An Industrial Application of Modal Process Logic

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

Abstract

Modal process logic is an extension of CCS that allows for more expressive specifications. We show how modal process logic was successfully applied in the development of a failure recovery protocol for an air-traffic information system now in service at Heathrow airport. Two example systems are used to show that CCS itself was not suitable for this application.

1 Introduction

Concurrency theory has become an important tool in the analysis of reactive and distributed systems. For example, equivalence and preorder relations from concurrency theory have been used to formalise what it means for one system to be a correct "implementation" or "refinement" of another. Examples can be found in [11, 12].

This paper describes the application of a recently-developed refinement relation to the development of an air-traffic information system now in service at Heathrow Airport, London. Part of the system is a protocol for recovery from communication failures. The designers used CCS to describe both the intended service and the implementation of the protocol. However, using the standard behavioural equivalence of CCS as a refinement relation, they were unable to show that the protocol satisfied its service specification. The main obstacle was that some of the failures mentioned in the specification could not occur in the implementation.

This problem did not disturb the designers. The intention of their specification was to show how failures should be handled, not to show that failures must occur. We solved the problem by using *modal process logic* (MPL), an extension of CCS. Actions in an MPL specification are classified as either necessary or admissible. The refinement relation of MPL requires only that necessary actions be implemented. The use of MPL led to the discovery of a serious error in the protocol that was corrected before detailed implementation began.

To avoid obscuring the main ideas with details of the protocol, we begin by presenting two small examples. We describe the problems faced if the common behavioural equivalences of CCS are used as refinement relations in the examples, and show that these problems persist even if other common refinement relations are used. Then we describe MPL and show how it solves the problem. Finally we describe the recovery protocol and show how MPL was applied to it.

2 Example – Lossy Buffers

Suppose we want to specify a buffer that can detect the loss of messages. The buffer normally acts like a FIFO queue with capacity n. The buffer may also lose a message, but the loss must be reported by replacing the lost message with a distinguished message. In CCS, such a specification might be written as follows.

$$\begin{array}{rcl} \mathit{LBuf}_n(s) & \stackrel{\text{def}}{=} & \text{if } \mathrm{len}(s) < n \text{ then } \mathit{in}(x).\mathit{LBuf}_n(s < x >) + \\ & & \text{if } \mathrm{len}(s) > 0 \text{ then } \overline{\mathit{out}}(\mathrm{hd}(s)).\mathit{LBuf}_n(\mathrm{tl}(s)) + \\ & & \sum_{s=s_1\langle x\rangle s_2} \mathit{loss}.\mathit{LBuf}_n(s_1 < \mathbf{l} > s_2) \end{array}$$

The buffered values are stored in sequence s. We write $< x_1, \ldots, x_n >$ for the sequence containing elements $x_1, \ldots, x_n$, and $s_1 s_2$ for sequence s_2 appended to sequence s_1. We write len(s), hd(s), and tl(s) respectively for the length of sequence s, its first element, and all but its first element. The last line of the definition shows that a lost message is replaced by the distinguished constant $\mathbf{l}$.

Now suppose we want to implement such a buffer with capacity two, and that the implementation is to be distributed across two machines. On each machine runs a *cell*, which buffers just one value. Suppose also that one machine is highly-reliable, while the other can be expected to occasionally suffer a loss. We model the system as two agents: one models a reliable cell; the other models a lossy cell. The reliable cell is modelled as:

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

The lossy cell is modelled as:

$$\begin{array}{rcl} \mathit{LBuf} & \stackrel{\text{def}}{=} & \mathit{in}(x).\mathit{LBuf}'(x) \\ \mathit{LBuf}'(x) & \stackrel{\text{def}}{=} & \overline{\mathit{out}}(x).\mathit{LBuf} + \mathit{loss}.\mathit{LBuf}'(\mathbf{l}) \end{array}$$

To connect two cells, we define the linking operator $\frown$ using the composition and restriction operators of CCS.

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

The action a synchronises an output of the first cell with an input of the second. We obtain the distributed buffer by linking a reliable and a lossy cell.

$$DBuf_2 \stackrel{\text{def}}{=} LBuf \frown Buf$$

In CCS we usually say that one agent implements another if the two are observation equivalent. But here the implementation and specification are not observation equivalent, because the specification can perform *loss* actions that cannot be matched by the implementation. For example, suppose that the specification is holding one value and the implementation is holding the same value in its reliable cell. Then the specification can perform a *loss* action that cannot be matched by the implementation.

If we use observation equivalence as our refinement relation we cannot specify that an implementation may lose messages without specifying that an implementation must be able to lose messages. Here we want to allow as implementations those agents that never lose messages. The reason for including losses in the specification is to show that losses that occur must be reported.

3 Example – Buffers with Updating

Our next example is a buffer that passes messages containing timely information from a source to a destination. Each message is either a normal message or a distinguished *update* message, which indicates that following messages contain more recent information. If the buffer holds no update messages then it acts like an ordinary FIFO queue. On arrival of an update message, the normal messages received earlier can optionally be passed along. However, as soon as the next normal message is received all messages are flushed from the buffer. An application of such a buffer might be in the delivery of news bulletins. Here an update message would signal the arrival of more recent news.

A CCS description of such a buffer is the following:

$$\begin{array}{rcl}
R(s) & \stackrel{\text{def}}{=} & in(m).\text{if } m = \mathit{update} \text{ then } R'(s) \text{ else } R(s < m >) \\
& + & \text{if } \text{len}(s) > 0 \text{ then } \overline{out}(\text{hd}(s)).R(\text{tl}(s)) \\
R'(s) & \stackrel{\text{def}}{=} & in(m).\text{if } m = \mathit{update} \text{ then } R'(s) \text{ else } R(< m >) \\
& + & \text{if } \text{len}(s) > 0 \text{ then } \overline{out}(\text{hd}(s)).R'(\text{tl}(s))
\end{array}$$

The two parts of the specification are nearly the same. The difference is that R' will dispose of all buffered values after a non-*update* message is received.

Again, the CCS agent does not capture what is intended in the informal specification. We want to allow implementations like the following one, which disposes of the buffer as soon as an update message arrives.

$$\begin{array}{rcl}
RI(s) & \stackrel{\text{def}}{=} & in(m).\text{if } m = \mathit{update} \text{ then } RI(< >) \text{ else } RI(s < m >) \\
& + & \text{if } \text{len}(s) > 0 \text{ then } \overline{out}(\text{hd}(s)).RI(\text{tl}(s))
\end{array}$$

However, agents $R(s)$ and $RI(s)$ are not observation equivalent.

The problem illustrated by these examples cannot be solved by using another existing relation on CCS agents as the refinement relation. For example, *trace inclusion* is commonly used as a refinement relation [6, 3]. Here we say an agent P implements agent Q if every sequence of actions that can be performed by P can also be performed by Q. Trace inclusion is unsuitable because it treats all actions uniformly. In the first example we want to distinguish between actions like *in* that must be implemented, and actions like *loss* that need not always be implemented. Any other refinement relation that handles all actions uniformly (e.g. the divergence preorder [13] or failures preorder [6]) will be unsuitable for the same reason.

Nor can this problem be solved by defining a new refinement relation in which certain actions are treated specially using process operators like hiding and restriction. In the first example we could say that an agent P is an implementation of agent $LBuf_2(<>)$ if:

1. all traces of P are traces of $LBuf_2(<>)$, and
2. all traces of $LBuf_2(<>)$ (with its *loss* actions restricted) are traces of P (with its *loss* actions restricted).

Here we take traces to be weak traces, where τ actions are ignored. According to this relation, $DBuf_2$ is an implementation of $LBuf_2(<>)$. However, this relation is too weak as it allows implementations that terminate after performing action *loss*, and thus never report the loss.

4 Modal Process Logic

MPL is an extension of CCS that allows more expressive specifications. In MPL prefixed actions are either *necessary* or *admissible*. The difference between them is important in refinement. Informally, each necessary action of the specification must be matched by a necessary action of the implementation, and each admissible action of the implementation must be matched by an admissible action of the specification. Thus, admissible actions of the specification need not be implemented.

We now briefly review MPL; details can be found in [7, 8]. In MPL we speak of *specifications*, not agents. The syntax of specifications is as follows:

$$S ::= 0 \bigm| \alpha_{\Box}.S \bigm| \alpha_{\Diamond}.S \bigm| S_1 + S_2 \bigm| S_1 \mid S_2 \bigm| S\backslash L \bigm| S[f]$$

Specifications, like CCS agents, can be defined recursively using constants. The set of all specifications is denoted by $\mathcal{S}$.

The meaning of specifications is given as a *modal transition system*, which is like the labelled transition system of CCS but with two types of arrows between specifications. Formally a modal transition system has the form $(\mathcal{S}, Act, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$, where $\longrightarrow_{\Box}, \longrightarrow_{\Diamond} \subseteq \mathcal{S} \times Act \times \mathcal{S}$ are modal transition relations. The semantic rules that define the transition relations are closely related to those of CCS, given in Appendix A. The rule for the prefix operator '.' of CCS is replaced by the rules

$$\Diamond \quad : \qquad \alpha_\Diamond.F \xrightarrow{\alpha}_\Diamond F$$

$$\Box \quad : \qquad \alpha_\Box.F \xrightarrow{\alpha}_\Diamond F \qquad\qquad \alpha_\Box.F \xrightarrow{\alpha}_\Box F$$

The $\Box$ rules shows that every necessary action is also an admissible one. In other words, $S \xrightarrow{\alpha}_\Box S'$ implies $S \xrightarrow{\alpha}_\Diamond S'$.

The other semantic rules of MPL are obtained by replacing each CCS rule (except the one for '.') by two new rules, one in which all $\longrightarrow$ transitions of the rule are replaced by $\longrightarrow_\Box$ transitions, and another in which all $\longrightarrow$ transitions are replaced by $\longrightarrow_\Diamond$ transitions.

Just as the equivalence of CCS agents is defined via bisimulation relations, the refinement of specifications is defined via refinement relations. Since we want a refinement relation based on observable transitions, we use a *weak modal transition relation* similar to the weak transition relation of CCS. By definition $S \overset{\varepsilon}{\Longrightarrow}_\Box S'$ if $S(\xrightarrow{\tau}_\Box)^* S'$, and $S \overset{l}{\Longrightarrow}_\Box S'$ if $S \overset{\varepsilon}{\Longrightarrow}_\Box \circ \overset{l}{\longrightarrow}_\Box \circ \overset{\varepsilon}{\Longrightarrow}_\Box S'$. The $\overset{\alpha}{\Longrightarrow}_\Diamond$ relation is defined similarly.

A binary relation $\mathcal{R}$ on specifications is a *weak refinement* if $(S,T) \in \mathcal{R}$ implies:

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

(The notation $\widehat{\alpha}$ is defined in Appendix A.) Specification S is a *refinement* of T, written $S \trianglelefteq T$, if there exists a weak refinement containing (S,T).

We can treat any CCS agent as a specification by pretending all its actions are necessary. Notice that the refinement relation collapses to observation equivalence when both specifications contain only necessary actions.

5 Revisiting the Examples

We now redefine the specification of a lossy buffer using MPL.

$$\begin{aligned} \mathit{LBuf}_n(s) \;\overset{\text{def}}{=}\; & \text{if len}(s) < n \text{ then } \mathit{in}_\Box(x).\mathit{LBuf}_n(s < x >) + \\ & \text{if len}(s) > 0 \text{ then } \overline{\mathit{out}}_\Box(\text{hd}(s)).\mathit{LBuf}_n(\text{tl}(s)) + \\ & \textstyle\sum_{s=s_1\langle x\rangle s_2} \mathit{loss}_\Diamond.\mathit{LBuf}_n(s_1 < \mathbf{1} > s_2) \end{aligned}$$

The specification states that input and output actions must be implemented, but that loss actions need not be.

The following weak refinement relation shows that a two-celled concurrent buffer with only one lossy cell is an implementation of a 2-capacity lossy buffer i.e., that $\mathit{DBuf}_2 \trianglelefteq \mathit{LBuf}_2(<>)$.

$$\{(\mathit{DBuf}_2 \quad , \quad \mathit{LBuf}_2(<>)),$$

$$
\begin{array}{rcl}
(LBuf'(x) \frown Buf & , & LBuf_2(< x >)), \\
(LBuf \frown Buf'(x) & , & LBuf_2(< x >)), \\
(LBuf'(y) \frown Buf'(x) & , & LBuf_2(< x, y >))\}
\end{array}
$$

To see how the admissible transitions are used in checking a refinement relation, consider the third pair above. The transition

$$LBuf \frown Buf'(x) \xrightarrow{in(y)}_{\Box} LBuf'(y) \frown Buf'(x)$$

is possible for the agent on the left. Since every necessary transition is also an admissible one, this transition must be matched by the specification on the right. It can be, by

$$LBuf_2(< x >) \xrightarrow{in(y)}_{\Diamond} LBuf_2(< x, y >).$$

The derived pair of specifications are themselves in the relation. Now consider transitions of the right-hand specification of the same pair. The transition

$$LBuf_2(< x >) \xrightarrow{loss}_{\Diamond} LBuf_2(< \mathbf{1} >)$$

is an admissible transition, so by the definition of refinement relation it need not be matched by the specification on the left.

The buffer with updating can also be neatly specified using MPL.

$$
\begin{array}{rcl}
R(s) & \stackrel{\text{def}}{=} & in_{\Box}(m).\text{if } m = \mathit{update} \text{ then } R'(s) \text{ else } R(s < m >) \\
 & + & \text{if } \mathrm{len}(s) > 0 \text{ then } \overline{out}_{\Box}(\mathrm{hd}(s)).R(\mathrm{tl}(s)) \\
R'(s) & \stackrel{\text{def}}{=} & in_{\Box}(m).\text{if } m = \mathit{update} \text{ then } R'(s) \text{ else } R(< m >) \\
 & + & \text{if } \mathrm{len}(s) > 0 \text{ then } \overline{out}_{\Diamond}(\mathrm{hd}(s)).R'(\mathrm{tl}(s))
\end{array}
$$

Agent RI can be shown to be an implementation of this specification by the weak refinement relation containing all pairs of the form

$$
\begin{array}{rcl}
(RI(s) & , & R(s)) \\
(RI(<>) & , & R'(s))
\end{array}
$$

6 A Failure Recovery Protocol

We have applied MPL to an aviation control information display system developed by Praxis plc for the British Civil Aviation Authority [5]. The system is regarded as "safety-relevant", since it has an indirect effect on air safety. A main job of the system is to communicate air traffic 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 local data, so it is important that update messages arrive in order and that message loss be detected.

Our concern is the protocol for communication between the central computer and display computers. A low-level communications layer can detect and re-send most lost messages without attention of the application programs on the sending and receiving computers. If a loss occurs that cannot be handled at this level, then the low-level layer 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 programs that use the recovery protocol and the lower system levels that provide the service. The arrows that cross the boundary 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 have been 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.

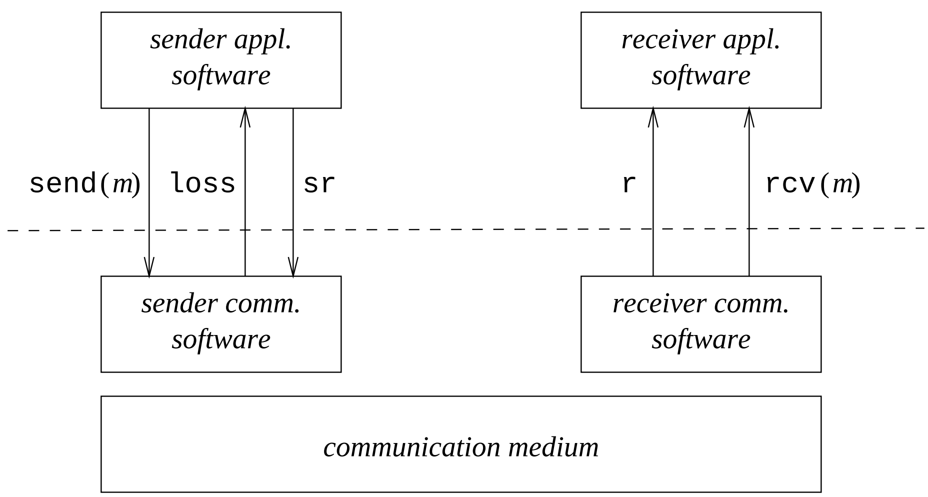

Figure 1: The Recovery Protocol

The 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 start-recovery action must not be delivered until after the notify-recovery action.

7 Specifying the Protocol

We now present a service specification of the protocol in CCS that is based closely on a version developed by Praxis. The following simplifying assumptions are made: there is a single receiving computer, the protocol has unbounded buffering capacity, message delivery is atomic, the only kind of loss is a group of consecutively sent messages, and a second loss cannot occur until after recovery is complete. Figure 2 is the flow diagram for the specification.

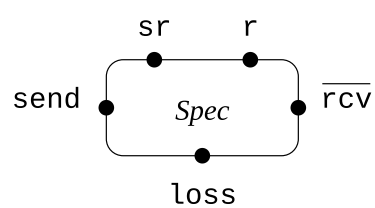

Figure 2: Flow Diagram of the Specification

$$
\begin{array}{rcl}
\mathit{Spec} & \stackrel{\mathrm{def}}{=} & E(<>) \\
E(s) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m).E(s<m>) + \\
 & & (\mathrm{if\,len}(s) > 0 \mathrm{\ then\ } \overline{\mathit{rcv}}(\mathrm{hd}(s)).E(\mathrm{tl}(s))) + \\
 & & \sum_{s=s_1 s_2} \mathit{loss}.E_l(s_1) \\
E_l(s) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m).E_l(s) + \\
 & & (\mathrm{if\,len}(s) > 0 \mathrm{\ then\ } \overline{\mathit{rcv}}(\mathrm{hd}(s)).E_l(\mathrm{tl}(s))) + \\
 & & \mathit{sr}.E_r(s,<>) \\
E_r(s_1,s_2) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m).E_r(s_1,s_2<m>) + \\
 & & (\mathrm{if\,len}(s_1) > 0 \mathrm{\ then\ } \overline{\mathit{rcv}}(\mathrm{hd}(s_1)).E_r(\mathrm{tl}(s_1),s_2)) + \\
 & & r.E(s_2)
\end{array}
$$

The agent has three phases. Initially, it behaves as a simple buffer. After a loss, messages are accepted but discarded, and messages that were sent before the loss are delivered. After the start-recovery message, messages are accepted and stored, but these messages are not delivered until after the recover action occurs, when all pre-loss messages are discarded.

The messages sequences of agent *Spec* have unbounded length, and hence the agent has infinitely many states.

8 Implementing the Protocol

An attempt to implement the protocol and show that it is observation equivalent to the specification will lead to the kinds of problems illustrated by the examples of Sections 2 and 3. Here we describe the implementation as a CCS agent, which again closely follows a Praxis version, and describe the problems of showing observation equivalence. In what follows we refer to the implementation as a *design*, since it lacks details that one would find in an implementation.

As implied by Figure 1, the design of the protocol has three concurrent components: the communications software running on the sending computer, the communications software running on the receiving computer, and the low-level communications layer. These components are represented by the CCS agents S, R, and C. Note that S and R represent communication processes in the recovery protocol, not application programs that use the protocol.

The key idea behind the design is that the only role of the communication layer 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 communication layer treats this message like any other. After the receiver is notified of the loss, it discards all incoming messages until after the distinguished recovery message is received.

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

Agent S is the communications process on the sending machine. It acts as a simple buffer until notification of a loss occurs (via action *cns*). Then, after performing the start-recovery action, it transmits the distinguished recovery message and reverts to its normal behaviour.

$$
\begin{aligned}
C(s) &\stackrel{\text{def}}{=} \mathit{csend}(m).C(s < m >) + \\
&\quad (\text{if } \text{len}(s) > 0 \text{ then } \overline{\mathit{crcv}}(\text{hd}(s)).C(\text{tl}(s))) + \\
&\quad \textstyle\sum_{s=s_1 s' s_2} \mathit{loss}.C_l(s_1, s_2) \\
C_l(s_1, s_2) &\stackrel{\text{def}}{=} \mathit{csend}(m).C_l(s_1, s_2 < m >) + \\
&\quad (\text{if } \text{len}(s_1) > 0 \text{ then } \overline{\mathit{crcv}}(\text{hd}(s_1)).C_l(\text{tl}(s_1), s_2)) + \\
&\quad \overline{\mathit{cns}}.C_{ls}(s_1, s_2) \\
C_{ls}(s_1, s_2) &\stackrel{\text{def}}{=} \mathit{csend}(m).C_{ls}(s_1, s_2 < m >) + \\
&\quad (\text{if } \text{len}(s_1) > 0 \text{ then } \overline{\mathit{crcv}}(\text{hd}(s_1)).C_{ls}(\text{tl}(s_1), s_2)) + \\
&\quad \overline{\mathit{cnr}}.C(s_1 s_2)
\end{aligned}
$$

Agent C is the communications layer. After a loss occurs, delivery of post-

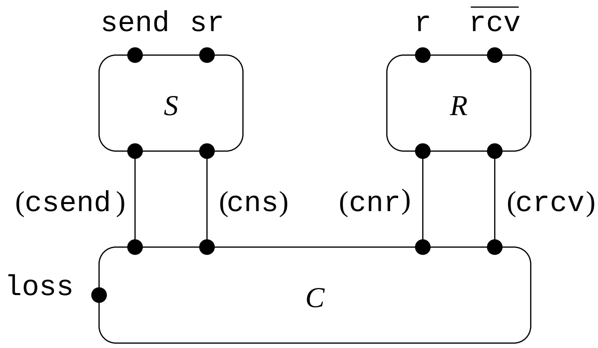

Figure 3: Flow Diagram of the Design

loss messages is delayed until the sender and receiver have been notified. After notification, C reverts to its normal behaviour.

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

Agent R is the communications process on the receiving machine. After notification of a loss (via action *cnr*), it accepts messages from the communication layer but discards them until after the special recovery message has been received.

The complete design is obtained by composing agents S, C, and R. To simplify notation, we write $P_1 \parallel P_2 \parallel P_3$ for $(P_1 \mid P_2 \mid P_3)\backslash\{\mathit{cnr}, \mathit{cns}, \mathit{csend}, \mathit{crcv}\}$. Figure 3 is a CCS flow diagram of the design.

$$
\mathit{Impl} \stackrel{\text{def}}{=} S \parallel C(<>) \parallel R
$$

The specification and design are observation equivalent if there is a bisimulation containing them. To try to find a bisimulation we can start by relating states that buffer the same messages. Taking into account buffered messages and the observable actions *loss*, *sr*, and *r*, a rough description of a candidate bisimulation between *Spec* and *Impl* is as follows:

- 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.

A problem with this proposal is that certain losses possible in the specification are not possible in the design. For example, the transition $E(< m >) \xrightarrow{loss} E_l(<>)$

of *Spec* cannot be matched by $S \parallel C(<>) \parallel R'(< m >)$. The specification allows all buffered messages to be lost, while the design only allows messages buffered by the communication layer to be lost — not messages buffered by the sending or receiving machines. This problem is like the one in the lossy buffer example of Section 2.

Another problem with the proposed bisimulation concerns buffered messages remaining after a *loss* action. Such messages can safely be delivered provided they were sent before the lost messages. In *Spec*, these pre-loss messages are delivered until the notify-recover action r occurs. In *Impl*, they are delivered until agent C notifies the receiver agent R that a loss has occurred (via the action *cnr*). Thus, *Impl* can reach a state $S_r \parallel C(s) \parallel R_d$ in which action r has not yet occurred but pre-loss messages cannot be delivered. This problem is like the one in the updating buffer example of Section 3.

A third problem is that *Impl* can perform a loss action while the recovery procedure is underway, while *Spec* cannot. In particular, a *loss* action can occur in *Impl* after synchronisation occurs on action *cmr* and before action r occurs. The possibility of losses during recovery is important and has been analysed as part of the system design, but is not treated in the specification here. We could modify the definition of agent *Impl* to make losses impossible during recovery through additional synchronisation between agents C and R. Since this would complicate the design unduly, we will simply ignore losses that occur during recovery in our analysis.

9 Showing Refinement

The CCS agents *Spec* and *Impl* are not observation equivalent. However, if *Spec* is taken as a MPL specification, and revised slightly, then it is related to *Impl* through the MPL refinement relation.

To revise *Spec*, we mark each action as either admissible or necessary. Recall that an implementation is acceptable if it never loses messages, and if it does not deliver pre-loss messages. Therefore, we mark all *loss* actions and all pre-loss $\overline{rcv}$ actions as admissible. The other actions are marked as necessary. The revised specification written in MPL is given below.

$$
\begin{array}{rcl}
\mathit{Spec}' & \stackrel{\mathrm{def}}{=} & E(<>) \\
E(s) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m)_{\Box}.E(s<m>) + \\
 & & (\text{if } \mathrm{len}(s) > 0 \text{ then } \overline{\mathit{rcv}}(\mathrm{hd}(s))_{\Box}.E(\mathrm{tl}(s))) + \\
 & & \sum_{s_1 \in \textbf{prefix}\, s} \mathit{loss}_{\Diamond}.E_l(s_1) \\
E_l(s) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m)_{\Box}.E_l(s) + \\
 & & (\text{if } \mathrm{len}(s) > 0 \text{ then } \overline{\mathit{rcv}}(\mathrm{hd}(s))_{\Diamond}.E_l(\mathrm{tl}(s)) + \\
 & & \mathit{sr}_{\Box}.E_r(s, <>) \\
E_r(s_1, s_2) & \stackrel{\mathrm{def}}{=} & \mathit{send}(m)_{\Box}.E_r(s_1, s_2<m>) + \\
 & & (\text{if } \mathrm{len}(s_1) > 0 \text{ then } \overline{\mathit{rcv}}(\mathrm{hd}(s_1))_{\Diamond}.E_r(\mathrm{tl}(s_1), s_2)) + \\
 & & r_{\Box}.E(s_2)
\end{array}
$$

Note that MPL is flexible enough to express that $\overline{\mathit{rcv}}$ actions are admissible specifically for the case of pre-loss messages. To revise the implementation, we simply take all actions to be necessary ones.

Next we prove that *Impl* is a refinement of *Spec*$'$. The automatic checking of refinement between finite-state MPL specifications is supported by tools [4], but *Impl* and *Spec*$'$ have infinitely-many states. We therefore do the refinement proof manually by finding and checking a weak refinement relation containing the specification and implementation.

Even for relatively small specifications like ours, finding and checking a refinement relation is tedious. Part of the problem is that in checking the relation we must check simple facts that are already known. To illustrate the problem we will look at the checking of a bisimulation relation. Consider a reliable version of the lossy buffer of Section 2.

$$
\begin{array}{rcl}
\mathit{Buf}_n(s) & \stackrel{\mathrm{def}}{=} & \text{if } \mathrm{len}(s) < n \text{ then } \mathit{in}(x).\mathit{Buf}_n(s<x>) + \\
 & & \text{if } \mathrm{len}(s) > 0 \text{ then } \overline{\mathit{out}}(\mathrm{hd}(s)).\mathit{Buf}_n(\mathrm{tl}(s))
\end{array}
$$

The following weak bisimulation relation proves that such a buffer of capacity two is observation equivalent to two linked cells.

$$
\begin{array}{rcl}
\{(\mathit{Buf}_2(<>) & , & \mathit{Buf} \frown \mathit{Buf}) \\
(\mathit{Buf}_2(<x>) & , & \mathit{Buf}'(x) \frown \mathit{Buf}), \\
(\mathit{Buf}_2(<x>) & , & \mathit{Buf} \frown \mathit{Buf}'(x)) \\
(\mathit{Buf}_2(<x,y>) & , & \mathit{Buf}'(y) \frown \mathit{Buf}'(x))\}
\end{array}
$$

The agents $\mathit{Buf}'(x) \frown \mathit{Buf}$ and $\mathit{Buf} \frown \mathit{Buf}'(x)$ are clearly observation equivalent from the definition of $\frown$ and the fact that $P \mid Q \approx Q \mid P$ for all agents P and Q. Yet both are shown above to be related to $\mathit{Buf}_2(<x>)$, and so in checking the relation we must essentially prove that $\mathit{Buf}'(x) \frown \mathit{Buf}$ and $\mathit{Buf} \frown \mathit{Buf}'(x)$ are observation equivalent.

The problem illustrated by this example becomes serious when larger agents are involved. To simplify bisimulation checking, a technique based on *expansion* relations [1] has been developed. We now describe this technique and show how it can be adapted to the checking of the MPL refinement relation.

A binary relation $\mathcal{R}$ on agents is an *expansion* if $(P, Q) \in \mathcal{R}$ implies:

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

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

Agent Q *expands* P, written $P \leq Q$, if there exists an expansion containing (P, Q).

The relation $\leq$ is a preorder but not an equivalence, and is strictly stronger than $\approx$. Intuitively, $P \leq Q$ holds if P and Q are observation equivalent, and P is at least as "fast" as Q in the sense that it performs no more τ actions than Q. The weak bisimulation relation shown above is also an expansion relation. In other words, the distributed buffer built from 2 cells expands the buffer of capacity 2.

Since the expands relation is stronger than observation equivalence, we can prove $P \approx Q$ by proving $P \leq Q$. The advantage in doing so is that we can avail ourselves of the *expansion up to* $\leq$ technique. A binary relation $\mathcal{R}$ on agents is an *expansion up to* $\leq$ if $(P, Q) \in \mathcal{R}$ implies:

1. Whenever $P \xrightarrow{\alpha} P'$, then $Q \overset{\alpha}{\Rightarrow} Q'$ for some Q' such that $(P', Q') \in \sim \circ \mathcal{R} \circ \leq$,

2. Whenever $Q \xrightarrow{\alpha} Q'$, then $P \xrightarrow{\widehat{\alpha}} P'$ for some P' such that $(P', Q') \in \leq \circ \mathcal{R} \circ \leq$.

If a pair (P, Q) of agents belongs to an expansion up to $\leq$, then Q expands P, and hence P and Q are observation equivalent.

Proposition 1 *If $\mathcal{R}$ is an expansion up to $\leq$ then $\mathcal{R} \subseteq \leq$.*

Proof. See [1].

Returning to the buffer example, the following is an expansion up to $\leq$:

$$\begin{array}{rcl} \{(Buf_2(<>) & , & Buf \frown Buf), \\ (Buf_2(<x>) & , & Buf \frown Buf'(x)), \\ (Buf_2(<x, y>) & , & Buf'(y) \frown Buf'(x))\} \end{array}$$

To prove that this relation is an expansion up to $\leq$ we first show that $Buf'(x) \frown Buf$ expands $Buf \frown Buf'(x)$ by any means we like. Then we use this fact in checking that the relation satisfies the definition of expansion up to $\leq$. Prop. 1 then tells us that $Buf \frown Buf$ expands $Buf_2(<>)$ and hence $Buf \frown Buf$ is observation equivalent to $Buf_2(<>)$.

The expansion up to $\leq$ technique has two advantages over the basic bisimulation proof technique. First, an expansion up to $\leq$ if often a smaller relation than the corresponding bisimulation, so less work is needed in verifying the relation. Secondly, the technique improves the modularity of a bisimulation proof: in one step the supporting facts are shown, in the next step the relation itself is checked.

We have adapted the expansion idea to refinement. A binary relation $\mathcal{R}$ on specifications is an *e-refinement* if $(S, T) \in \mathcal{R}$ implies:

1. Whenever $S \xrightarrow{\alpha}_{\diamond} S'$, then $T \xrightarrow{\widehat{\alpha}}_{\diamond} T'$ for some T' such that $(S', T') \in \mathcal{R}$,

2. Whenever $T \xrightarrow{\alpha}_{\Box} T'$, then $S \overset{\alpha}{\Rightarrow}_{\Box} S'$ for some S' such that $(S', T') \in \mathcal{R}$.

Agent S *e-refines* T, written $S \preceq T$, if (S, T) belongs to some e-refinement $\mathcal{R}$. Intuitively, $S \preceq T$ 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 $P \preceq Q$ coincides with $Q \leq P$ when P and Q are agents.

A binary relation $\mathcal{R}$ on specifications is an *e-refinement up to* $\preceq$ if $(S, T) \in \mathcal{R}$ implies:

1. Whenever $S \xrightarrow{\alpha}_{\diamond} S'$, then $T \xrightarrow{\widehat{\alpha}}_{\diamond} T'$ for some T' such that $(S', T') \in \preceq \circ \mathcal{R} \circ \preceq$,

2. Whenever $T \xrightarrow{\alpha}_{\Box} T'$, then $S \overset{\alpha}{\Rightarrow}_{\Box} S'$ for some S' such that $(S', T') \in \preceq \circ \mathcal{R} \circ \sim$.

Proposition 2 $\preceq \subset \unlhd$

Proof. The inclusion follows trivially from the definition of weak refinement, since $T \xrightarrow{\widehat{\alpha}}_{\diamond} T'$ implies $T \overset{\widehat{\alpha}}{\Rightarrow}_{\diamond} T'$, and $S \overset{\alpha}{\Rightarrow}_{\Box} S'$ implies $S \overset{\widehat{\alpha}}{\Rightarrow}_{\Box} S'$. For strictness we have that $P \not\preceq \tau.P$ and $P \unlhd \tau.P$. □

Proposition 3 *The MPL operators .,* $|$*,* $\backslash L$*, and* $[f]$ *all preserve* $\preceq$*.*

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

Proposition 4 *If* $\mathcal{R}$ *is an e-refinement up to* $\preceq$ *then* $\mathcal{R} \subseteq \preceq$*.*

Proof. The proof is similar to the proof of Proposition 4.6 in [10]. First one shows by diagram chasing that, if $\mathcal{R}$ is an e-refinement up to $\preceq$, then $\preceq \circ \mathcal{R} \circ \preceq$ is an e-refinement. Then, since $\preceq$ is the largest e-refinement, we have that $\preceq \circ \mathcal{R} \circ \preceq \subseteq \preceq$, and since the identity relation is contained in $\preceq$, we have that $\mathcal{R} \subseteq \preceq$. □

Now we shall prove that *Impl* is a refinement of *Spec'*. First we show that the refines relation holds between certain states of the implementation. Then we exhibit an e-refinement up to $\preceq$ containing the pair $(Impl, Spec')$.

Lemma 1 *For all sequences* s*,* s_1*,* s_2*, and all messages* m*:*

$$\begin{array}{rcl}
(S'(m) \mid C(s))\backslash L & \preceq & (S \mid C(s < m >))\backslash L \\
(S'(m) \mid C_l(s))\backslash L & \preceq & (S \mid C_l(s < m >))\backslash L \\
(S'(m) \mid C_{ls}(s_1, s_2))\backslash L & \preceq & (S \mid C_{ls}(s_1, s_2 < m >))\backslash L \\
(S'_r(m) \mid C(s))\backslash L & \preceq & (S_r \mid C(s < m >))\backslash L \\
(S'_r(m) \mid C_l(s_1, s_2))\backslash L & \preceq & (S_r \mid C_l(s_1, s_2 < m >))\backslash L \\
(S''_r \mid C_{ls}(s_1, s_2))\backslash L & \preceq & (S \mid C_{ls}(s_1, s_2 < \mathbf{R} >))\backslash L \\
(S''_r \mid C(s_1, s_2))\backslash L & \preceq & (S \mid C(s < \mathbf{R} >))\backslash L \\
(S'_r(m) \mid C_{ls}(s_1, s_2))\backslash L & \preceq & (S_r \mid C_{ls}(s_1, s_2 < m >))\backslash L
\end{array}$$

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

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

$$\begin{array}{rcl} ((S \mid C(s < m >)\backslash L & , & ((S'(m) \mid C(s))\backslash L) \\ ((S \mid C_l(s < m >)\backslash L & , & ((S'(m) \mid C_l(s))\backslash L) \\ ((S \mid C_{ls}(s_1, s_2 < m >)\backslash L & , & ((S'(m) \mid C_{ls}(s_1, s_2))\backslash L) \end{array}$$

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

Theorem 1 *Impl* $\trianglelefteq$ *Spec'*.

Proof. We exhibit a relation $\mathcal{R}$ that is an e-refinement up to $\preceq$. The pairs of $\mathcal{R}$ are as follows, for all sequences s, s_1, s_2, and s_3, and all messages m:

$$\begin{array}{rcll} (S \parallel C(s) \parallel R & , & E(s)) & (1) \\ (S \parallel C(s) \parallel R'(m) & , & E(< m > s)) & (2) \\ (S \parallel C_l(s_1, s_2) \parallel R & , & E_l(s_1)) & (3) \\ (S_r \parallel C(s) \parallel R_d & , & E_l(s)) & (4) \\ (S \parallel C_{ls}(s_1, s_2 < \mathbf{R} > s_3) \parallel R & , & E_r(s_1, s_3)) & (5) \\ (S \parallel C(s_1 < \mathbf{R} > s_2) \parallel R_d & , & E_r(s_1, s_2)) & (6) \\ (S \parallel C_l(s_1, s_2) \parallel R'(m) & , & E_l(s_1)) & (7) \\ (S \parallel C(s_1 < \mathbf{R} > s_2) \parallel R'_d(m) & , & E_r(< m > s_1, s_2)) & (8) \\ (S_r \parallel C_{ls}(s_1, s_2) \parallel R & , & E_l(s_1)) & (9) \\ (S_r \parallel C_{ls}(s_1, s_2) \parallel R'(m) & , & E_l(s_1)) & (10) \\ (S \parallel C_{ls}(s_1, s_2 < \mathbf{R} > s_3) \parallel R'(m) & , & E_r(< m > s_1, s_3)) & (11) \end{array}$$

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

In applying Lemma 1, we use preservation of $\preceq$ by the composition operator (Proposition 3). For example, from the first relation of the lemma we derive the relation $S'(m) \parallel C(s) \parallel R_d \preceq S \parallel C(s < m >) \parallel R$.

We shall not present every detail of the proof that $\mathcal{R}$ is an e-refinement up to $\preceq$. To show what is involved we present the checking of pair 4 of $\mathcal{R}$. First we check the admissible transitions that can be performed by $S_r \parallel C(s) \parallel R_d$. We indent matching transitions of $E_l(s')$, and write $\tau(a)$ for a τ action that results from synchronisation on action a.

$$S_r \parallel C(s) \parallel R_d \xrightarrow{send(m)}_\diamond S'_r(m) \parallel C(s) \parallel R_d \preceq S_r \parallel C(s < m >) \parallel R_d$$

$$\qquad E_l(s') \xrightarrow{send(m)}_\diamond E_l(s')$$

$$S_r \parallel C(s) \parallel R_d \xrightarrow{sr}_\diamond S''_r \parallel C(s < m >) \parallel R_d \preceq S \parallel C(s < \mathbf{R} >) \parallel R_d$$

$$\qquad E_l(s') \xrightarrow{sr}_\diamond E_r(s', <>)$$

$$S_r \parallel C(s) \parallel R_d \xrightarrow{\tau(crcv)}_\diamond S_r \parallel C(s) \parallel R_d$$

$$\qquad E_l(s') \xrightarrow{\varepsilon}_\diamond E_l(s')$$

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 $S_r \parallel C(s) \parallel R_d$ was not checked because only single losses are being considered (as described at the end of Section 8).

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

$$
\begin{array}{lll}
E_l(s') & \xrightarrow{send(m)}_{\Box} & E_l(s') \\
& S_r \parallel C(s) \parallel R_d \xrightarrow{send(m)}_{\Box} S'_r(m) \parallel C(s) \parallel R_d \xrightarrow{\tau(csend)}_{\Box} S_r \parallel C(s < m >) \parallel R_d & \\
E_l(s') & \xrightarrow{sr}_{\Box} & E_r(s', <>) \\
& S_r \parallel C(s) \parallel R_d \xrightarrow{sr}_{\Diamond} S''_r C R_d(s < m >) \xrightarrow{\tau(csend)}_{\Box} S \parallel C(s < \mathbf{R} >) \parallel R_d &
\end{array}
$$

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. $\Box$

10 Conclusions

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

The process of attempting the refinement proof exposed some minor errors in the definition of *Impl*; the version presented here is the corrected version. For example, agent R of the implementation was originally defined as follows:

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

The proof failed in checking a pair in the refinement relation containing $S \parallel C_{ls}(s_1, s_2 < \mathbf{R} > s_3) \parallel R'$ and $E_r(s_1, s_3))$. Here the specification could perform action r as a necessary action, but the design, with its component R in state R', needed to perform $\overline{rcv}$ before eventually performing r. To fix the problem, the definition of $R'(m)$ was changed to the following:

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

The change allows R' to synchronise on cnr, and thus to eventually perform 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.

The protocol described here was extended by the designers at Praxis to handle the case where losses can occur during recovery. A serious error was discovered in this extension during analysis. The discovery of this error before implementation of the system avoided expensive debugging and rework.

We performed the refinement proof by hand as there are no tools that support refinement checking for MPL specifications with infinitely-many states. Indeed, although there are tools that support algebraic reasoning on infinite-state CCS agents [9], we know of no tool that supports the checking of bisimulation relations on infinite-state CCS agents. This is surprising, since bisimulation is a powerful proof technique but one that is tedious to perform by hand.

A CCS

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 "invisible" action τ. We let α, β range over *Act*. The set of *labels* contains all actions except τ. We let l range over labels. Complementation extends to all labels: $\overline{\overline{l}} = l$.

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

$$P ::= A \;\Big|\; \alpha.P \;\Big|\; \sum_{i \in I} P_i \;\Big|\; P_1 \mid P_2 \;\Big|\; P \backslash L \;\Big|\; P[f]$$

We write $P_1 + P_2$ as an abbreviation for $\sum_{i \in \{1,2\}} P_i$, and 0 (the *inactive* agent) as an abbreviation for summation over the empty set. Constant definitions are of the form $A \stackrel{\text{def}}{=} P$. We refer to the set of all agents expressions and agent constants as simply *agents*, denoted by $\mathcal{P}$. We let P, Q range over $\mathcal{P}$.

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

$$. \quad : \quad \alpha.P \stackrel{\alpha}{\rightarrow} P$$

$$+ \quad : \quad \frac{P_j \stackrel{\alpha}{\rightarrow} P_j'}{\sum_{i \in I} P_i \stackrel{\alpha}{\rightarrow} P_j'} \; j \in I$$

$$\mid \quad : \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P \mid Q \stackrel{\alpha}{\rightarrow} P' \mid Q} \quad \frac{Q \stackrel{\alpha}{\rightarrow} Q'}{P \mid Q \stackrel{\alpha}{\rightarrow} P \mid Q'} \quad \frac{P \stackrel{l}{\rightarrow} P' \quad Q \stackrel{\overline{l}}{\rightarrow} Q'}{P \mid Q \stackrel{\tau}{\rightarrow} P' \mid Q'}$$

$$\backslash \quad : \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P \backslash L \stackrel{\alpha}{\rightarrow} P' \backslash L} \; \alpha \notin L \cup \overline{L}$$

$$f \quad : \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P[f] \stackrel{f(\alpha)}{\longrightarrow} P'[f]}$$

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

We next define a behavioural equivalence on agents. A binary relation $\mathcal{R}$ on agents is a *bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

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

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

Agents P and Q are *strongly equivalent*, written $P \sim Q$, if (P, Q) belongs to some bisimulation $\mathcal{R}$.

Strong equivalence is too strong in the sense that it does not treat τ actions as invisible to an observer. A weaker equivalence is based on a transition relation on agents that represents observable transitions. We define $P \stackrel{\varepsilon}{\Rightarrow} P'$ as $P(\xrightarrow{\tau})^* P'$ and $P \stackrel{l}{\Rightarrow} P'$ as $P \stackrel{\varepsilon}{\Rightarrow} \circ \xrightarrow{l} \circ \stackrel{\varepsilon}{\Rightarrow} P'$ (where $\circ$ denotes the composition of relations). We also define $\widehat{\alpha}$ as ε if $\alpha = \tau$ and α otherwise.

An observation equivalence can now be defined as the bisimulation equivalence generated by the relation $\Rightarrow$. A binary relation $\mathcal{R}$ on agents is a *weak bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

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

2. Whenever $Q \xrightarrow{\alpha} Q'$, then $P \stackrel{\widehat{\alpha}}{\Rightarrow} P'$ for some P' such that $(P', Q') \in \mathcal{R}$.

Agents P and Q are *observation equivalent*, written $P \approx Q$, if (P, Q) belongs to some bisimulation $\mathcal{R}$.

Value-passing CCS adds parameterised actions, parameterised agents, and conditional statements to CCS. The new notational features are defined by translation to basic CCS. Assume a fixed set V of data values, and a notation for data expressions. Input action $a(x).P$ is defined to be $\sum_{v \in V} a_v.P[v/x]$, where $P[v/x]$ is agent obtained by substituting v for all free occurrences of x within P. Output action $\overline{a}(e).P$ is defined to be $\overline{a}_v.P$, where v is the value obtained by evaluating data expression e. Parameterised agent definition $A(x_1, x_2, \ldots, x_n) \stackrel{\text{def}}{=} P$ is defined as the set of definitions $\{(A_{v_1, v_2, \ldots, v_n} \stackrel{\text{def}}{=} P[v_1/x_1, \ldots, v_n/x_n]) : (v_1, \ldots, v_n) \in V_{x_1} \times \cdots \times V_{x_n}\}$. Parameterised agent constant $A(e_1, \ldots, e_n)$ is defined to be $A_{v_1, \ldots, v_n}$, where v_i is the value obtained by evaluating data expression e_i. Finally, conditional statement *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.

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", and EPSRC project GR/J 58619 "Communication in Safety Cases – A Semantic Approach".

This paper is an expanded and revised version of a chapter of [2].

References

[1] 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 526. To appear in Acta Informatica.

[2] Glenn Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.

[3] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In *Proceedings of the 3rd Intl. Workshop on Computer Aided Verification, LNCS 575*, 1991.

[4] Jens C. Godskesen, Kim G. Larsen, and Michael Zeeberg. TAV (Tools for Automatic Verification) users manual. Technical Report R 89-19, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg, Denmark, August 1989.

[5] Anthony Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, 1996.

[6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[7] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, 1988.

[8] Kim Gulstrand Larsen. Modal specifications. In *Proceedings of the 1st Intl. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.

[9] Huimin Lin. PAM: A process algebra manipulator. In *Proceedings of the 3rd Intl. Workshop on Computer Aided Verification, LNCS 575*, pages 136–146, 1991.

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

[11] R. J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR 90*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[12] R. J. van Glabbeek. The linear time – branching time spectrum II (the semantics of sequential systems with silent moves). In E. Best, editor, *Proceedings CONCUR 93,* Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.

[13] D. J. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.