Applying Process Refinement to a Safety-Relevant System

Glenn Bruns
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, UK

April 14, 1994

1 Introduction

Development by sound refinement steps is an important method for showing the correctness of a system relative to its specification. The refinement of sequential systems has been studied extensively over the past two decades, but the refinement of concurrent systems has received much less attention. Various notions of process refinement have been suggested, such as observation equivalence[7], language inclusion [5], and other preorder relations [9, 3], but so far little evidence of their utility has been garnered from attempts at applying them to real systems.

Here we present a case study showing the application of modal process logic [6] to the formal refinement of a failure-recovery protocol used in a safety-relevant air transport application. Intuitively, modal process logic allows specifications to express that certain actions *must* be performed by an implementation, and that other actions *may* be performed by an implementation. This sense of refinement was found to fit the application problem well.

The paper is organised as follows. We begin by describing the application and presenting a formal specification and implementation in CCS. We describe observation equivalence, a standard equivalence relation for processes, and show that the specification and implementation are not equivalent in this sense. Next, modal process logic, and its notions of "refinement" and "implementation" are explained. The specification is then rewritten in modal process logic and the CCS implementation is shown to be acceptable in the sense of modal process logic. To simplify the refinement proof, we adopt an "up-to" technique [8, 2]. The final section presents our conclusions.

2 A Failure Recovery Protocol

The application problem is an aviation control information display system developed by Praxis Systems plc for the British Civil Aviation Authority. The system is regarded as "safety-relevant", since it has an indirect impact on air safety. A main job of the system is to communicate flight information stored on a reliable central computer over a duplicated ring network to display computers. The display computers keep a complete local copy of the master data. As the master data in the central computer is changed, update messages are sent to the display computers. The content of update messages is relative to the expected state of the local data, so it is vital that update messages arrive in order and that message loss be detected.

Our study concerns the protocol for communication between the central computer and display computers. A low-level protocol is usually able to detect and resend lost messages without attention of the application programs on the sending and receiving computers. If losses occur that cannot be handled at this level, then the low-level protocol must announce the loss. A *recovery protocol* is then invoked, in which the complete store of master data is sent to the receiver that suffered the loss.

The dotted line in Figure 1 shows the boundary between the application program using the recovery protocol and the lower system levels that provide the service. The arrows that cross the boundary

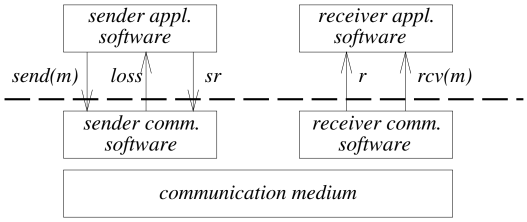

Figure 1: The Recovery Protocol

represent the actions of the protocol. In normal operation, messages pass from sender to receiver via actions *send*(m) and *rcv*(m). Messages that have been sent but not yet delivered are buffered by the communication software. The action *loss* is received by the sender if messages are lost that cannot be recovered by the usual low-level mechanisms. When the sender is ready to transmit the recovery data, it performs a *sr* ("start recovery") action. In the meantime, the receiver can continue to receive messages sent before the messages that were lost. However, after the receiver is notified by the *r* ("notify recovery") action, it will thereafter receive the recovery data.

The two most important properties of the protocol are the following:

- Messages sent after the lost messages, but before the "start recovery" action, must not be delivered.
- Messages sent after the the "start recovery" action must not be delivered until after the "notify recovery" action.

Before presenting the specification and implementation, we give a brief description of CCS [7].

The terms of CCS represent processes that perform *actions*. The set of CCS actions contains *names* $(a, b, \ldots)$, *co-names* $(\overline{a}, \overline{b}, \ldots)$, and the special synchronisation action τ. We let α, β range over Act. The set of *labels* contains all actions except τ. We let l, l' range over labels. Complementation extends to all labels: $\overline{\overline{a}} = a$.

Processes are given as *agent expressions* having the following syntax, where L ranges over sets of labels and f ranges over functions from actions to actions:

$$E ::= 0 \mid a.E \mid E_1 + E_2 \mid E_1 \mid E_2 \mid E\backslash L \mid P[f]$$

The set of agent expressions is denoted by $\mathcal{E}$. Agent constants $(A, B, \ldots)$ are also allowed as process terms, having definitions of the form $A \stackrel{\mathrm{def}}{=} E$. We refer to the set of all agents expressions and agent constants as simply *agents*, denoted by $\mathcal{P}$.

The meaning of agents is given as a labelled transition system $(\mathcal{P}, Act, \rightarrow)$, where $\rightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$ is a transition relation. We write $P \xrightarrow{\alpha} P'$ if $(P, \alpha, P') \in \rightarrow$. The relation $\rightarrow$ is defined to be the least relation satisfying the following rules:

$$\text{prefix} \quad : \quad \alpha.E \xrightarrow{\alpha} E$$

$$+ \quad : \quad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'} \qquad \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 + E_2 \xrightarrow{\alpha} E_2'}$$

$$\mid \quad : \quad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1|E_2 \xrightarrow{\alpha} E_1'|E_2} \qquad \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1|E_2 \xrightarrow{\alpha} E_1|E_2'} \qquad \frac{E_1 \xrightarrow{l} E_1',\ E_2 \xrightarrow{\overline{l}} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'}$$

$$\backslash \quad : \quad \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \quad \alpha \notin L \cup \overline{L}$$

$$f \quad : \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\text{def} \quad : \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad A \stackrel{\mathrm{def}}{=} P$$

For convenience, we will use *value-passing CCS*, which extends CCS to include parameterised actions, parameterised agents, and conditional statements. These new notational features are defined by translation to basic CCS, under the assumption of a sub-language for data values. The input action $a(x).P$ is defined as $\sum_{v \in V_a} a_v.P[v/x]$, where V_a is the set of values associated with action a, and $P[v/x]$ is the term obtained by substituting v for all free occurrences of x within the agent P. The output action $\overline{a}(e).P$ is defined as $\overline{a}_v.P$, where v is the value of the expression e. The agent definition $A(x_1, x_2, \ldots, x_n) \stackrel{\text{def}}{=} P$ is defined as the set of definitions $\{(A_{v_1,v_2,\ldots,v_n} \stackrel{\text{def}}{=} P[v_1/x_1, \ldots, v_n/x_n]) : (v_1, \ldots, v_n) \in V_{x_1} \times \ldots \times V_{x_n}\}$. The agent constant $A(e_1, \ldots, e_n)$ is defined to be $A_{v_1,\ldots,v_n}$, where v_i is the value of expression e_i. Finally, the expression *if* b *then* P *else* Q is defined to be P if the boolean expression b evaluates to true, else P. The simpler form *if* b *then* P is defined to be *if* b *then* P *else* 0.

As an example of a value-passing CCS process, we present a two-celled buffer. A single cell is defined as follows:

$$\begin{aligned} Buf &\stackrel{\text{def}}{=} in(x).Buf'(x) \\ Buf'(x) &\stackrel{\text{def}}{=} \overline{out}(x).Buf \end{aligned}$$

To connect two cells, we define a linking operator $\frown$ using composition and restriction:

$$P_1 \frown P_2 \stackrel{\text{def}}{=} (P_1[c/in] \mid P_2[c/out])\backslash\{c\}$$

A two celled buffer process can now be defined as two single cells linked together:

$$BufPair \stackrel{\text{def}}{=} Buf \frown Buf$$

Sequences are used as values in the recovery protocol. Sequence construction is denoted by $\langle x_1, x_2, \ldots, x_n \rangle$, with $\langle\rangle$ denoting the empty sequence. The append operation is denoted by concatenation. The head and tail of a sequence s are denoted by $\mathbf{hd}\, s$ and $\mathbf{tl}\, s$. The length of a sequence s is denoted by $\mathbf{len}\, s$. The set of all prefixes of a sequence s is denoted $\mathbf{prefix}\, s$.

We are now ready to present a formal service specification of the recovery protocol in CCS. This specification, closely based on a version developed by Praxis, makes the following simplifying assumptions: there is a single receiving computer, the protocol has unbounded buffering capacity, messages are atomic, the only kind of loss is a single group of consecutively sent messages, and a second loss cannot occur until after recovery is complete.

$$\begin{aligned} Spec &\stackrel{\text{def}}{=} E(\langle\rangle) \\ E(s) &\stackrel{\text{def}}{=} send(m).E(s\langle m\rangle) + \\ &\quad\ (\text{if } s \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s).E(\mathbf{tl}\, s)) + \\ &\quad\ \textstyle\sum_{s_1 \in \mathbf{prefix}\, s} \overline{loss}.E_l(s_1) \\ E_l(s) &\stackrel{\text{def}}{=} send(m).E_l(s) + \\ &\quad\ (\text{if } s \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s).E_l(\mathbf{tl}\, s)) + \\ &\quad\ sr.E_r(s, \langle\rangle) \\ E_r(s_1, s_2) &\stackrel{\text{def}}{=} send(m).E_r(s_1, s_2\langle m\rangle) + \\ &\quad\ (\text{if } s_1 \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s_1).E_r(\mathbf{tl}\, s_1, s_2)) + \\ &\quad\ \overline{r}.E(s_2) \end{aligned}$$

Initially, *Spec* behaves as a simple buffer. A loss of messages is reported by the action $\overline{loss}$. Messages sent after the loss are accepted but discarded; messages that were sent before the loss are delivered. After reception of the *sr* message, messages are accepted again, but these messages are not delivered until after the $\overline{r}$ action occurs, when all pre-loss messages are discarded.

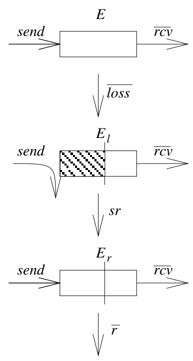

Figure 2: Behaviour of the Specification

The behaviour of *Spec* is illustrated in Figure 2, a kind of transition graph. A node in the graph represents a set of process states that differ only in their buffered messages. For example, the top node of the graph, labelled E, represents processes of the form $E(s)$, where s is a sequence of messages. Each node is drawn as a box, suggesting a message buffer. The $send$ and $\overline{rcv}$ arrows attached to each box represent the actions by which messages enter and exit the buffer. Edges in the graph represent actions that take the protocol from one "phase" to another. Note that Figure 2 does not precisely describe process *Spec*. For example, $\overline{rcv}$ actions are not possible for states in which the buffer is empty.

The line dividing the bottom box is intended to suggest that the buffer has two parts, and that messages in the left-hand part cannot be output through port $\overline{rcv}$. The shading in the middle box represents the messages lost by the $\overline{loss}$ action.

3 Implementing the Protocol

As implied by Figure 1, the protocol is implemented by three concurrent components: the communications software running on the sending computer, the communications software running on the receiving computer, and the low-level communications medium. These components are represented by the CCS processes S, R, and M. Note that S and R represent communication processes, not the application programs.

The key idea behind the implementation is that the communication medium is blind to the details of the recovery protocol; the medium's only role is to postpone delivery of messages received after the loss until both sender and receiver have been notified of the loss. The sender and receiver coordinate through a distinguished message transmitted by the sender after the application program on the sending computer has signalled the start of recovery. The medium treats this special message like any other. After the receiver is notified of the loss, it discards all incoming messages until after the special recovery message is received.

$$\begin{aligned} S &\stackrel{\text{def}}{=} send(m).S'(m) + cns.S_r \\ S'(m) &\stackrel{\text{def}}{=} \overline{csend}(m).S \\ S_r &\stackrel{\text{def}}{=} send(m).S_r'(m) + sr.S_r'' \\ S_r'(m) &\stackrel{\text{def}}{=} \overline{csend}(m).S_r \\ S_r'' &\stackrel{\text{def}}{=} \overline{csend}(\mathbf{R}).S \end{aligned}$$

The process S is the communications process on the sending machine. Some time after receiving notification of the loss through action cns ("C notifies S"), it transmits the special recovery message and reverts to its normal behaviour.

$$
\begin{aligned}
C(s) &\stackrel{\text{def}}{=} csend(m).C(s\langle m\rangle) + \\
&\quad (\text{if } s \neq \langle\rangle \text{ then } \overline{crcv}(\mathbf{hd}\, s).C(\mathbf{tl}\, s)) + \\
&\quad \textstyle\sum_{(s_1,s_2)\in\{(s_1,s_2)\mid s=s_1s's_2\}} \overline{loss}.C_l(s_1,s_2) \\
C_l(s_1,s_2) &\stackrel{\text{def}}{=} csend(m).C_l(s_1,s_2\langle m\rangle) + \\
&\quad (\text{if } s_1 \neq \langle\rangle \text{ then } \overline{crcv}(\mathbf{hd}\, s_1).C_l(\mathbf{tl}\, s_1, s_2)) + \\
&\quad \overline{cns}.C_{ls}(s_1,s_2) \\
C_{ls}(s_1,s_2) &\stackrel{\text{def}}{=} csend(m).C_{ls}(s_1,s_2\langle m\rangle) + \\
&\quad (\text{if } s_1 \neq \langle\rangle \text{ then } \overline{crcv}(\mathbf{hd}\, s_1).C_{ls}(\mathbf{tl}\, s_1, s_2)) + \\
&\quad \overline{cnr}.C(s_1s_2)
\end{aligned}
$$

The process C is the communications medium. After a loss occurs, delivery of post-loss messages is delayed until the sender and receiver have been notified. After notification, C reverts to its normal behaviour.

$$
\begin{aligned}
R &\stackrel{\text{def}}{=} crcv(m).R'(m) + cnr.R_d \\
R'(m) &\stackrel{\text{def}}{=} \overline{rcv}(m).R + cnr.R_d \\
R_d &\stackrel{\text{def}}{=} crcv(m).(\text{if } m = \mathbf{R} \text{ then } \overline{r}.R \text{ else } R_d)
\end{aligned}
$$

The process R is the communications process on the receiving machine. After notification of a loss, it accepts messages from the medium but discards them until after the special recovery message has been received.

$$
Impl \stackrel{\text{def}}{=} (S \mid C(\langle\rangle) \mid R)\backslash\{cnr, cns, csend, crcv\}
$$

We use some shorthand when referring to states of the implementation. The process $(S \mid C(s) \mid R)\backslash\{cnr, cns, csend, crcv\}$ is written as $SCR(s)$. If a message m is buffered by process S, as in state $S'(m)$, we write $S'CR(s\langle m\rangle)$. The message is shown at the end of the sequence because it was the last message received. Similarly, if process R holds a message m in its buffer, as in state $R'(m)$, we write $SCR'(\langle m\rangle s)$. The same idea is used when process C has two parameters.

Figure 3 illustrates the behaviour of *Impl*. As in Figure 2, each node of Figure 3 represents a sets of states. For example, the top node in the figure represents the set $\{SCR(s), S'CR(s), SCR'(s), S'CR'(s) \mid s \text{ a sequence of messages}\}$. The right-most flow arrow of nodes S_rCR_d and SCR_d is missing to indicate that no such processes can perform the action $\overline{rcv}$. We write τ_{cns} to denote the τ action that results from the synchronisation of a cns and a $\overline{cns}$ action.

4 Bisimulation As Refinement

One view of refinement is that a refinement may alter the structure of a system, but not its behaviour. In CCS, the most common notion of behavioural equivalence is called *observation equivalence* [7], which treats the synchronisation action τ as invisible to an observer. Informally, two processes are observation equivalent if an observable action of one process can be matched by another, and conversely, so that all pairs of derivative processes are themselves observation equivalent.

We will define observable actions using the notation $P \Rightarrow P'$ to stand for $P(\stackrel{\tau}{\rightarrow})^* P'$. An observable action, $P \stackrel{\alpha}{\Rightarrow} P'$ can then be defined as $P \Rightarrow \stackrel{\alpha}{\rightarrow} \Rightarrow P'$. Before giving the definition of observation equivalence, we also need to define the notation $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$, which stands for $P \Rightarrow P'$ if $\alpha = \tau$, and $P \stackrel{\alpha}{\Rightarrow} P'$ otherwise.

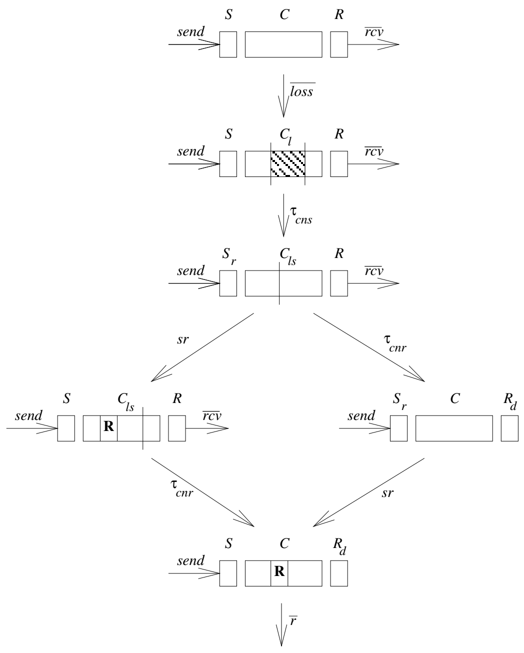

Figure 3: Behaviour of the Implementation

Definition 1 *A* weak bisimulation $\mathcal{B}$ *is a binary relation on* $\mathcal{P}$ *such that* $(P,Q) \in \mathcal{B}$ *implies, for all* $\alpha \in Act$*:*

1. *Whenever* $P \xrightarrow{\alpha} P'$*, then* $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ *for some* Q' *such that* $(P',Q') \in \mathcal{B}$*,*

2. *Whenever* $Q \xrightarrow{\alpha} Q'$*, then* $P \overset{\hat{\alpha}}{\Rightarrow} P'$ *for some* P' *such that* $(P',Q') \in \mathcal{B}$*.*

Two processes are *observation equivalent*, written $P \approx Q$, if (P,Q) belongs to some bisimulation $\mathcal{B}$. In other words, to prove observation equivalence, it is enough to find a bisimulation relating the agents of interest. For example, consider a sequential buffer process that can hold n values:

$$\begin{aligned} Buf_n(s) \;\; &\stackrel{\mathrm{def}}{=}\;\; \text{if}\,(\mathbf{len}\; s) < n \;\text{then}\; in(x).Buf_n(s\langle x\rangle) + \\ & \qquad \text{if}\,(\mathbf{len}\; s) > 0 \;\text{then}\; \overline{out}(\mathbf{hd}\; s).Buf_n(\mathbf{tl}\; s) \end{aligned}$$

A weak bisimulation $\mathcal{B}$ that proves that $Buf_2(\langle\rangle)$ is observation equivalent to the two-celled buffer *BufPair* defined earlier is the following:

$$\begin{aligned} \mathcal{B} \;\; \stackrel{\mathrm{def}}{=} \;\; & \{(Buf_2(\langle\rangle), BufPair), \\ & (Buf_2(\langle x\rangle), Buf'(x) \frown Buf), \\ & (Buf_2(\langle x\rangle), Buf \frown Buf'(x)), \\ & (Buf_2(\langle x, y\rangle), Buf'(y) \frown Buf'(x))\} \end{aligned}$$

To prove that $\mathcal{B}$ is indeed a bisimulation, each pair of processes in $\mathcal{B}$ is checked to show that it satisfies the conditions given in the definition of bisimulation. The processes of each pair are intuitively similar as they each buffer the same messages.

In attempting to build a bisimulation to show that $Spec \approx Impl$, a useful way to begin is by relating states that buffer the same messages. If we were to relate two states that buffered different messages, then one would be deliver a particular sequence of messages and the other could not. Taking into account buffered messages and the observable actions $\overline{loss}$, sr, and $\overline{r}$, a rough description of a candidate bisimulation between *Spec* and *Impl* is as follows:

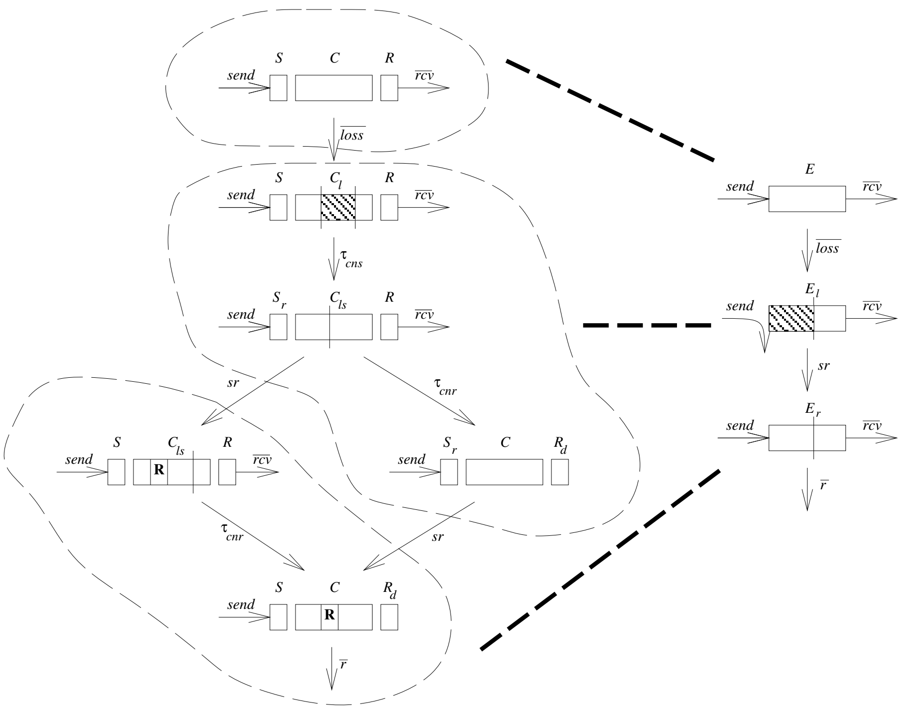

Figure 4: A Proposed Bisimulation

- All pre-loss states having the same messages buffered are related.
- All post-loss states before the "start recovery" message having the same messages buffered for transmission are related.
- All states after the "start recovery" message but before the "notify recovery" message are related if the same post-loss messages are buffered.

The proposed relation is sketched in Figure 4. The thick dotted lines in the figure relate implementation states (grouped with thin dotted lines) to specification states. The precise matching of states follows the informal constraints just given on buffer contents.

A problem with this proposed bisimulation is that certain losses possible in the specification are not possible in the implementation. For example, the transition $E(\langle m \rangle) \stackrel{\overline{loss}}{\rightarrow} E_l(\langle\rangle)$ of *Spec* cannot be matched by $SCR'(\langle m \rangle)$. The specification allows all buffered messages to be lost, while the implementation only allows messages buffered by the medium to be lost — not messages buffered by the sending or receiving machines.

Another problem with the proposed bisimulation concerns buffered messages remaining after a $\overline{loss}$ action. Such messages can safely be delivered provided they were sent before the lost messages. In *Spec*, these "pre-loss" messages can be delivered until the action *sr* occurs. In *Impl*, pre-loss messages can be delivered until either *sr* occurs *or* until process C notifies the receiver process R that a loss has occurred (via the action *cnr*). Thus, *Impl* can reach a state $S_rCR_d(s)$ in which *sr* has not yet occurred but pre-loss messages cannot be delivered.

Another way of understanding these problems is by considering the groups of implementation nodes in Figure 4. The first problem can be regarded as that is that not all states of the top group are observation equivalent. For example, $SCR(\langle m \rangle) \not\approx SCR'(\langle m \rangle)$. Similarly, for the second problem, $Sri_{ls}R(\langle m \rangle, \langle\rangle) \not\approx S_rCR_d(\langle\rangle)$.

These two obstacles to showing that $Spec \approx Impl$ might be seen as irrelevant. The action $\overline{loss}$ was included in the specification because it was expected that the communication medium of an implementation might suffer occasional losses. An implementation that cannot suffer losses is acceptable, however, as

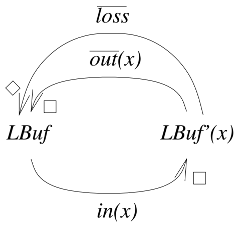

Figure 5: Specification of a Lossy Buffer

are implementations that can only suffer some of the losses "allowed" in the specification. The important point is that if a loss does occurs, then recovery takes place as specified.

A similar argument could be made for the delivery of pre-loss messages. If the recovery procedure lasts only a short time, the delivery of pre-loss messages may be unimportant. However, no harm is done by delivering pre-loss messages, as they will help keep the set of data stored in the receiver timely until recovery. There is no reason not to "allow" such actions in the specification.

A third difference in behaviour between *Spec* and *Impl* is that *Impl* can perform a loss action while the recovery procedure is underway, while *Spec* cannot. Losses that occur during recovery are important and have been analysed as part of the system design, but in this paper we will not consider them. We could modify the definition of process *Impl* to make losses impossible during recovery, but instead we will simply not examine losses that occur during recovery.

5 Modal Process Logic

In this section we give a brief description of modal process logic. A complete account can be found in [6].

Modal process logic formalises the idea of specifications that allow certain actions, and require others. An implementation of such a specification must perform all the actions required in the specification, but may perform others provided they are allowed by the specification.

As a simple example consider the specification of a lossy, single-celled buffer. A modal transition system for such a buffer is shown in Figure 5. The transitions labelled with $\Diamond$ are *admissible* transitions, which may be performed by an implementation; those labelled with $\Box$ are *necessary* transitions, which must be performed by an implementation. Thus, an implementation of *LBuf* must be able to accept a message, because the transition $in(x)$ is a necessary one. However, after accepting a message, an implementation need not be capable of losing it, because the $\overline{loss}$ action is an admissible one.

Modal process logic is a notation for describing such specifications. The form of modal process logic that we use here is just like CCS except that two kinds of action prefixing are possible, reflecting the two kinds of transitions in a modal transition system. The syntax of specifications is as follows:

$$F ::= 0 \mid a_{\Box}.F \mid a_{\Diamond}.F \mid F_1 + F_2 \mid F_1 \mid F_2 \mid F[f] \mid F \backslash L \mid$$

Specifications, like CCS processes, can also be defined recursively using constants. The set of all specifications is denoted by $\mathcal{S}$. The lossy buffer example expressed in modal process logic is as follows:

$$\begin{aligned} LBuf &\stackrel{\text{def}}{=} in_{\Box}(x).LBuf'(x) \\ LBuf' &\stackrel{\text{def}}{=} \overline{loss}_{\Diamond}.LBuf + \overline{out}_{\Box}(x).LBuf \end{aligned}$$

The meaning of specifications is given as a *modal transition system* $(\mathcal{S}, Act, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$, where $\longrightarrow_{\Box}, \longrightarrow_{\Diamond} \subseteq \mathcal{S} \times Act \times \mathcal{S}$ are modal transition relations. The idea that an action that is necessary is also admissible is reflected by the assumption that $\longrightarrow_{\Box} \subseteq \longrightarrow_{\Diamond}$. In other words, $S \xrightarrow{\alpha}_{\Box} S'$ implies $S \xrightarrow{\alpha}_{\Diamond} S'$. The relations $\longrightarrow_{\Diamond}, \longrightarrow_{\Box}$ are defined to be the least relations satisfying the following (the subscript m

stands for either $\Diamond$ or $\Box$):

$$
\begin{array}{lll}
\Diamond & : & a_{\Diamond}.F \xrightarrow{a}_{\Diamond} F \\
\Box & : & a_{\Box}.F \xrightarrow{a}_{\Diamond} F \\
 & & a_{\Box}.F \xrightarrow{a}_{\Box} F \\
+ & : & \dfrac{F_1 \xrightarrow{\alpha}_m F_1'}{F_1 + F_2 \xrightarrow{\alpha}_m F_1'} \qquad \dfrac{F_2 \xrightarrow{\alpha}_m F_2'}{F_1 + F_2 \xrightarrow{\alpha}_m F_2'} \\
\mid & : & \dfrac{F_1 \xrightarrow{\alpha}_m F_1'}{F_1|F_2 \xrightarrow{\alpha}_m F_1'|F_2} \qquad \dfrac{F_2 \xrightarrow{\alpha}_m F_2'}{F_1|F_2 \xrightarrow{\alpha}_m F_1|F_2'} \qquad \dfrac{F_1 \xrightarrow{l}_m F_1',\ F_2 \xrightarrow{\overline{l}}_m F_2'}{F_1|F_2 \xrightarrow{\tau}_m F_1'|F_2'} \\
\backslash & : & \dfrac{F \xrightarrow{\alpha}_m F'}{F\backslash L \xrightarrow{\alpha}_m F'\backslash L} \quad \alpha \notin L \cup \overline{L} \\
f & : & \dfrac{F \xrightarrow{\alpha}_m F'}{F[f] \xrightarrow{f(\alpha)}_m F'[f]} \\
\text{def} & : & \dfrac{P \xrightarrow{\alpha}_m P'}{A \xrightarrow{\alpha}_m P'} \quad A \stackrel{\text{def}}{=} P
\end{array}
$$

A specification is a *refinement* of another specification S if every necessary action of S can be matched by a necessary action of the refinement, and if every admissible action of the refinement can be matched by an admissible action of S. Just as observation equivalence of CCS processes was defined via a bisimulation relation, the refinement of specifications is defined via a refinement relation. The weak modal transition $P \stackrel{\alpha}{\Longrightarrow}_{\Box} P'$ is defined as $P {\Longrightarrow_{\Box}} \xrightarrow{\alpha}_{\Box} {\Longrightarrow_{\Box}} P'$, where $P \Longrightarrow_{\Box} P'$ stands for $P(\xrightarrow{\tau}_{\Box})^* P'$. The transition $P \stackrel{\alpha}{\Longrightarrow}_{\Diamond} P'$ is defined similarly.

Definition 2 *A* weak refinement $\mathcal{R}$ *is a binary relation on* $\mathcal{S}$ *such that* $(S, T) \in \mathcal{R}$ *implies, for all* $\alpha \in Act$*:*

1. *Whenever* $S \xrightarrow{\alpha}_{\Diamond} S'$*, then* $T \stackrel{\hat{\alpha}}{\Longrightarrow}_{\Diamond} T'$ *for some* T' *such that* $(S', T') \in \mathcal{R}$*,*
2. *Whenever* $T \xrightarrow{\alpha}_{\Box} T'$*, then* $S \stackrel{\hat{\alpha}}{\Longrightarrow}_{\Box} S'$ *for some* S' *such that* $(S', T') \in \mathcal{R}$*.*

Specification S *is a refinement of* T*, written* $S \trianglelefteq T$*, if* (S, T) *belongs to some weak refinement relation* $\mathcal{R}$*.*

An *implementation* is a special kind of specification in which all transitions are necessary ones. CCS processes can be regarded as implementations by deriving, from the labelled transition system $(\mathcal{P}, Act, \rightarrow)$ that defines the behaviour of CCS processes, a modal transition system $(\mathcal{P}, Act, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$, with $\longrightarrow_{\Box} = \longrightarrow_{\Diamond} = \rightarrow$. If a process P is a refinement of S, then we say that P is an implementation of S.

Note that for two processes, the notion of refinement coincides with that of bisimulation. So, for processes P and Q, $P \approx Q implies P \trianglelefteq Q \wedge P \trianglelefteq Q$. Another useful property of refinement is that it is preserved by the static operators $|$, $\backslash L$, and $[f]$. For example, $S \trianglelefteq T implies S \mid U \trianglelefteq T \mid U$.

6 Applying Refinement

As explained in Section 4, the specification and implementation cannot be shown to be bisimilar for two reasons:

- the implementation cannot match all message losses of the specification
- the implementation can silently decide to stop delivering pre-loss messages

Now the main issue is whether the specification can be described in modal process logic so that the implementation is valid in the formal sense of the last section. In particular, can certain transitions of the specification be identified as admissible transitions such that a refinement can be found? Recalling the discussion of Section 4, an implementation is acceptable if it never loses messages, and delivery of pre-loss messages is not vital. Therefore, a natural idea is to mark all $\overline{loss}$ transitions and all pre-loss $\overline{rcv}$ transitions as admissible.

The revised specification written in modal process logic is given below. Modal process logic is flexible enough to express that $\overline{rcv}$ actions are admissible specifically for the case of pre-loss messages.

$$
\begin{array}{rcl}
Spec' & \stackrel{\mathrm{def}}{=} & E(\langle\rangle) \\
E(s) & \stackrel{\mathrm{def}}{=} & send(m)_{\Box}.E(s\langle m\rangle) + \\
 & & (\text{if } s \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s)_{\Box}.E(\mathbf{tl}\, s)) + \\
 & & \sum_{s_1 \in \mathbf{prefix}\, s} \overline{loss}_{\Diamond}.E_l(s_1) \\
E_l(s) & \stackrel{\mathrm{def}}{=} & send(m)_{\Box}.E_l(s) + \\
 & & (\text{if } s_1 \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s)_{\Diamond}.E_l(\mathbf{tl}\, s) + \\
 & & sr_{\Box}.E_r(s, \langle\rangle) \\
E_r(s_1, s_2) & \stackrel{\mathrm{def}}{=} & send(m)_{\Box}.E_r(s_1, s_2\langle m\rangle) + \\
 & & (\text{if } s_1 \neq \langle\rangle \text{ then } \overline{rcv}(\mathbf{hd}\, s_1)_{\Diamond}.E_r(\mathbf{tl}\, s_1, s_2)) + \\
 & & \overline{r}_{\Box}.E(s_2)
\end{array}
$$

To show that *Impl* is a refinement of $Spec'$, i.e., that $Impl \trianglelefteq Spec'$, we can produce a refinement $\mathcal{R}$ containing the pair $(Impl, Spec)$. Producing such a relation and checking that it is indeed a refinement by hand is tedious, however. The same problem occurs with bisimulations. Recall the bisimulation $\mathcal{B}$ given in the example of Section 4. Proving that $\mathcal{B}$ is a bisimulation is slightly tedious because $Buf'(x) \frown Buf$ and $Buf \frown Buf'(x)$ are both contained in $\mathcal{B}$, although it seems obvious that these two processes are observation equivalent. Similarly, in other bisimulations we might find both pair $(P, Q \mid 0)$ and (P, Q), although we know that $Q \mid 0 \approx Q$. A technique enabling simpler bisimulation proofs is based on expansion relations [2].

Definition 3 *An* expansion $\mathcal{E}$ *is a binary relation on* $\mathcal{P}$ *such that* $(P, Q) \in \mathcal{E}$ *implies, for all* $\alpha \in Act$*:*

1. *Whenever* $P \xrightarrow{\alpha} P'$*, then* $Q \stackrel{\alpha}{\Rightarrow} Q'$ *for some* Q' *such that* $(P', Q') \in \mathcal{E}$*,*

2. *Whenever* $Q \xrightarrow{\alpha} Q'$*, then* $P \xrightarrow{\hat{\alpha}} P'$ *for some* P' *such that* $(P', Q') \in \mathcal{E}$*.*

Process Q expands P*, written* $P \lesssim Q$*, if* (P, Q) *belongs to some expansion* $\mathcal{E}$*.*

The relation $\lesssim$ is a preorder, not an equivalence, and is strictly stronger than $\approx$. Intuitively, $P \lesssim Q$ holds if P and Q are observation equivalent, and P is at least as "fast" as Q. Like $\approx$, $\lesssim$ preserves the static operators of CCS. Expansions are important because of an "up-to" technique they allow:

Definition 4 *A binary relation* $\mathcal{E}$ *on* $\mathcal{P}$ *is an* expansion up to $\lesssim$ *if* $(P, Q) \in \mathcal{E}$ *implies, for all* $\alpha \in Act$*:*

1. *Whenever* $P \xrightarrow{\alpha} P'$*, then* $Q \stackrel{\alpha}{\Rightarrow} Q'$ *for some* Q' *such that* $(P', Q') \in \sim \circ \mathcal{E} \circ \lesssim$*,*

2. *Whenever* $Q \xrightarrow{\alpha} Q'$*, then* $P \xrightarrow{\hat{\alpha}} P'$ *for some* P' *such that* $(P', Q') \in \lesssim \circ \mathcal{E} \circ \lesssim$*.*

Theorem 1 *If* $\mathcal{E}$ *is an expansion up to* $\lesssim$*, then* $\mathcal{E} \subseteq \lesssim$*.*

Proof. See [2].

The theorem says that, if (P, Q) belongs to an expansion up to $\lesssim$, then $P \lesssim Q$.

Returning again to our buffer example, one can show that the following relation $\mathcal{E}$ is an expansion up to $\lesssim$:

$$\begin{aligned}\mathcal{E} \stackrel{\text{def}}{=} \; & \{(Buf_2(\langle\rangle), BufPair),\\ & (Buf_2(\langle x\rangle), Buf \frown Buf'(x)),\\ & (Buf_2(\langle x, y\rangle), Buf'(y) \frown Buf'(x))\}\end{aligned}$$

The proof that $\mathcal{E}$ is an expansion up to $\lesssim$ has two parts. In the first, we need to show that $Buf \frown Buf'(x) \lesssim Buf'(x) \frown Buf$. In the second, this fact is used when checking the relation $\mathcal{E}$. Once it has been shown that $\mathcal{E}$ is an expansion up to $\lesssim$, we know that $Buf_2(\langle\rangle) \lesssim BufPair$, and since $\lesssim \subseteq \approx$, we have that $Buf_2(\langle\rangle)$ and $BufPair$ are observation equivalent.

The expansion up to $\lesssim$ technique has two advantages over the basic bisimulation proof technique. First, an expansion up to $\lesssim$ may often be a much smaller relation than the corresponding bisimulation, so that less work is needed in verifying the relation. Second, the expansion up to $\lesssim$ technique provides a modularisation of the proof work: in one step the supporting facts are shown, in the other the relation itself is checked.

Applying the idea of expansion to refinement, we get two new kinds of relations:

Definition 5 *An* e-refinement $\mathcal{R}$ *is a binary relation on* $\mathcal{S}$ *such that* $(S, T) \in \mathcal{R}$ *implies, for all* $\alpha \in Act$*:*

1. *Whenever* $S \xrightarrow{\alpha}_{\Diamond} S'$, *then* $T \xrightarrow{\hat{\alpha}}_{\Diamond} T'$ *for some* T' *such that* $(S', T') \in \mathcal{R}$,
2. *Whenever* $T \xrightarrow{\alpha}_{\Box} T'$, *then* $S \stackrel{\alpha}{\Longrightarrow}_{\Box} S'$ *for some* S' *such that* $(S', T') \in \mathcal{R}$.

Process S e-refines T, *written* $S \leqslant T$,*if* (S, T) *belongs to some expansion* $\mathcal{R}$.

Definition 6 *A binary relation* R *on* $\mathcal{S}$ *is an* e-refinement up to $\leqslant$ *if* $(S, T) \in \mathcal{R}$ *implies, for all* $a \in Act$*:*

1. *Whenever* $S \xrightarrow{\alpha}_{\Diamond} S'$, *then* $T \xrightarrow{\hat{\alpha}}_{\Diamond} T'$ *for some* T' *such that* $(S', T') \in \, \leqslant \circ \mathcal{E} \circ \leqslant$,
2. *Whenever* $T \xrightarrow{\alpha}_{\Box} T'$, *then* $S \stackrel{\alpha}{\Longrightarrow}_{\Box} S'$ *for some* S' *such that* $(S', T') \in \, \leqslant \circ \mathcal{E} \circ \sim$.

Theorem 2 $\leqslant \, \subset \, \trianglelefteq$

Proof. The inclusion follows trivially from the definition of weak refinement, since $T \xrightarrow{\hat{\alpha}}_{\Diamond} T'$ implies $T \stackrel{\hat{\alpha}}{\Longrightarrow}_{\Diamond} T'$, and $S \stackrel{\alpha}{\Longrightarrow}_{\Box} S'$ implies $S \stackrel{\hat{\alpha}}{\Longrightarrow}_{\Box} S'$. For strictness, we have that $P \not\leqslant \tau.P$ and $P \trianglelefteq \tau.P$. $\Box$

Theorem 3 *The static operators of modal process logic preserve* $\leqslant$.

Proof. This is easily shown by exhibiting e-refinements. For example, to show that $|$ preserves $\leqslant$, show that $\mathcal{R} \stackrel{\text{def}}{=} \{(S_1 \mid T, S_2 \mid T) : S_1 \leqslant S_2\}$ is an e-refinement. $\Box$

Theorem 4 *If* $\mathcal{R}$ *is an e-refinement up to* $\leqslant$, *then* $\mathcal{R} \subseteq \, \leqslant$.

Proof. The proof is similar to the proof of Proposition 4.6 in [7]. First one shows by diagram chasing that, if $\mathcal{R}$ is an e-refinement up to $\leqslant$, then $\leqslant \circ \mathcal{R} \circ \leqslant$ is an e-refinement. Then, since $\leqslant$ is the largest e-refinement, we have that $\leqslant \circ \mathcal{R} \circ \leqslant \, \subseteq \, \leqslant$, and since the identity relation *Id* is contained in $\leqslant$, we have that $\mathcal{R} \subseteq \, \leqslant$. $\Box$

Intuitively, $S \leqslant T$ holds if S is a refinement of specification T that is slower than T. In practice, implementations are often more concurrent than specifications, and thereby slower because of additional τ actions. Note that, for processes P and Q, $P \leqslant Q$ coincides with $Q \lesssim P$.

Our proof that $Impl \trianglelefteq Spec'$ is organised as two parts. The first part is an e-refinement $\mathcal{R}$ containing the pair $(Impl, Spec')$. The second part is a group of lemmas stating refinement relations between states of the implementation, analogous to the relation $Buf \frown Buf'(x) \lesssim Buf'(x) \frown Buf$ used in the expansion example.

Theorem 5 *$Impl \trianglelefteq Spec'$.*

Proof. We will show that the following relation $\mathcal{R}$ is an e-refinement up to $\leqslant$. The pairs of $\mathcal{R}$ are as follows, for all sequences $s, s', s_1, s_2, and s_3$:

$$
\begin{aligned}
&(SCR(s) &,\quad& E(s)) && (1)\\
&(SCR'(s) &,\quad& E(s)) && (2)\\
&(SC_lR(s_1, s_2) &,\quad& E_l(s_1)) && (3)\\
&(S_rCR_d(s) &,\quad& E_l(s')) && (4)\\
&(SC_{ls}R(s_1, s_2\langle \mathbf{R}\rangle s_3) &,\quad& E_r(s_1, s_3)) && (5)\\
&(SCR_d(s_1\langle \mathbf{R}\rangle s_2) &,\quad& E_r(s', s_2)) && (6)\\
&(SC_lR'(s_1, s_2) &,\quad& E_l(s_1)) && (7)\\
&(SCR'_d(s_1\langle \mathbf{R}\rangle s_2) &,\quad& E_r(s', s_2)) && (8)\\
&(S_rC_{ls}R(s_1, s_2) &,\quad& E_l(s_1)) && (9)\\
&(S_rC_{ls}R'(s_1, s_2) &,\quad& E_l(s_1)) && (10)\\
&(SC_{ls}R'(s_1, s_2\langle \mathbf{R}\rangle s_3) &,\quad& E_r(s_1, s_3)) && (11)
\end{aligned}
$$

If $\mathcal{R}$ can be shown to be an e-refinement up to $\leqslant$, then $Impl \leqslant Spec'$, and so $Impl \trianglelefteq Spec$.

Before proving that $\mathcal{R}$ is indeed an e-refinement, we will present and prove the supporting lemmas (for all sequences s, s_1, s_2, and all messages m):

$$
\begin{aligned}
(S'(m) \mid C(s))\backslash L &\leqslant (S \mid C(s\langle m\rangle))\backslash L\\
(S'(m) \mid C_l(s))\backslash L &\leqslant (S \mid C_l(s\langle m\rangle))\backslash L\\
(S'(m) \mid C_{ls}(s_1, s_2))\backslash L &\leqslant (S \mid C_{ls}(s_1, s_2\langle m\rangle))\backslash L\\
(S'_r(m) \mid C(s))\backslash L &\leqslant (S_r \mid C(s\langle m\rangle))\backslash L\\
(S'_r(m) \mid C_l(s_1, s_2))\backslash L &\leqslant (S_r \mid C_l(s_1, s_2\langle m\rangle))\backslash L\\
(S''_r \mid C_{ls}(s_1, s_2))\backslash L &\leqslant (S \mid C_{ls}(s_1, s_2\langle \mathbf{R}\rangle))\backslash L\\
(S''_r \mid C(s_1, s_2))\backslash L &\leqslant (S \mid C(s\langle \mathbf{R}\rangle))\backslash L\\
(S'_r(m) \mid C_{ls}(s_1, s_2))\backslash L &\leqslant (S_r \mid C_{ls}(s_1, s_2\langle m\rangle))\backslash L
\end{aligned}
$$

where $L \stackrel{\text{def}}{=} \{cns, csend\}$.

Proof. (of lemmas). Since the lemmas concern only processes, they are in effect expansions, and are proved by exhibiting expansions up to $\lesssim$. For example, we present a relation $\mathcal{E}$, containing the following pairs (for all sequences s, s_1, s_2, and all messages m):

$$
\begin{aligned}
((S \mid C(s\langle m\rangle))\backslash L &\quad,\quad ((S'(m) \mid C(s))\backslash L)\\
((S \mid C_l(s\langle m\rangle))\backslash L &\quad,\quad ((S'(m) \mid C_l(s))\backslash L)\\
((S \mid C_{ls}(s_1, s_2\langle m\rangle))\backslash L &\quad,\quad ((S'(m) \mid C_{ls}(s_1, s_2))\backslash L)
\end{aligned}
$$

It is easy to show that $\mathcal{E}$ is an expansion up to $\lesssim$, which proves the first three lemmas. The other lemmas are proved similarly. □

In applying the lemmas, we use the preservation of $\leqslant$ by the static combinators of CCS. For example, from the first lemma it follows that $S'CR(s\langle m\rangle) \leqslant SCR(s\langle m\rangle)$.

We will not give full details of the proof that $\mathcal{R}$ is an e-refinement up to $\leqslant$. To give a flavour of the proof we show the details involved in checking pair 4 of $\mathcal{R}$. First we check the admissible transitions that can be performed by $S_rCR_d(s)$(matching actions by $E_l(s')$ are indented):

$$\begin{array}{lll} S_rCR_d(s) & \xrightarrow{send(m)}_{\Diamond} & S'_rCR_d(s\langle m\rangle) \leqslant S_rCR_d(s\langle m\rangle) \\ & & E_l(s') \xrightarrow{send(m)}_{\Diamond} E_l(s') \\ S_rCR_d(s) & \xrightarrow{sr}_{\Diamond} & S''_rCR_d(s\langle m\rangle) \leqslant SCR_d(s\langle \mathbf{R}\rangle) \\ & & E_l(s') \xrightarrow{sr}_{\Diamond} E_r(s', \langle\rangle) \\ S_rCR_d(s) & \xrightarrow{\tau_{crcv}}_{\Diamond} & S_rCR_d(s) \\ & & E_l(s') \xrightarrow{\hat{\tau}}_{\Diamond} E_l(s') \end{array}$$

Each pair of derivatives belongs to the relation $\mathcal{R}$. Observe that two of the lemmas were used as part of the up-to technique, and that the $\overline{loss}$ action that can be performed by $S_rCR_d(s)$ was not checked because only single losses are being considered (as described at the end of Section 4).

Considering now the necessary actions of the specification, we have:

$$\begin{array}{lll} E_l(s') & \xrightarrow{send(m)}_{\Box} & E_l(s') \\ & & S_rCR_d(s) \xrightarrow{send(m)}_{\Box} S'_rCR_d(s\langle m\rangle) \xrightarrow{\tau_{csend}}_{\Box} S_rCR_d(s\langle m\rangle) \\ E_l(s') & \xrightarrow{sr}_{\Box} & E_r(s', \langle\rangle) \\ & & S_rCR_d(s) \xrightarrow{sr}_{\Diamond} S''_rCR_d(s\langle m\rangle) \xrightarrow{\tau_{csend}}_{\Box} SCR_d(s\langle \mathbf{R}\rangle) \end{array}$$

Here the important point is that the loss action of $E_l(s')$ is not checked, because it is an admissible action, which need not be performed by an implementation.

The other pairs of $\mathcal{R}$ are checked in the same way. $\Box$

7 Conclusions

The notion of refinement found in modal process logic is a natural one for this application. Indeed, we sensed that certain actions in a specification should be thought of as admissible even before knowing of modal process logic. Furthermore, modal process handled two different kinds of issues, that of losses and that of the optional transmission of certain messages.

Most other process refinement relations are less well suited for this problem. As already shown, the observation equivalence relation of CCS is too strong, requiring that the implementation be able to perform all losses possible in the specification. The may preorder relation (also known as trace inclusion) [5] is too weak, allowing the inactive agent 0 as a legal implementation of *any* specification. The bisimulation preorder relation [9, 3] captures a sense of refinement in which an implementation determines states left undefined in the specification. In the failure recovery protocol, however, we do not want to leave unspecified the effect of a loss. Finally, Holmström's refinement calculus [4] is very flexible and could probably capture the kind of looseness we want in the specification. However, it is able to capture only safety properties, and its generality might lead to a more involved refinement proof than the one we have given.

We have not described the changes that had to be made to the implementation to make the refinement proof work. Some of these changes were corrections to errors that we were glad to uncover; others were demanded by the refinement relation but not necessary for proper operation of the protocol. For example, process R of the implementation was originally defined as follows:

$$
\begin{aligned}
R &\stackrel{\mathrm{def}}{=} crcv(m).R'(m) + cnr.R_d \\
R'(m) &\stackrel{\mathrm{def}}{=} \overline{rcv}(m).R \\
R_d &\stackrel{\mathrm{def}}{=} crcv(m).(\text{if } m = \mathbf{R} \text{ then } \overline{r}.R \text{ else } R_d)
\end{aligned}
$$

The refinement proof failed because in pair $(SC_{ls}\, R'(s_1, s_2\langle\mathbf{R}\rangle s_3), E_r(s_1, s_3))$ of the refinement relation the specification could perform action $\overline{r}$ as a necessary action, but the implementation, with its component R in state R', needed to perform $\overline{rcv}$ before eventually performing $\overline{r}$. To fix the problem, the definition of $R'(m)$ was changed to the following:

$$
R'(m) \stackrel{\mathrm{def}}{=} \overline{rcv}(m).R + cnr.R_d
$$

The change allows R' to synchronise on cnr, and thus to eventually perform $\overline{r}$. However, the original $R'(m)$ is only unsatisfactory in an environment that can refuse to synchronise with $\overline{rcv}(m)$, blocking $R'(m)$ from reaching state R_d. In reality, it is expected that output actions such as $\overline{rcv}(m)$ can always be performed.

Our refinement proof was carried out completely by hand. We do not hope for a fully automatic proof of refinement, since the specification and implementation have an infinite state space, but would like our proof to be machine-checked. Because much of the tedium of the refinement proof is in finding the actions that can be performed by the specification and implementation, even simple tool support would be helpful.

A future paper will report on the study of the more realistic protocol actually used which handles losses during recovery.

Acknowledgements

Special thanks to Anthony Hall of Praxis Systems for suggesting this problem, for providing the CCS specification and implementation on which the versions in this paper are closely based, and for many helpful discussions. Thanks also to Stuart Matthews of Praxis Systems for help in understanding the problem. This work was supported by SERC/IED project 1224, "Mathematically-Proven Safety Systems".

References

[1] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, 1989.

[2] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. In *Proceedings of Theoretical Aspects of Computer Science*. Springer Verlag, 1991. Lecture Notes in Computer Science, volume 526. To appear in Acta Informatica.

[3] Rance Cleaveland and Bernhard Steffen. When is "partial" adequate? A logic-based proof technique using partial specifications. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1990.

[4] Sören Hölmstrom. A refinement calculus for specifications in Hennessy-Milner Logic with recursion. *Formal Aspects of Computing*, 1:242–272, 1989.

[5] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 239–247, 1989.

[6] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988.

[7] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[8] Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In *Proceedings of CONCUR '92*, 1992.

[9] D. J. Walker. Bisimulations and divergence. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988.