.4 (a customer having specified a prompt mail system will not accept an implementation containing delay); nor is it even a satisfactory specification of a system with delay (it is too weak). It is only the most we can say about delay while retaining the state of Figure 22.4.

Because we constructed our system with delay (Figure 22.6) essentially by guesswork, we are now double-checking against Figure 22.7 to see whether it has those 'reasonable' properties *at least*.

To compare Figure 22.6 with Figure 22.7, we must make the same distinction in Figure 22.6 between reading received messages and attempting to read not-yet-

module *MailSys*
    var $msgs : Id \nrightarrow Msg;$
        $sent : Usr \leftrightarrow Id \bullet$

    procedure *Send* (**value** $me : Usr;\ msg : Msg;\ tos :$ **set** $Usr;$
                        result $id : Id)$
    $\widehat{=}\ id\colon [id \notin \mathsf{ran}\, sent]\ ;$
        $msgs[id] := msg;$
        $sent := sent \cup (tos \times \{id\});$

    procedure *Receive* (**value** $me : Usr;$ **result** $ids :$ **set** $Id)$
    $\widehat{=}\ ids\colon [ids \subseteq sent[me]]\ ;$

    procedure *ReadReceived* (**value** $me : Usr;\ id : Id;$
                                result $msg : Msg)$
    $\widehat{=}\ msg\colon [(me, id) \in sent \Rightarrow msg = msgs[id]]\ ;$
        $sent := sent - \{(me, id)\};$

    procedure *ReadNotReceived* (**value** $me : Usr;\ id : Id;$
                                    result $msg : Msg)$
    $\widehat{=}$ **choose** $msg;$

    initially $msgs = sent = \{\}$
end

Figure 22.7 Delay ‘mock-up’ — compare Figure 22.4

received ones. We can do that with a pair of coercions.

Recall that a coercion $[post]$ behaves like **skip** if *post* holds, and like **magic** otherwise: if *post* does not hold then $[post]$ is essentially ‘unexecutable’. We make our procedures *ReadReceived* and *ReadNotReceived* from Figure 22.6 by exploiting that unexecutability. The body of *ReadReceived* will be as for *Read* but with an initial coercion expressing ‘the message has been received’:

$$\begin{array}{l} [(me, id) \in recd]\ ; \\ msg\colon [(me, id) \in recd \Rightarrow msg = msgs[id]]\ ; \\ sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}\ . \end{array}$$

Naturally, we can use the coercion to simplify the rest of the procedure; we continue from immediately above:

$=$ “*introduce assumption* 17.19”

$$
\begin{array}{ll}
& [(me, id) \in recd]\,; \\
& \{(me, id) \in recd\}; \\
& msg\colon [(me, id) \in recd \Rightarrow msg = msgs[id]]\,; \\
& sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\} \\
= & \text{“}\textit{absorb assumption } 1.8\text{”} \\
& [(me, id) \in recd]\,; \\
& msg\colon [(me, id) \in recd\ ,\ (me, id) \in recd \Rightarrow msg = msgs[id]]\,; \\
& sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\} \\
= & \text{“Note this is equality”}^{4} \\
& [(me, id) \in recd]\,; \\
& msg := msgs[id]; \\
& sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}\ .
\end{array}
$$

If disinclined to work through the above, one could simply note that the coercion $(me, id) \in recd$ simplifies the following postcondition to $msg = msgs[id]$.

The body of *ReadNotReceived* will have the opposite coercion added, and we are able to simplify it as follows: the coercion can by *introduce assumption* 17.19 spawn an assumption which can then be distributed throughout the procedure body, exploiting the fact that almost all commands there change none of its variables. We have

$$
\begin{array}{ll}
& [(me, id) \notin recd]\,; \\
& msg\colon [(me, id) \in recd \Rightarrow msg = msgs[id]]\,; \\
& sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\} \\
= & [(me, id) \notin recd]\,; \\
& \{(me, id) \notin recd\}; \\
& msg\colon [(me, id) \in recd \Rightarrow msg = msgs[id]]\,; \\
& \{(me, id) \notin recd\}; \\
& sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\} \\
= & \text{“}\left\{\begin{array}{l} \textit{remove assumption } 1.10 \\ \textit{absorb assumption } 1.8 \\ \textit{leading assignment } 8.5 \end{array}\right.\quad\text{”} \\
& [(me, id) \notin recd]\,; \\
& msg\colon [(me, id) \notin recd\ ,\ (me, id) \in recd \Rightarrow msg = msgs[id]]\,; \\
& sent := sent - \{(me, id)\}; \\
& recd\colon [(me, id) \notin recd\ ,\ recd = recd_0 - \{(me, id)\}] \\
\sqsubseteq & [(me, id) \notin recd]\,; \\
& msg\colon [\mathsf{true}]\,; \\
& sent := sent - \{(me, id)\}; \\
& \mathbf{skip} \\
\sqsubseteq & [(me, id) \notin recd]\,; \\
& \mathbf{choose}\ msg; \\
& sent := sent - \{(me, id)\}\ .
\end{array}
$$

[4]For simple refinement, rather than equality, the precondition would not have been necessary.

module *MailSys*
  var $msgs : Id \nrightarrow Msg$;
    $sent, recd : Usr \leftrightarrow Id \bullet$

  procedure *Send* (**value** $me : Usr$; $msg : Msg$; $tos :$ **set** Usr;
            result $id : Id$)
  $\mathrel{\widehat=}$ id: $[id \notin \mathsf{ran}\, sent]$;
    $msgs[id] := msg$;
    $sent := sent \cup (tos \times \{id\})$;

  procedure *Receive* (**value** $me : Usr$; **result** $ids :$ **set** Id)
  $\mathrel{\widehat=}$ $recd$: $[recd_0 \subseteq recd \subseteq sent]$;
    $ids := recd[me]$;

  procedure *ReadReceived* (**value** $me : Usr$; $id : Id$;
            result $msg : Msg$)
  $\mathrel{\widehat=}$ $[(me, id) \in recd]$;
    $msg := msgs[id]$;
    $sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}$;

  procedure *ReadNotReceived* (**value** $me : Usr$; $id : Id$;
            result $msg : Msg$)
  $\mathrel{\widehat=}$ $[(me, id) \notin recd]$;
    choose msg;
    $sent := sent - \{(me, id)\}$;

  initially $msgs = sent = recd = \{\}$
end

Figure 22.8 Delayed receipt of messages, with 'split' *Read*

The result of all of these changes is shown in Figure 22.8, and we now — finally — investigate whether Figure 22.7 is refined by Figure 22.8. We choose as coupling invariant $recd \subseteq sent$, with $recd$ being our concrete variable; we have no abstract variable.

Procedure *Send* we can look at informally: imagine that abstract and concrete states are coupled as above, and that we execute the abstract *Send* on the abstract state and the concrete *Send* on the concrete state. Of the variables appearing in the coupling invariant, only *sent* is modified, having $tos \times \{id\}$ added to it in both cases. As *sent* is therefore not made smaller (actually it is made strictly bigger, but that follows from earlier statements and we do not need it), the coupling invariant is maintained. We need also check that if the concrete *Send* can abort then so can the abstract (but neither can), and that all concrete results (*id* in this case) are possible also for the abstract: they are, as the choice of *id* depends only on *sent*, unaffected by the data refinement.

For procedure *Receive* we reason

$$ids\colon [ids \subseteq sent[me]]$$

becomes

$$\begin{array}{ll}
 & ids, recd\colon [recd \subseteq sent\ ,\ recd \subseteq sent \wedge ids \subseteq sent[me]] \\
\sqsubseteq & \text{“}\textit{following assignment } 3.5\text{”} \\
 & recd\colon [recd \subseteq sent\ ,\ recd \subseteq sent \wedge recd[me] \subseteq sent[me]]\,; \qquad \triangleleft \\
 & ids := recd[me] \\
\sqsubseteq & recd\colon [recd_0 \subseteq recd \subseteq sent]\ .
\end{array}$$

Procedure *ReadReceived* has two commands; for the first we proceed

$$msg\colon [(me, id) \in sent \Rightarrow msg = msgs[id]]$$

becomes

$$\begin{array}{ll}
 & \left[\begin{array}{c} msg, recd \\ \hline recd \subseteq sent \\ \hline recd \subseteq sent \\ (me, id) \in sent \Rightarrow msg = msgs[id] \end{array}\right] \\
\sqsubseteq & \text{“}\textit{assignment } 5.2\text{”} \\
 & msg := msgs[id] \\
\sqsubseteq & \text{“}\textit{introduce coercion } 17.3\text{”} \\
 & [(me, id) \in recd]\,; \\
 & msg := msgs[id]\ ,
\end{array}$$

and for the second command we have

$$sent := sent - \{(me, id)\}$$

becomes “*augment assignment* 17.7”

$$\begin{array}{ll}
 & \{recd \subseteq sent\}; \\
 & sent, recd := sent - \{(me, id)\}, ?; \\
 & [recd \subseteq sent] \\
\sqsubseteq & \text{“}\textit{advance coercion } 22.1\text{”} \\
 & \{recd \subseteq sent\}; \\
 & [recd - \{(me, id)\} \subseteq sent - \{(me, id)\}]\,; \\
 & sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\} \\
\sqsubseteq & \text{“}\left\{\begin{array}{l} \text{weaken assumption} \\ \textit{remove coercion } 17.20 \\ \textit{remove assumption } 1.10 \end{array}\right. \text{”} \\
 & sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}\ .
\end{array}$$

Finally we deal with *ReadNotReceived*:

$$
\begin{array}{l}
\quad \textbf{choose } msg \\
\textit{becomes} \\
\quad \{recd \subseteq sent\}; \\
\quad msg, recd :=?, ?; \\
\quad [recd \subseteq sent] \ .
\end{array}
$$

And here we have a problem. Our target code, in *ReadNotReceived* of Figure 22.8, appears to alter *sent*; the above code does not. Doing our best to aim for the code we want, we introduce the beginnings of our assignment to *sent* and continue:

$$
\begin{array}{ll}
= & \text{“}sent\text{: } [sent = sent_0] = \textbf{skip}\text{”} \\
& \{recd \subseteq sent\}; \\
& msg, recd :=?, ?; \\
& sent\text{: } [sent = sent_0] \, ; \\
& [recd \subseteq sent] \\
= & \text{“}\textit{following assignment } 3.5\text{”} \\
& \{recd \subseteq sent\}; \\
& msg, recd :=?, ?; \\
& [sent - \{(me, id)\} = sent] \, ; \\
& sent := sent - \{(me, id)\}; \\
& [recd \subseteq sent] \\
= & \text{“}\textit{advance coercion } 22.1\text{”} \\
& \{recd \subseteq sent\}; \\
& [(me, id) \notin sent] \, ; \\
& msg, recd :=?, ?; \\
& [recd \subseteq sent - \{(me, id)\}] \, ; \\
& sent := sent - \{(me, id)\} \\
\sqsubseteq & \text{“because } recd :=? \ \sqsubseteq \ \textbf{skip}\text{”} \\
& \{recd \subseteq sent\}; \\
& [(me, id) \notin sent] \, ; \\
& [recd \subseteq sent - \{(me, id)\}] \, ; \\
& \textbf{choose } msg; \\
& sent := sent - \{(me, id)\} \\
\sqsubseteq & [(me, id) \notin sent] \, ; \\
& \textbf{choose } msg; \\
& sent := sent - \{(me, id)\} \ .
\end{array}
$$

The last step of removing the assertion loses us no generality, since *recd* no longer appears in the following code. The earlier refinement of $recd :=?$ to **skip** was forced by the fact that *recd* is unchanged by the code of Figure 22.8.

The coercion $[(me, id) \notin sent]$, however, is not the one we want. It is too strong, and we can do nothing about it: stronger coercions cannot be refined into weaker ones. Thus our actual behaviour differs from our desired behaviour precisely when those two coercions differ: when $(me, id) \notin recd$ (from Figure 22.8) but $(me, id) \in sent$ (negating the above).

The appellants withdrew today in the *CMSK* (Common Mail-System Kernel) case, after it was shown that the specification of the system indeed allowed accidental deletion of messages before they had been read. Unexplained message loss had been widely reported and documented in the user community, and in a joint action by users of *CMSK* it was claimed that since the specification guaranteed no loss, the manufacturer was liable for damages.

In a rare move in such cases, the manufacturer showed that its own specification did after all allow such undesirable behaviour, in particular when message identifiers were used for reading before they had been registered as received: a randomly chosen message was in that case returned to the user, and the legitimate message was deleted from the system.

Users generally are now looking more closely at the published specification of *CMSK*, the future of which has been thrown into in doubt.

Figure 22.9

Thus we have not been able to show that Figure 22.8 refines 22.7, and must conclude therefore that our introduction of delay, in Figure 22.6, may have brought with it some unexpected consequences. Does that matter? Consider Figure 22.9 (and Exercise 22.4).

Now it is clear that the problem was essentially a coding trick that came back to haunt us: in the original specification of *Read* we allowed the command

$$sent := sent - \{(me, id)\}$$

to be executed even when (me, id) is not an element of $sent$. Later that became, without our noticing it, "executing $sent := sent - \{(me, id)\}$ even when (me, id) is not an element of $recd$" — altogether different, quite dangerous, and hard to detect without some kind of formal analysis.

We remedy matters by using more straightforward coding in *Read*, as shown in Figure 22.10. If we now performed the above analysis, the concrete procedure

$$
\begin{array}{l}
\textbf{module } \mathit{MailSys} \\
\quad \textbf{var } \mathit{msgs} : \mathit{Id} \nrightarrow \mathit{Msg}; \\
\qquad \mathit{sent}, \mathit{recd} : \mathit{Usr} \leftrightarrow \mathit{Id} \bullet \\
\\
\quad \textbf{procedure } \mathit{Send}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{msg} : \mathit{Msg};\ \ \mathit{tos} : \textbf{set } \mathit{Usr}; \\
\qquad\qquad\qquad\qquad \textbf{result } \mathit{id} : \mathit{Id}) \\
\quad \mathrel{\widehat{=}}\ \mathit{id}\colon [\mathit{id} \notin \mathsf{ran}\, \mathit{sent}]\ ; \\
\qquad \mathit{msgs}[\mathit{id}] := \mathit{msg}; \\
\qquad \mathit{sent} := \mathit{sent} \cup (\mathit{tos} \times \{\mathit{id}\}); \\
\\
\quad \textbf{procedure } \mathit{Receive}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \textbf{result } \mathit{ids} : \textbf{set } \mathit{Id}) \\
\quad \mathrel{\widehat{=}}\ \mathit{recd}\colon [\mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent}]\ ; \\
\qquad \mathit{ids} := \mathit{recd}[\mathit{me}]; \\
\\
\quad \textbf{procedure } \mathit{Read}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{id} : \mathit{Id};\ \ \textbf{result } \mathit{msg} : \mathit{Msg}) \\
\quad \mathrel{\widehat{=}}\ \textbf{if } (\mathit{me}, \mathit{id}) \in \mathit{recd} \rightarrow \\
\qquad\qquad \mathit{msg} := \mathit{msgs}[\mathit{id}]; \\
\qquad\qquad \mathit{sent}, \mathit{recd} := \mathit{sent} - \{(\mathit{me}, \mathit{id})\}, \mathit{recd} - \{(\mathit{me}, \mathit{id})\} \\
\qquad [\!]\ \ (\mathit{me}, \mathit{id}) \notin \mathit{recd} \rightarrow \textbf{choose } \mathit{msg} \\
\qquad \textbf{fi}; \\
\\
\quad \textbf{initially } \mathit{msgs} = \mathit{sent} = \mathit{recd} = \{\} \\
\textbf{end}
\end{array}
$$

Figure 22.10 The 'final' specification

ReadNotReceived would simply be

$$
\begin{array}{l}
[(\mathit{me}, \mathit{id}) \notin \mathit{recd}]\ ; \\
\textbf{choose } \mathit{msg}\ ,
\end{array}
$$

which we could reach without difficulty by direct refinement from Figure 22.7.

22.5 A first development: asynchronous delivery

With Figure 22.10 we have — finally — a specification that describes a reasonably realistic system in which messages may take some time to be delivered. We take it as our 'final' specification. (Why the quotes? Very few specifications are never changed, final or not.)

Our first move towards implementation will be concerned with the 'delay' built in to Procedure *Receive* of Figure 22.10. That describes the user's-eye view of it,

procedure *Deliver* $\mathrel{\widehat{=}}$ $recd$: $[recd_0 \subseteq recd \subseteq sent]$

procedure *Receive* (**value** $me : Usr$; **result** ids : **set** Id)
$\mathrel{\widehat{=}}$ $ids := recd[me]$

Figure 22.11 Asynchronous delivery, modifying Figure 22.10

but of course the delay does not necessarily happen in *Receive* itself; that is only where it is noticed.

Through the implementor's eyes instead, we would see messages moving towards their destination even while no user-accessible procedures are called. The subset relation in $recd \subseteq sent$ merely reflects the effect of calling *Receive* before they have arrived.

Our first development step is to introduce asynchronous message delivery, 'in the background'. Not having refinement rules for concurrency, however, we proceed informally: the actual delivery $recd$: $[recd_0 \subseteq recd \subseteq sent]$ is relocated from Procedure *Receive* into a new procedure *Deliver* which, it is understood, is called 'by the operating system' to move messages about. (Calls of *Deliver* are not even seen by the users.) The result is shown in Figure 22.11; note that *Deliver* needs no parameters.

With such a modest excursion into concurrency as this new module represents, we need only require that in an actual implementation there never be destructive interference between apparently concurrent calls on the procedures: a simple way of doing that is to introduce mutual exclusion so that at any time at most one procedure is active within it. In fact we would need to do that in any case — even without asynchronous delivery — if we were to share the mail system module between concurrently executing users.

Although we admit we are not strictly speaking implementing a refinement, we still should strive for as much confidence as possible in justifying the new behaviour, having learned our lessons in the scenarios of Figures 22.3 and 22.9 above. The key change, the new procedure, is not visible to users at all; it is called, 'in between' users' access to the module, by the operating system. Can we isolate that change, bringing the rest within the reach of our rigorous techniques?

If we were to add an 'asynchronous' procedure *Deliver* to our specification, Figure 22.10, we would only have to make its body **skip** to be sure that the change would not affect the users' perception of the module's behaviour. (We would have to ignore however the possibility that *Deliver* could be called 'so often' that users' access to *MailSys* is forever delayed, just as we have already ignored such starvation of one user by another.) Thus to justify the step we have just taken, we go back and add

$$\textbf{procedure } \mathit{Deliver} \;\widehat{=}\; \textbf{skip}$$

to Figure 22.10, and attempt to show that Figure 22.11 is a refinement of that. To find a coupling invariant, imagine executions of the abstract and concrete *Deliver* together: we can see that even if *recd* and *recd'* were equal beforehand, afterwards the concrete *recd'* could have grown. Thus we choose *recd* as abstract variable, introduce concrete variable *recd'*, and couple the two with

$$\mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}\ .$$

We look at the procedures in turn, taking a slightly informal view where matters seem clear enough.

As earlier, we can look at *Send* informally: since *sent* is only increased, the coupling invariant cannot be broken. For *Deliver* we must introduce a statement allowing *recd'* to grow, and we reason as follows:

$$
\begin{array}{ll}
 & \textbf{skip} \\
\multicolumn{2}{l}{\textit{becomes } \text{“}\textit{augment specification } 17.6\text{”}} \\
 & \mathit{recd}' \colon [\mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}\ ,\ \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}] \\
\sqsubseteq & \mathit{recd}' \colon [\mathit{recd}_0' \subseteq \mathit{recd}' \subseteq \mathit{sent}]\ .
\end{array}
$$

In *Receive* we must on the other hand *remove* the statement affecting *recd*. We have first

$$
\begin{array}{ll}
 & \mathit{recd} \colon [\mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent}] \\
\multicolumn{2}{l}{\textit{becomes } \text{“}\textit{augment specification } 17.6\text{”}} \\
 & \left[\begin{array}{c} \mathit{recd}, \mathit{recd}' \\ \hline \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent} \\ \hline \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent} \\ \mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent} \end{array}\right] \\
\sqsubseteq & \textbf{skip}\ .
\end{array}
$$

Note that it is in that last step that we need $\mathit{recd}' \subseteq \mathit{sent}$ in the coupling invariant.

The second command is $\mathit{ids} := \mathit{recd}[\mathit{me}]$ and here, apparently, we have a problem. Naturally, we want for its concrete equivalent $\mathit{ids} := \mathit{recd}'[\mathit{me}]$, but with our coupling invariant $\mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}$ we cannot show that the *ids* returned by the concrete *Receive* could have been returned also by the abstract. If *recd* and *recd'* differ initially — and they may — then the concrete and abstract *ids* will differ also.

On the other hand, we see that if the coupling invariant holds at the beginning of the whole procedure, then any concrete choice of *ids* can be mimicked by the abstract: the first abstract command could after all establish that $\mathit{recd} = \mathit{recd}'$. Thus we can sidestep this difficulty by going back and data-refining the two commands together, as follows:

$$
\begin{array}{l}
\mathit{recd} \colon [\mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent}]\ ; \\
\mathit{ids} := \mathit{recd}[\mathit{me}]
\end{array}
$$

$$= recd, ids\colon [recd_0 \subseteq recd \subseteq sent \wedge ids = recd[me]]$$

becomes "*augment specification* 17.6"

$$\left[\begin{array}{c} recd, ids, recd' \\ \hline recd \subseteq recd' \subseteq sent \\ \hline recd \subseteq recd' \subseteq sent \\ recd_0 \subseteq recd \subseteq sent \\ ids = recd[me] \end{array}\right]$$

$\sqsubseteq$ "again using $recd' \subseteq sent$"

$$recd, ids := recd', recd'[me]$$

becomes "*diminish assignment* 17.13"

$$ids := recd'[me]\ .$$

In *Read*, however, we have a problem we cannot sidestep. The guards of the alternation make essential use of *recd*: we cannot replace them by guards involving *recd'* only, as *recd'* is essentially an arbitrary superset of *recd*. We are stuck.

The difference between that and our earlier problem, with *Receive*, is that the abstract *Receive* allowed a nondeterministic alteration of *recd*: deliveries could occur there. But deliveries cannot occur in our abstract *Read*. Does it really matter? See Figure 22.12 (and Exercise 22.5).

Thus our Figure 22.11 is not a refinement of Figure 22.10 with its extra

procedure *Deliver* $\widehat{=}$ **skip** .

We are forced instead to add a third variable *deld* for 'delivered', independent of *sent* and *recd*. Our proposed concrete module is shown in Figure 22.13.

To show refinement between the extended Figure 22.10 and Figure 22.13, our coupling invariant is $read \subseteq deld' \subseteq sent$; there are no abstract variables. Arguing informally, we can see that *Send* is successfully data-refined, since *sent* is only increased. Similarly in *Read*, variables *sent*, *deld'* and *recd* are decreased 'in step'. For Procedure *Deliver* we have

skip

becomes "*augment specification* 17.6"

$$deld'\colon [recd \subseteq deld' \subseteq sent\ ,\ recd \subseteq deld' \subseteq sent]$$

$$\sqsubseteq\ deld'\colon [deld'_0 \subseteq deld' \subseteq sent]\ .$$

Note that even though the last command appears miraculous, we can as in Section 22.4 introduce a module invariant, in this case $deld' \subseteq sent$ (direct from the coupling invariant in fact), that would allow us to write

$$deld'\colon [deld' \subseteq sent\ ,\ deld'_0 \subseteq deld' \subseteq sent]$$

if we wished to.

Finally, for *Receive* we have

$$recd\colon [recd_0 \subseteq recd \subseteq sent]$$

A large-scale computer fraud was discovered today, involving an accounting loophole in the national electronic mail system.

It had been noticed that messages could be read before their delivery was reported. Believing their own specification, however, the mail authority had installed accounting software only at the actual point of reporting delivery. Thus 'unreported' messages could be read free of charge.

The loophole was exploited by a company that offered greatly reduced rates on bulk electronic mail. Its customers' messages would be collected and sent all at once as a single very long message. The identifier returned would then be sent as the body of an immediately following very short message. Since the mail system tended to allow short messages to overtake long ones, the second message was often delivered before the first: the identifier it contained would then be used to read the first, bypassing report of delivery — and bypassing charging as well. The occasional failure of the second message to overtake the first was easily covered by the enormous profit made overall.

Figure 22.12

becomes "*augment specification* 17.6"

$$\left[\begin{array}{c} recd, deld' \\ \hline recd \subseteq deld' \subseteq sent \\ \hline recd \subseteq deld' \subseteq sent \\ recd_0 \subseteq recd \subseteq sent \end{array}\right]$$

$$\sqsubseteq \; recd := deld' \; .$$

and our data-refinement is proved. We therefore accept Figure 22.13 as our first development step, introducing asynchronous delivery of messages.

$$
\begin{array}{l}
\textbf{module } \mathit{MailSys} \\
\quad \textbf{var } \mathit{msgs} : \mathit{Id} \nrightarrow \mathit{Msg}; \\
\qquad \mathit{sent}, \mathit{deld}, \mathit{recd} : \mathit{Usr} \leftrightarrow \mathit{Id} \bullet \\
\\
\quad \textbf{procedure } \mathit{Send}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{msg} : \mathit{Msg};\ \ \mathit{tos} : \textbf{set } \mathit{Usr}; \\
\qquad\qquad\qquad\qquad \textbf{result } \mathit{id} : \mathit{Id}) \\
\quad \mathrel{\widehat=}\ \mathit{id}\colon [\mathit{id} \notin \mathsf{ran}\, \mathit{sent}]\ ; \\
\qquad \mathit{msgs}[\mathit{id}] := \mathit{msg}; \\
\qquad \mathit{sent} := \mathit{sent} \cup (\mathit{tos} \times \{\mathit{id}\}); \\
\\
\quad \textbf{procedure } \mathit{Deliver} \\
\quad \mathrel{\widehat=}\ \mathit{deld}\colon [\mathit{deld}_0 \subseteq \mathit{deld} \subseteq \mathit{sent}] \\
\\
\quad \textbf{procedure } \mathit{Receive}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \textbf{result } \mathit{ids} : \textbf{set } \mathit{Id}) \\
\quad \mathrel{\widehat=}\ \mathit{recd} := \mathit{deld}; \\
\qquad \mathit{ids} := \mathit{recd}[\mathit{me}]; \\
\\
\quad \textbf{procedure } \mathit{Read}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{id} : \mathit{Id};\ \ \textbf{result } \mathit{msg} : \mathit{Msg}) \\
\quad \mathrel{\widehat=}\ \textbf{if } (\mathit{me}, \mathit{id}) \in \mathit{recd} \rightarrow \\
\qquad\qquad \mathit{msg} := \mathit{msgs}[\mathit{id}]; \\
\qquad\qquad \mathit{sent}, \mathit{deld}, \mathit{recd} := \mathit{sent} - \{(\mathit{me}, \mathit{id})\}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{deld} - \{(\mathit{me}, \mathit{id})\}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{recd} - \{(\mathit{me}, \mathit{id})\} \\
\qquad [\!]\ \ (\mathit{me}, \mathit{id}) \notin \mathit{recd} \rightarrow \textbf{choose } \mathit{msg} \\
\qquad \textbf{fi}; \\
\\
\quad \textbf{initially } \mathit{msgs} = \mathit{sent} = \mathit{deld} = \mathit{recd} = \{\} \\
\textbf{end}
\end{array}
$$

Figure 22.13 Asynchronous delivery — corrected

22.6 A second development: acknowledgements

In spite of our having introduced asynchronous delivery, the system is still centralised: variables *sent*, *deld* and *recd* each represent information that in a running system would probably be physically distributed. In particular, the execution of $\mathit{sent} := \mathit{sent} - \{(\mathit{me}, \mathit{id})\}$ in *Read* can make an identifier available 'instantaneously' for reallocation by $\mathit{id}\colon [\mathit{id} \notin \mathsf{ran}\, \mathit{sent}]$ in *Send*.

We remedy those implementation problems as follows. A new variable *used* will be introduced to record which identifiers may not yet be (re-)allocated. That frees *sent* to represent, not *all* of the $\mathit{Usr} \times \mathit{Id}$ pairs in use, but now only those that have been sent but not yet delivered. Similarly *deld* will now represent those delivered but not yet read. Finally, a new variable *ackd* will contain pairs that have been read but are not yet available for reuse. (In effect they are acknowledgements 'on the way back'.) That last breaks the 'instantaneous' link referred to above.

The abstract variables are *sent* and *deld* (leaving *recd* as it is); the concrete are $used'$, $sent'$, $deld'$ and $ackd'$; and the coupling invariant is

$$\begin{array}{l} sent = sent' \cup deld' \cup recd \\ deld = deld' \cup recd \\ used' = sent \cup ackd' \\ \\ \mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd'\ . \end{array}$$

We have thus a functional coupling invariant, with a data-type invariant showing that the four sets listed are disjoint. The third conjunct can be seen as part of the data-type invariant by taking the first conjunct into account. (Replace *sent* in the third conjunct by $sent' \cup deld' \cup recd$.)

We begin the data refinement with *Send* (of Figure 22.13). In its first command we can replace *sent* by $used'$, since the coupling invariant gives us that $sent \subseteq used'$. The second command is unaffected. For the third we recall in an assumption that $id \notin \mathsf{ran}\, used'$ is established by the new first command, and then we reason

$$\begin{array}{ll}
& \{id \notin \mathsf{ran}\, used'\}; \\
& sent := sent \cup (tos \times \{id\}) \\
\textit{becomes} & \text{``}\left\{\begin{array}{l}\textit{simple specification } 8.1 \\ \textit{data-refine specification } 17.15\end{array}\right.\text{''} \\
& \left[\begin{array}{c}
used', sent', deld', ackd' \\ \hline
id \notin \mathsf{ran}\, used' \\
used' = sent' \cup deld' \cup recd \cup ackd' \\
\mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\ \hline
used' = sent' \cup deld' \cup recd \cup ackd' \\
\mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\
sent' \cup deld' \cup recd = sent'_0 \cup deld'_0 \cup recd \cup (tos \times \{id\}) \\
deld' \cup recd = deld'_0 \cup recd
\end{array}\right] \\
\sqsubseteq & \\
& \left[\begin{array}{c}
used', sent' \\ \hline
id \notin \mathsf{ran}(deld' \cup recd \cup ackd') \\
\mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\ \hline
used' = used'_0 \cup (tos \times \{id\}) \\
\mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\
sent' = sent'_0 \cup (tos \times \{id\})
\end{array}\right] \\
\sqsubseteq & sent', used' := sent \cup (tos \times \{id\}), used' \cup (tos \times \{id\})\ .
\end{array}$$

In *Deliver* we have

$$deld\colon [deld_0 \subseteq deld \subseteq sent]$$

becomes "*data-refine specification* 17.15"

$$\left[\begin{array}{c} used', deld', sent', ackd' \\ \hline used' = sent' \cup deld' \cup recd \cup ackd' \\ \mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\ \hline used' = sent' \cup deld' \cup recd \cup ackd' \\ \mathsf{disjoint}\ sent'\ \ deld'\ \ recd\ \ ackd' \\ sent' \cup deld' = sent'_0 \cup deld'_0 \\ deld'_0 \subseteq deld' \end{array}\right]$$

$$\begin{array}{ll} \sqsubseteq & \mathbf{var}\ muis, auis : Usr \leftrightarrow Id \bullet \\ & muis, auis\colon [muis \subseteq sent' \wedge auis \subseteq ackd']\,; \\ & sent', ackd' := sent' - muis, ackd' - auis; \\ & deld', used' := deld' \cup muis, used' - auis \end{array}$$

The local variables *muis* and *auis* represent the arrival of new messages and acknowledgements respectively; the former are added to $deld'$, and the latter deleted from $used'$.

In *Receive* we have

$$recd := deld$$

becomes

$$\left[\begin{array}{c} recd, deld' \\ \hline \mathsf{disjoint}\ deld'\ \ recd \\ \hline deld'_0 \cup recd_0 = deld' \cup recd \\ \mathsf{disjoint}\ deld'\ \ recd \\ recd = deld'_0 \cup recd_0 \end{array}\right]$$

$$\sqsubseteq \ deld', recd := \{\}, recd \cup deld' \ ,$$

and the second command is unchanged.

Finally, in *Read* we need be concerned only with the assignments to the three sets, and we will need that $(me, id) \in recd$ (which we have from the guard); thus we proceed

$$\begin{array}{l} \{(me, id) \in recd\}; \\ sent, deld, recd := sent - \{(me, id)\}, \\ \qquad\qquad\qquad\quad deld - \{(me, id)\}, \\ \qquad\qquad\qquad\quad recd - \{(me, id)\} \end{array}$$

becomes

$$\left[\begin{array}{c} recd, ackd' \\ \hline (me, id) \in recd \\ \mathsf{disjoint}\ recd\ \ ackd' \\ \hline recd_0 \cup ackd'_0 = recd \cup ackd' \\ \mathsf{disjoint}\ recd\ \ ackd' \\ recd = recd_0 - \{(me, id)\} \end{array}\right]$$

$$\sqsubseteq \ recd, ackd' := recd - \{(me, id)\}, ackd' \cup \{(me, id)\} \ .$$

The result is Figure 22.14.

module *MailSys*
var $msgs : Id \nrightarrow Msg$;
$used, sent, deld, recd, ackd : Usr \leftrightarrow Id \bullet$

procedure *Send* (**value** $me : Usr$; $msg : Msg$; $tos :$ **set** Usr;
result $id : Id$)
$\widehat{=}$ id: $[id \notin \mathsf{ran}\, used]$;
$msgs[id] := msg$;
$sent, used := sent \cup (tos \times \{id\}), used \cup (tos \times \{id\})$;

procedure *Deliver*
$\widehat{=}$ $|[$ **var** $muis, auis : Usr \leftrightarrow Id \bullet$
$muis, auis$: $[muis \subseteq sent \wedge auis \subseteq ackd]$;
$sent, ackd := sent - muis, ackd - auis$;
$deld, used := deld \cup muis, used - auis$
$]|$;

procedure *Receive* (**value** $me : Usr$; **result** $ids :$ **set** Id)
$\widehat{=}$ $deld, recd := \{\}, recd \cup deld$;
$ids := recd[me]$;

procedure *Read* (**value** $me : Usr$; $id : Id$; **result** $msg : Msg$)
$\widehat{=}$ **if** $(me, id) \in recd \rightarrow$
$msg := msgs[id]$;
$recd, ackd := recd - \{(me, id)\}, ackd \cup \{(me, id)\}$
$[\!]$ $(me, id) \notin recd \rightarrow$ **choose** msg
fi;

initially $msgs = used = sent = deld = recd = ackd = \{\}$
end

Figure 22.14 Acknowledgements and distribution

22.7 The final development: packets

In a real mail system it would be the messages themselves that moved from place to place, not just their identifiers, and it finally is time to develop our specification in that direction. We introduce a type *Pkt* to represent both messages and acknowledgements; the new concrete variables will be *sets* of those and the procedures will, after data refinement, move packets between them:

$$\begin{aligned}&\textbf{type } \mathit{From} \mathrel{\widehat{=}} \mathit{Usr};\\ &\qquad \mathit{To} \mathrel{\widehat{=}} \mathit{Usr};\\ &\qquad \mathit{Pkt} \mathrel{\widehat{=}} \mathsf{msg}\,\mathit{Id}\ \mathit{From}\ \mathit{To}\ \mathit{Msg} \mid \mathsf{ack}\,\mathit{Id}\ \mathit{From}\ \mathit{To}\ .\end{aligned}$$

The type synonyms *From* and *To* are introduced to make the use of the two *Usr* fields clear.

To express our coupling invariant — to recover the abstract variables from the new concrete sets — we must define several projection functions that extract sets of $Usr \times Id$ pairs from sets of packets:

$$\mathsf{mui}, \mathsf{aui}, \mathsf{ui} : \textbf{set}\ \mathit{Pkt} \to (\mathit{Usr} \leftrightarrow \mathit{Id})$$

$$\begin{aligned}&\mathsf{mui}\,pp \mathrel{\widehat{=}} \{i : \mathit{Id};\ f : \mathit{From};\ t : \mathit{To};\ m : \mathit{Msg} \mid \mathsf{msg}\ i\ f\ t\ m \in pp \bullet (t, i)\}\\ &\mathsf{aui}\,pp \mathrel{\widehat{=}} \{i : \mathit{Id};\ f : \mathit{From};\ t : \mathit{To} \mid \mathsf{ack}\ i\ f\ t \in pp \bullet (t, i)\}\\ &\mathsf{ui}\,pp \mathrel{\widehat{=}} \mathsf{mui}\,pp \cup \mathsf{aui}\,pp\ .\end{aligned}$$

We will need to express also that any given $Usr \times Id$ pair is represented uniquely in a set of packets — that is, that there are never two distinct packets with the same pair:

$$\mathsf{ui1}\,pp \mathrel{\widehat{=}} (\forall\, p, p' : pp \bullet \mathsf{ui}\{p\} = \mathsf{ui}\{p'\} \Rightarrow p = p')\ .$$

And finally, we need a projection function to extract the message texts from our packets:

$$\mathsf{im} : \textbf{set}\ \mathit{Pkt} \to (\mathit{Id} \leftrightarrow \mathit{Msg})$$

$$\mathsf{im}\,pp \mathrel{\widehat{=}} \{i : \mathit{Id};\ f : \mathit{From};\ t : \mathit{To};\ m : \mathit{Msg} \mid \mathsf{msg}\ i\ f\ t\ m \in pp \bullet (i, m)\}\ .$$

Our abstract *sent* and *ackd* will be represented by the single concrete *sent'*, representing the packets of either kind in transit, with the tags msg and ack available to separate them; the coupling invariant is straightforward, if a bit lengthy:

$$\begin{aligned}&\mathit{sent} = \mathsf{mui}\,\mathit{sent}'\\ &\mathit{deld} = \mathsf{mui}\,\mathit{deld}'\\ &\mathit{recd} = \mathsf{mui}\,\mathit{recd}'\\ &\mathit{ackd} = \mathsf{aui}\,\mathit{sent}'\end{aligned}$$

$$\mathsf{ui1}(\mathit{sent}' \cup \mathit{deld}' \cup \mathit{recd}')$$

$$\mathit{msgs} \supseteq \mathsf{im}(\mathit{sent}' \cup \mathit{deld}' \cup \mathit{recd}')\ .$$

The last conjunct states that our sets of packets do contain message bodies consistent with *msgs* — and our aim is to remove *msgs*.

Data refinement of *Send*, *Receive* and *Read* is straightforward; and since the coupling invariant allows $msgs[id]$ in *Read* to be recovered from *recd*, we find that *msgs* has become auxiliary and that we can remove it. It can be removed from *Send* in any case.

For *Deliver* we first rewrite our abstract program slightly, anticipating that *sent* and *ackd* will be collected together:

$$
\begin{array}{ll}
 & |[\ \mathbf{var}\ muis, auis : Usr \leftrightarrow Id \bullet \\
 & \quad muis, auis\colon [muis \subseteq sent' \wedge auis \subseteq ackd]\,; \\
 & \quad sent, ackd := sent - muis, ackd - auis; \\
 & \quad deld, used := deld \cup muis, used - auis \\
 &]| \\
= & |[\ \mathbf{var}\ uis, muis, auis : Usr \leftrightarrow Id \bullet \\
 & \quad uis\colon [uis \subseteq sent]\,; \\
 & \quad muis, auis := uis \cap sent, uis \cap ackd; \\
 & \quad sent, ackd := sent - uis, ackd - auis; \\
 & \quad deld, used := deld \cup muis, used - auis\ . \\
 &]|
\end{array}
$$

For the data refinement we introduce concrete local variables pp, mpp and app to replace the abstract local variables uis, $muis$ and $auis$, with coupling invariant

$$\mathsf{ui}\, pp = uis \wedge \mathsf{mui}\, mpp = muis \wedge \mathsf{aui}\, app = auis\ .$$

The assignment $sent, ackd := \cdots$ will be data-refined to a single assignment to $sent'$, because it effectively represents the union of $sent$ and $ackd$.

The result — our final module — is in Figure 22.15.

22.8 Exercises

Ex. 22.1 ♡ Modify the original specification of Figure 22.1 to include a procedure *Cancel* that can be used to remove all unreceived copies of a message from the system. Does the specification contain enough detail to prevent 'unauthorised' removal?

Ex. 22.2 Modify the original specification of Figure 22.1 to include a procedure *Unread* that can be used to determine which users have not yet read a given message.

Ex. 22.3 ♡ Explain precisely how the specification of Figure 22.1, as amended in Figure 22.2, could have led to the scenario described in Figure 22.3.

Ex. 22.4 Explain precisely how the module of Figure 22.6 could have behaved as suggested in the scenario described in Figure 22.9.

Ex. 22.5 Explain precisely how the module of Figure 22.10, as amended in Figure 22.11, could have behaved as suggested in the scenario described in Figure 22.12.

Ex. 22.6 Imagine a building with one lift serving several floors. Outside the lift door, on each floor, is a panel of buttons and lights with one button/light pair for each floor. Inside the lift are no buttons or lights at all.

module *MailSys*
  type $From \mathrel{\widehat{=}} Usr$;
    $To \mathrel{\widehat{=}} Usr$;
    $Pkt \mathrel{\widehat{=}} \mathsf{msg}\, Id\ From\ To\ Msg \mid \mathsf{ack}\, From\ To\ Id$;

  var $used : Usr \leftrightarrow Id$;
    $sent, deld, recd : \mathsf{set}\, Pkt \bullet$

  procedure *Send* (**value** $me : Usr;\ msg : Msg;\ tos : \mathsf{set}\, Usr$;
      result $id : Id$)
  $\mathrel{\widehat{=}}$ id: $[id \notin \mathsf{ran}\, used]$;
    $used := used \cup (tos \times \{id\})$;
    $sent := sent \cup \{t : tos \mid \mathsf{msg}\, id\ me\ t\ msg\}$;

  procedure *Deliver*
  $\mathrel{\widehat{=}}$ $|[$ **var** $pp, mpp, app : \mathsf{set}\, Pkt \bullet$
      pp: $[pp \subseteq sent]$;
      $sent := sent - pp$;
      $mpp, app := pp \cap \mathsf{ran}\, \mathsf{msg}, pp \cap \mathsf{ran}\, \mathsf{ack}$;
      $deld, used := deld \cup mpp, used - \mathsf{ui}\, app$
    $]|$;

  procedure *Receive* (**value** $me : Usr$; **result** $ids : \mathsf{set}\, Id$)
  $\mathrel{\widehat{=}}$ $deld, recd := \{\}, recd \cup deld$;
    $ids := (\mathsf{ui}\, recd)[me]$;

  procedure *Read* (**value** $me : Usr;\ id : Id$; **result** $msg : Msg$)
  $\mathrel{\widehat{=}}$ $|[$ **var** $p : Pkt;\ u : Usr; \bullet$
      if $(me, id) \in \mathsf{ui}\, recd \rightarrow$
        p, f, msg: $[p = \mathsf{msg}\, id\ f\ me\ msg \land p \in recd]$;
        $recd, sent := recd - \{p\}, sent \cup \{\mathsf{ack}\, id\ f\ me\}$
      $[\!]$ $(me, id) \notin \mathsf{ui}\, recd \rightarrow$ **choose** msg
      fi
    $]|$;

  initially $used = sent = deld = recd = \{\}$
end

Figure 22.15 A refinement of Figure 22.10

To use the lift one presses the button, next to the doors, for the desired destination; the corresponding light should light if it is not lit already. When the doors open, one enters the lift in the hope that it will eventually visit that destination (whose light should be lit).

Design a module based on the type

$$Floor \mathrel{\widehat{=}} 0 \rightarrow F$$

that contains these procedures with the meanings informally indicated:

- *Press* (**value** $f, b : Floor$) — Press button b outside the lift doors on floor f. (Called by lift user.)
- *Check* (**value** $f, l : Floor$; **result** $b : Boolean$) — Check whether the light l on floor f is lit. (Called by lift user.)
- *Visit* (**result** $f : Floor$) — Close the doors, select a floor f 'randomly' which it would be useful to visit, go there, and open the doors. (Called by lift operator.)

Hint: There are probably unanswered questions about the informal specification above; answer them yourself. Consider using set-valued variables inside the module.

Ex. 22.7 Let T be a set of *telephones* connected to an exchange that supports conference calls, so that collections of (people using) telephones can hold group conversations.

Declare a variable xns of appropriate type that could represent the set of conversations in progress at any moment; write then, in English *and* in mathematics, an invariant that ensures there is no telephone in more than one conversation.

Now suppose rqs is to represent the set of conversations requested but not in progress (thus 'pending'). Specify and justify an operation (with the default precondition, true)

$$xns, rqs\colon [???]$$

that connects as many new conversations as is possible without disturbing existing conversations. Note that the invariant over xns must be respected. *Hint*: The set xns should be made locally maximal in some sense.

Then use the structures above to supply (abstract) program text for the informally described module in Figure 22.16. (You need not fill in *Connect*, already specified in the text above.)

Finally, give a sensible definition of a new procedure *Chat* (**value** $t : T$) that causes t immediately to join a single 'chat line', able then to converse with all others that have not executed *HangUp* since they last executed *Chat*. Modify your other definitions if necessary (but the less, the better).

Ex. 22.8 ♡ Show that Figure 22.10, the 'final' specification, is refined by Figure 22.17. Is that a problem?

module *TelephoneExchange*
  var *xns* : ???
    rqs : ???•

  procedure *Request* (**value** *tt* : ???)
  $\widehat{=}$ "Request a conversation *tt*";

  procedure *Connect* $\widehat{=}$ "described in the text";

  procedure *Converse* (**value** *t* : *T*;
              result *tt* : ???)
  $\widehat{=}$ "Identify *all* participants in any conversation involving *t*";

  procedure *HangUp* (**value** *t* : *T*)
  $\widehat{=}$ "Withdraw *t* from any conversation in which it is involved";

  initially "no conversations"
end

Note that

- A single telephone may be part of many requests (but of at most one conversation).
- *Connect* may be thought of as being executed at suitable moments by the exchange itself.
- *HangUp* should allow other participants in a conversation to continue.

Figure 22.16 Telephone module

$$
\begin{array}{l}
\textbf{module } \mathit{MailSys} \\
\quad \textbf{var } \mathit{used} : \textbf{set } \mathit{Id}; \\
\\
\quad \textbf{procedure } \mathit{Send}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{msg} : \mathit{Msg};\ \ \mathit{tos} : \textbf{set } \mathit{Usr}; \\
\qquad\qquad\qquad\qquad\quad \textbf{result } \mathit{id} : \mathit{Id}) \\
\quad \widehat{=}\ \ \mathit{id} \!:\notin \mathit{used}; \\
\qquad\ \mathit{used} := \mathit{used} \cup \{\mathit{id}\}; \\
\\
\quad \textbf{procedure } \mathit{Receive}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \textbf{result } \mathit{ids} : \textbf{set } \mathit{Id}) \\
\quad \widehat{=}\ \ \mathit{ids} := \{\}; \\
\\
\quad \textbf{procedure } \mathit{Read}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \ \mathit{id} : \mathit{Id};\ \ \textbf{result } \mathit{msg} : \mathit{Msg}) \\
\quad \widehat{=}\ \ \textbf{choose } \mathit{msg} \\
\textbf{end}
\end{array}
$$

Figure 22.17 An unexpected implementation