# A Functional Solution to the RPC-Memory Specification Problem[⋆]

Manfred Broy

Institut für Informatik
Technische Universität München
80290 München, Germany

**Abstract.** We give a functional specification of the syntactic interface and the black box behavior of an unreliable and a reliable memory component and a remote procedure (RPC) call component. The *RPC* component controls the access to the memory. In addition, we specify a clerk for driving the *RPC* component. The used method is modular and therefore it allows us to specify each of these components independently and separately. We discuss the specifications shortly and then compose them into a distributed system of interacting components. We prove that the specification of the composed system fulfills again the requirement specification of the unreliable memory component. Finally we give a timed version of the *RPC* component and of a clerk component and compose them.

## 1 Introduction

For describing the behavior of a reactive component we can either use state transition models or communication/action history based models. Using states, we specify the behavior of a system by a state machine that models its state changes. We specify liveness properties, that we cannot express easily by state transition techniques, separately, for instance by temporal logic.

Besides states we may use history based descriptions of the behavior of components. With them we model the behaviors of systems by their traces or by their input and output histories. In the following, we choose a history based modeling technique and describe the behavior of components by relations on their timed input and output streams. We provide modular specifications that model the behavior of the components independently.

We treat the specification problem of an unreliable memory, a *RPC* component and a clerk as posed in [Broy, Lamport]. In a first chapter, we briefly repeat the basic mathematical concepts of the used approach. Then we give interface

[⋆] This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the industrial research project SysLab.

specifications of the unreliable memory, the reliable memory, the RPC component, and the clerk. We put the latter three components together in parallel. Due to the modularity of our approach the specification of the composed system is obtained in a schematic modular way from the specifications of the components. For the composed system we prove that it is a refinement of the requirement specification for the unreliable memory again. We carry out this proof in the appendix.

We then go through the same exercise for a *RPC* component that reacts within time bounds. In particular, it determines by a time-out that certain *RPC* calls do not return. We specify, compose and verify the components as required in [Broy, Lamport]. So we solve all the tasks of the RPC-Memory Specification Problem. We do not give a proof of the correctness of the timed implementation.

## 2 The Basic System Model

As a basic model for describing the behavior of system components we use relations on timed streams. Timed streams are streams carrying data messages and time ticks. A timed stream is accordingly a finite or infinite sequence of messages and time ticks. A timed stream is called complete, if it contains an infinite number of time ticks. Apart from the time ticks a complete timed stream may carry a finite or an infinite number of messages. The basic idea is that the time ticks indicate the time bounds (the bounds of the time intervals) in which the messages are sent on a channel. On the basis of this simple model, we introduce a quite flexible notation in this section that we will use throughout the paper when writing specifications. For a detailed introduction into the theory of streams see appendix A.

We model the time flow by a special time signal called a time tick that indicates the end of a time interval. By the symbol

$$\surd$$

we denote the time tick signal. Let $M$ be a set of messages that does not contain the time signal $\surd$. By $M^{\omega}$ we denote the set of streams of messages from the set $M$ and by

$$M^{\underline{\omega}}$$

we denote the set of complete timed streams of elements from the set $M \cup \{\surd\}$ with an infinite number of ticks[2]. Every element in the set $M^{\underline{\omega}}$ denotes a complete timed communication history over an unbounded time period.

In the following, we use the notations and operators as given in Tab. 1 in the formulas specifying the components.

[2] Perhaps it is helpful to point out that of course the time ticks are not thought as signals that are actually transmitted. They are introduced rather as an auxiliary concept that allows us to model time.

For a stream $x \in (M \cup \{\surd\})^\omega$ we denote by:

$S\copyright x$ the substream of the elements from the set $S \subseteq M \cup \{\surd\}$ in stream $x$; if $S = \{a\}$ we write $a\copyright x$ instead of $S\copyright x$, $\copyright$ is called the *filter operator*,

$x.i$ the $i$-th element in the stream $x$ different from the signal $\surd$, more precisely the $i$-th element in the message stream $M\copyright x$,

$x : i$ the least prefix of the stream $M\copyright x$ that contains $i$ elements,

$x \downarrow i$ the largest prefix of the stream $x$ containing $i$ time ticks (the history till time point $i$),

$\#x$ the number of elements in stream $x$,

$S\ddagger x$ $\#S\copyright x$,

$x^{\mathsf{TM}} i$ the number of time ticks till the $i$-th message different from $\surd$ in the stream $x$, formally:

$$x^{\mathsf{TM}} i = \min\{\#\surd\copyright z : z \sqsubseteq x \land M\ddagger z \geq i\}$$

Here $\sqsubseteq$ denotes the prefix order which is formally introduced in appendix A. If $\#M\copyright x < i$ then $x^{\mathsf{TM}} i = \infty$.

**Table 1.** Table of the Used Notation

Fig. 1 shows a timed stream with its time ticks representing the beginning and the end of a time interval. In this example we have $x^{\mathsf{TM}} i = k$, since the $i$-th message (which is $x.i$) occurs after the $k$-th time tick.

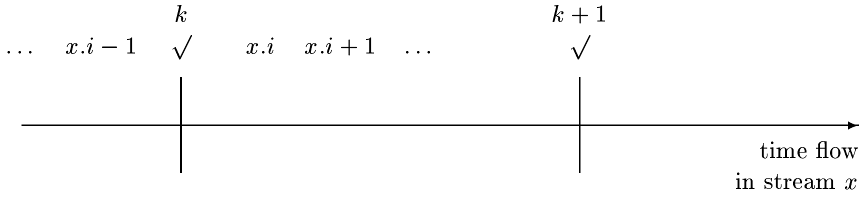

**Fig. 1.** Stream with Time Ticks

We describe the black box behavior of a component by a behavior relation. A behavior relation is a relation between the input streams and the output streams of a component that fulfills certain conditions with respect to their timing. Let $I_1, \dots, I_n, O_1, \dots, O_m$ be message sets where $m, n \in I\!N$. A graphical representation of a component $f$ with $n$ input channels of the sorts $I_1, \dots, I_n$ and $m$ output channels of the sorts $O_1, \dots, O_m$ is shown by Fig. 2.

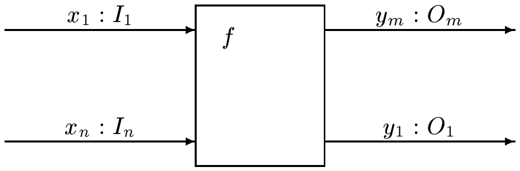

**Fig. 2.** Graphical representation of component $f$

A behavior relation for this component is represented by a predicate on the timed streams of input and output messages.

$$f : (I_1^{\underline{\omega}} \times \ldots \times I_n^{\underline{\omega}} \times O_1^{\underline{\omega}} \times \ldots \times O_n^{\underline{\omega}}) \to \mathbb{B}$$

For a behavior relation we always assume the following timing property (principle of time flow and delay):

$$x \downarrow i = z \downarrow i \Rightarrow \{y \downarrow i+1 : f(x,y)\} = \{y \downarrow i+1 : f(z,y)\}$$

The timing property expresses that the set of possible output histories for the first $i+1$ time intervals only depends on the input histories for the first $i$ time intervals. In other words, the processing of messages in a component takes at least one tick of time. We could work with more liberal conditions by dropping the $+1$ in the formula above. However, this timing condition is very convenient for us, since it leads to guarded recursion which is very useful in proofs.

## 3 Basic Message Sets Involved

In this section we shortly list the basic message sets and some additional auxiliary functions that we use in our specifications. We introduce the data sets given in Tab. 2.

| | | |
|---|---|---|
| $MemLocs$ | | memory locations, |
| $MemVals$ | | memory values, |
| $PrIds$ | | identifiers for processes, |
| $Procs$ | $= \{Read, Write\}$ | procedure names, |
| $Args$ | $= (MemLocs \times MemVals) \cup MemLocs$ | arguments, |
| $RetVals$ | $= MemVals \cup \{BadArg, MemFail, Ack\}$ | return values, |
| $RetMem$ | $= Calls \times RetVals$ | return messages of memory, |
| $Returns$ | $= Calls \times (RetVals$<br>$\cup \{RPCFailure, BadCall\})$ | return messages of $RPC$ comp., |
| $Calls$ | $= (PrIds \times Procs \times Args)$ | calls. |

**Table 2.** Table of data sets

By $MemLoc(c)$ we denote for every call $c \in Calls$ the memory location referenced in the arguments of the call $c$. By $MemVal(c)$ we denote the written value of a write call $c \in Calls$. We use the following subsets of the sets of calls and return values as abbreviations in specifications:

$$\begin{aligned} W(e) &= \{(p, Write, (e, v)) \in Calls : p \in PrIds \land v \in MemVals\} \\ R(c) &= \{(c', b) \in Returns : c = c'\} \end{aligned}$$

We assume that for every call $c \in Calls$ the identifier for the process that issued the call is denoted by $PrId(c)$. We define in addition the following sets in specifications:

$$\begin{aligned} C(p) &= \{c \in Calls : PrId(c) = p\} \\ RP(p) &= \{(c, b) \in Returns : PrId(c) = p\} \end{aligned}$$

For simplicity we assume that the set $RetVals$ does not contain the element $RPCFailure$ nor the element $BadCall$ and that the set $MemVals$ does not contain the element $BadArg$.

According to the informal specification some calls are bad. We assume a Boolean function

$$IsBadCall : Calls \to Bool$$

that allows us to distinguish bad calls from proper calls.

This set of definitions constitutes what software engineers call the *data model* of our little application.

## 4 The Unreliable and the Reliable Memory Component

The unreliable memory component is a simple device that receives a stream of memory calls and returns a stream of memory return messages. We model the memory component by a relation between its input and output streams, the timed stream of calls and the timed streams of memory return messages.

**Note:** On the Conceptional Model
In the informal specification as given in [Broy, Lamport] a scenario is used to explain the behavior of the memory component informally in terms of a conceptual model that refers to a particular implementation. It refers to a simple memory and to a clerk component that executes every call several times (or no call at all). Therefore a specification based on this scenario might be most suggestive when starting from the informal specification. We are rather interested in a black box specification that specifies the behavior of the unreliable memory by the relation between its input and output histories represented by streams. Therefore we do not give a specification in terms of an abstract implementation – as suggested by the informal description – but rather isolate its characteristic properties and formalize them.

The particular example of the *RPC* component includes a subtle difficulty for our specification technique. The fact that the memory may assume that for each processor at most one call is active makes it necessary to refer to the time order between output (returns to memory calls) and input (further calls). For (nontimed) stream processing functions it is easy to express that a certain input message is causal for a certain output message. This means that the output does never occur before the input occurred. However, it is difficult to express without time information that an input message occurs only after a particular output message[3]. We can express such a relationship in our model without problems due to the fact that we work with timed input and output streams. Therefore we can formulate the assumption that the next call of a process arrives only after the return to the previous one has been issued. □

The memory component MC with its syntactic interface is graphically shown in Fig. 3. Its behavior is specified formally along the lines of the informal statements (1) - (5) by the relation MC given below.

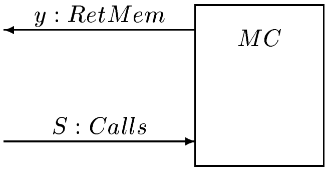

**Fig. 3.** The Memory Component MC as a Data Flow Node

For all component specifications we use the same format. We specify the syntactic interface that indicates the number of channels, their internal names and which messages are sent along the channels by a data flow node. Then we give an informal description of the properties and finally specify the relation MC by a logical formula describing a relation between input and output streams.

We formulate the specification in the so-called assumption/commitment format. In that format we write a specification by an implicative formula

$$A \Rightarrow C$$

where $A$ is the *assumption* and $C$ is the *commitment*. In the assumption we express the condition about the input streams that have to be fulfilled to be able to guarantee that the component works properly. The commitment $C$ formalizes what it means that the component works properly (for a detailed description of assumption/commitment specifications for stream processing components see [Stølen et al. 93] and [Broy 94b]).

For the unreliable memory the assumption, abbreviated by ProcAssumption, expresses the following property:

[3] This can easily be expressed by traces. A trace specification is given in the appendix.

(0) A call of a process to the memory only may occur when no other call of this process is active.

In other words, we require that at every point in time the number of calls of a process in the input stream is at most by one larger than the number of returns.

The commitment (the behavior guaranteed provided the assumption holds) of the unreliable memory component is described by the following basic statements:

There exists a history of calls that we may name the *internal access stream*[4] $z$ with the following properties:

(1) For every call in the input stream exactly one return message is issued.
(2) For every call in the input stream its corresponding return message fits into the set of allowed return messages for that call.
(3) Whenever there occurs a call in the internal access stream $z$ such a call is active at that point of time. A call is called *active* at a time point if it has been issued (received by the memory) but not answered yet.
(4) If a return message is an acknowledgment for a write call then there is an entry in the internal access stream at a moment where this call is active.
(5) If there is a successful return message for a read call c for location e which delivers the value v as the result there is an entry in the internal access stream at a time point where the call c is active such that the last value written for location e (or the special value InitVal, if such a write call does not exist) coincides with v.

The formal specification of the component MC follows exactly these informal statements. It reads as follows:

$$MC \equiv (s \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}):$$

$$ProcAssumption(s, y) \Rightarrow$$

$$\exists z \in Calls^{\underline{\omega}} : \forall v \in MemVals, c \in Calls, i \in [1 : c\ddagger s] :$$

$$(1) \quad c\ddagger s = R(c)\ddagger y$$

$$(2) \wedge Fit(d.i)$$

$$(3) \wedge \forall k \in [1 : c\ddagger z] : \exists j \in [1 : c\ddagger s] : active(j, (\{c, \surd\} \copyright z)^{TM} k)$$

$$(4) \wedge (d.i = (c, Ack) \Rightarrow \exists k : active(i, z^{\mathsf{TM}} k) \wedge z.k = c)$$

$$(5) \wedge (d.i = (c, v) \Rightarrow \quad \exists k : active(i, z^{\mathsf{TM}} k) \wedge v = last(z : k, MemLoc(c)))$$

$$\textbf{where } b = \{c, \surd\} \copyright s, d = (R(c) \cup \{\surd\}) \copyright y,$$

$$active(i, t) \equiv b^{TM} i \leq t \leq d^{\mathsf{TM}} i$$

[4] This internal access stream is introduced as an auxiliary construct. It reflects the informal problem description expressing that every call may perform a sequence of atomic accesses to the memory while active.

In this specification we use the following auxiliary functions and predicates. The term $last(y, e)$ denotes the last value written into the memory location $e$ in the finite stream $y$; it is $InitVal$ if such a call does not exist. Formally, this is specified by

$$last(y, c) = if \# x = 0 \text{ then } InitVal \text{ else } MemVal(x.\# x)$$

$$\text{where } x = W(MemLoc(c))©y$$

Here $W(e)$ denotes the set of all return messages to write calls to location $e$. We define the input assumption as follows:

$$ProcAssumption \equiv (s \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}) :$$

$$\forall p \in PrIds : \forall i \in I\!N : C(p)\ddagger(s \downarrow i+1) \leq 1 + RP(p)\ddagger(y \downarrow i)$$

We specify the predicate Fit that indicates whether a return message fits with a call as follows:

$$\begin{aligned}
Fit \equiv\ & ((j, p, a) \in (PrIds \times Procs \times Args), b \in RetVals) : \\
& (p = Write \Rightarrow \quad (b = Ack \wedge a \in MemLocs \times MemVals) \\
& \qquad\qquad\qquad \vee\ (b = BadArg \wedge a \notin MemLocs \times MemVals) \\
& \qquad\qquad\qquad \vee\ (b = MemFail)) \quad \wedge \\
& (p = Read \quad \Rightarrow \quad (b \in MemVals \wedge a \in MemLocs) \\
& \qquad\qquad\qquad \vee\ (b = BadArg \wedge a \notin MemLocs) \\
& \qquad\qquad\qquad \vee\ (b = MemFail))
\end{aligned}$$

This concludes the specification of the unreliable memory.

The assumption/commitment format inhibits a subtle point, which has to be clarified. Obviously the assumption does not only refer to the input stream $s$, but also to the output stream $y$. This may seem paradoxical, since then the assumption might be falsified by choosing a respective output stream. As explained in detail in [Broy 94a] the input assumption is – due to the timing property – in fact a restriction of the input stream $s$. Whenever the assumption $ProcAssumption(s, y)$ yields false, there exists a least time point $i > 0$ such that for all streams $s'$ and $y'$ with

$$s \downarrow i + 1 \sqsubseteq s' \quad and \quad y \downarrow i \sqsubseteq y'$$

we have $\neg ProcAssumption(s', y')$. Furthermore, we have[5]

$$ProcAssumption(s \downarrow i\hat{\ }\surd^{\infty}, y \downarrow (i-1)\hat{\ }\surd^{\infty})$$

[5] Here $\surd^{\infty}$ stands for an infinite stream of time signals $\surd$.

so $y \downarrow (i-1)\hat{}\sqrt{}^{\infty}$ fulfills the commitment. This imposes also, by the timing condition, that $y \downarrow i$ fulfills the safety properties of the commitment.

The specification MC is our solution to task 1(a) of [Broy, Lamport].

The reliable memory (task 1(a)) is easily specified similarly to the MC component as follows:

$RMC(s, y) \equiv$ like $MC(s, y)$, but in the definition of predicate Fit the branch
"$\ldots \vee b = MemFail \ldots$"
is omitted.

With this specification the answer to problem 1(b) is trivially "Yes" Since the reliable memory is specified just like the unreliable memory besides leaving out one disjunctive branch, we have immediately the theorem

$$RMC(s, y) \Rightarrow MC(s, y)$$

This is what we expect, since every behavior the reliable memory shows is also a behavior of the unreliable memory. So the reliable memory component $RMC$ is a property refinement of $MC$.

Also the answer to problem 1(c) is trivially "Yes". The memory component allows an implementation that always returns $MemFail$. We have not included any liveness assumption that expresses that a memory failure cannot be returned all the time.

## 5 The Remote Procedure Call Component

In this section, we specify the behavior of the remote procedure call component (Problem 2 in [Broy, Lamport]) that we call RPC. We specify it again by a relation between the input/output streams of the component. The remote procedure call component has two input channels, called x and y and two output channels called $s$ and $r$. The $RPC$ component is graphically shown as a data flow node in Fig. 4.

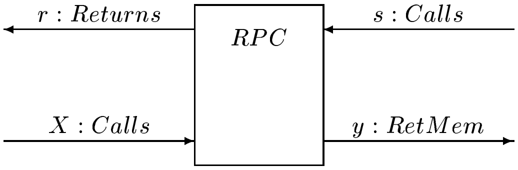

**Fig. 4.** RPC Component as a Data Flow Node

On its input channel $x$ the component $RPC$ receives calls and forwards them on its output channel $s$. On its input channel $y$ it receives memory return messages and forwards them on its output channel $r$ as return messages to its environment. We specify this relation again in the assumption/commitment format. The *RPC* assumption expresses that

(1) for each process at most one call is active at a time,
(2) on its return memory line $y$ at most as many return messages are received as resp. calls have been issued on the channel $s$ before and not answered so far,
(3) all calls issued on the channel $s$ eventually receive a memory return message on the channel $y$.

The specification of the *RPC* component is quite straightforward. It expresses the following four properties. For each call $c$ we require:

(0) If the call $c$ received on its input channel $x$ is a bad call then it is not forwarded on channel $s$ but returned with a *BadCall* return message on the return channel $r$.
(1) At most those calls are forwarded on channel $s$ that have been received on channel $x$.
(2) Only memory return messages are forwarded on channel $r$ that have been received on channel $y$.
(3) For each call on channel $x$ a return message is issued eventually on channel $r$.

Note that this way we model the calls returned with the *RPC* return message *RPCFailure* quite implicitly as a default by the fact that all calls arriving on channel $x$ eventually return.

The syntactic interface specification of the *RPC* component and its semantic interface defining its black box behavior are given by the following formula:

$$RPC \equiv (x \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}, s \in Calls^{\underline{\omega}}, r \in Returns^{\underline{\omega}}):$$

$$RPCAssumption(x, r, s, y) \Rightarrow \forall c \in Calls, v \in RetVals:$$

$$\begin{aligned}
&(0) \quad (IsBadCall(c) \Rightarrow c \ddagger x = (c, BadCall) \ddagger r \wedge c \ddagger s = 0)\\
&(1) \wedge c \ddagger s \le c \ddagger x\\
&(2) \wedge (c, v) \ddagger r \le (c, v) \ddagger y\\
&(3) \wedge R(c) \ddagger r = c \ddagger x
\end{aligned}$$

Here we define the assumption *RPCAssumption* as follows:

$$RPCAssumption \equiv (x \in Calls^{\underline{\omega}}, r \in Returns^{\underline{\omega}}, s \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}):$$

$$\forall c \in Calls, i \in I\!N:$$

$$\begin{aligned}
&(1) \quad ProcAssumption(x, r)\\
&(2) \wedge c \ddagger s \downarrow i \ge R(c) \ddagger y \downarrow i + 1
\end{aligned}$$

$(3) \wedge R(c)\ddagger y \geq c\ddagger s$

Again we express both safety and liveness properties by the specifying predicate for the *RPC* component.

## 6 Implementation with the Help of a Clerk

In this section we specify a clerk which can be used to drive an *RPC* component. It has a syntactic interface similar to the *RPC* component, but in contrast to this it forwards calls and receives returns from the *RPC* component and turns them into memory returns. The specification is quite similar. In [Broy, Lamport] the clerk is not mentioned and described explicitly but rather implicitly by describing the way the implementation works.

The clerk component *CLK* (Problem 3) is graphically shown in Fig. 5. It is specified again by an assumption/commitment specification.

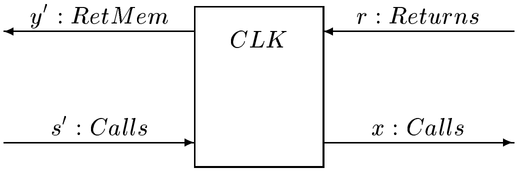

**Fig. 5.** The Clerk CLK as a Data Flow Node

We use the same assumption for the channels of the clerk as for the respective channels of the *RPC* component.

The commitment of the clerk component formalizes the following properties:

(1) The calls returned with *BadArg* messages on the channel $y'$ are those that are returned with *BadCall* and *BadArg* messages on the channel $r$.
(2) All memory failures that are received on the channel $r$ are forwarded on the channel $y'$.
(3) The calls on the channel $s'$, the *RPC* Failure return messages and Memory Failures on $r$ are exactly those calls forwarded on the channel $x$ or returned on the channel $y'$ with a memory failure message.
(4) All returns received on the channel $r$ which are not memory failure return messages are forwarded on the channel $y'$.
(5) All calls return.

These properties of the clerk are formally specified by a relation between its input and output streams as follows. By this specification we give both the syntactic interface and the history relation:

$$CLK \equiv (s' \in Calls^{\underline{\omega}}, r \in Returns^{\underline{\omega}}, x \in Calls^{\underline{\omega}}, y' \in RetMem^{\underline{\omega}}) :$$

$$RPCAssumption(s', y', x, r) \Rightarrow \forall c \in Calls, v \in RetVals\{MemFail\} :$$

$$\begin{array}{ll}
(1) & \quad (c, BadArg)\ddagger y' = (c, BadCall)\ddagger r + (c, BadArg)\ddagger r \\
(2) & \wedge\ (c, MemFail)\ddagger y' \geq (c, MemFail)\ddagger r \\
(3) & \wedge\ (c, MemFail)\ddagger y' + c\ddagger x = c\ddagger s' + (c, RPCFailure)\ddagger r + (c, MemFail)\ddagger r \\
(4) & \wedge\ (c, v)\ddagger y' = (c, v)\ddagger r \\
(5) & \wedge\ R(c)\ddagger y' = c\ddagger s'
\end{array}$$

Again this is a pure property specification just counting and relating the numbers of calls and return messages for them in the input and output channels.

## 7 The Composed System

The composition of the *RPC* component with the reliable memory component *RMC* and the clark CLK is straightforward along the lines of Fig. 6:

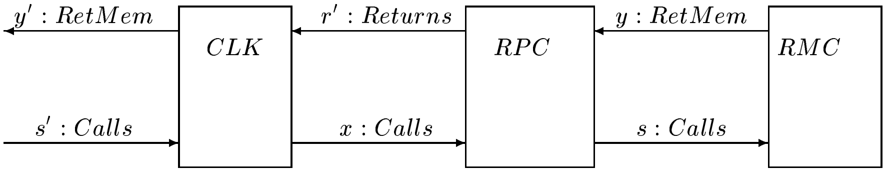

**Fig. 6.** Data Flow Diagram of the Composed System Called CI

The data flow diagram can be translated in a straightforward way into a logical formula specifying the component *CI*:

$$CI \equiv (s' \in Calls^{\underline{\omega}}, y' \in RetMem^{\underline{\omega}}) :$$

$$\exists x, s, r, y : CLK(s', r, x, y') \wedge RPC(x, y, s, r) \wedge RMC(s, y)$$

We claim that *CI* is an implementation (or in other words a refinement) for the unreliable memory. Formally this corresponds to the following verification condition:

$$CI(s', y') \Rightarrow MC(s', y')$$

A proof for the verification condition is given in the appendix. This is our solution to problem 3.

## 8 Specification of the Lossy RPC Component

The lossy *RPC* component can be specified along the lines of the *RPC* component. But now we work with a weaker input assumption. We do no longer assume that all calls that are issued on the output channel s actually return. Furthermore, we now refer to an explicit timing of the messages. Fig. 7 shows a graphical representation of the lossy *RPC* component. Its specification is given below.

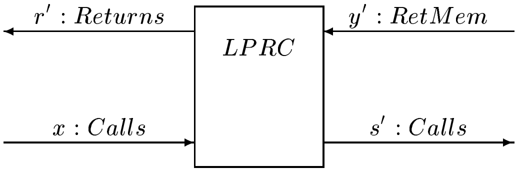

**Fig. 7.** Lossy *RPC* Component *LRPC* as a Data Flow Node

The specification of the component *LRPC* is quite straightforward. It expresses the following three properties. For each call c we require:

(1) At most those instances of the calls are forwarded on the channel $s$ that have been received on the channel $x$.
(2) Only answers are forwarded on the channel $r$ that have been received on the channel $y$ within $\delta$ units of time.
(3) For each call on channel $x$ a return message is issued on the channel $r$ within $\delta$ units of time after the return was received on the channel $y$ or after the call has been received on the channel $x$ (if the call has not been not forwarded on $s$ after $\delta$ units of time).

We specify the component *LRPC* by the following formula:

$$LRPC \equiv (x \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}, s \in Calls^{\underline{\omega}}, r \in Returns^{\underline{\omega}}):$$

$$\exists r' : r = NRPCF \copyright r' \wedge$$

$$((ProcAssumption(x, r') \wedge \forall c \in Calls, i \in I\!N : c \ddagger s \downarrow i \geq R(c) \ddagger y \downarrow i+1) \Rightarrow$$

$$\forall i \in I\!N, c \in Calls, v \in RetVals :$$

$$(1) \quad (IsBadCall(c) \Rightarrow R(c) \ddagger r' = (c, BadCall) \ddagger r' \wedge c \ddagger s = 0)$$

$$(2) \wedge (c, v) \ddagger r' \downarrow i + \delta \leq (c, v) \ddagger y \downarrow i$$

$$(3) \wedge R(c) \ddagger r' \downarrow i + \delta = (c \ddagger x \downarrow i) - (c \ddagger s \downarrow i + \delta) + R(c) \ddagger y \downarrow i$$

$$\textbf{where } NRPCF = (c, b) \in Returns : b \neq RPCFailure) \cup \{\surd\}$$

Hint: $(c\ddagger x \downarrow i) - (c\ddagger s \downarrow i + \delta)$ denotes the calls that are not forwarded. The RPCFailure signals are allowed inside the specification and then are filtered out. Note that this way we model the return message RPCFailure like in the case of the nonlossy $RPC$ component and filter them out later. This implies that for certain calls no return messages are issued.

The specification $LRPC$ is our solution to the problem 3' in [Broy, Lamport].

## 9 A Clerk for the Lossy RPC Component

In this section we specify a clerk component LCLK for the lossy $RPC$ component. It is illustrated in Fig. 8 and specified below.

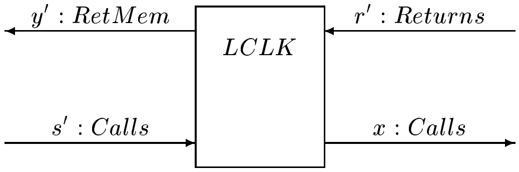

**Fig. 8.** Clerk of Lossy $RPC$ Component as a Data Flow Node

We use the same assumption for the channels of the clerk as for the respective channels of the $RPC$ component. The specification formalizes the following properties:

(1) All calls that are received on the channel $s'$ are forwarded on the channel $x$.
(2) The calls forwarded on the channel $x$ for which return messages are not returned within $2\delta+\epsilon$ time units after issued are returned with a $RPCFailure$, otherwise forwarded on the channel $y'$ as returned.

These properties are formally specified as follows:

$$LCLK \equiv (s' \in Calls^{\underline{\omega}}, r \in Returns^{\underline{\omega}}, x \in Calls^{\underline{\omega}}, y' \in RetMem^{\underline{\omega}}) :$$

$$(ProcAssumption(s', y') \wedge$$

$$\forall c \in Calls, i \in I\!N : c\ddagger x \downarrow i \geq R(c)\ddagger r \downarrow i + 1) \Rightarrow$$

$$(1)\ s' : \infty = x : \infty \wedge$$

$$(2)\ \forall c \in Calls, i \in [1 : c\ddagger x] :$$

$$(R(c) \copyright y').i = \textbf{if } i > R(c)\ddagger r \downarrow ((c, \surd \copyright x)^{\text{TM}} i + 2\delta + \epsilon) \textbf{ then } (c, RPCFailure) \textbf{ else } r.i \textbf{ fi}$$

Again this is a pure black box specification just specifying the properties of the external behavior by counting the numbers of calls and corresponding return messages within the required time bounds.

## 10 Composing the Clerk and the Lossy RPC Component

In this section we compose the clerk with the lossy *RPC* component. We obtain a composed system called CRPC as shown in Fig. 9. Its specification is a straightforward transliteration of the data flow graph into logic.

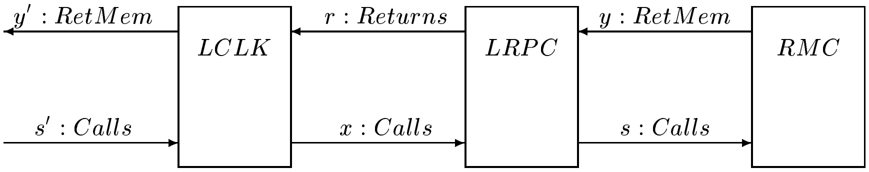

**Fig. 9.** Data Flow Diagram of the System CRPC Composed of the Lossy *RPC* Component and the Clerk

Again we obtain the component specification in a straightforward way from the data flow diagram:

$$\begin{aligned} CRPC \equiv\ & (s' \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}, s \in Calls^{\underline{\omega}}, y' \in RetMem^{\underline{\omega}}) : \\ & \quad \exists r, x : LCLK(s', r, x, y') \wedge LRPC(x, y, s, r) \end{aligned}$$

We obtain the following verification condition:

$$CRPC(s', y, s, y') \wedge (\forall c \in Calls, i \in \mathbb{N} : R(c) \ddagger y \downarrow i + \epsilon = c \ddagger s \downarrow i)$$
$$\Rightarrow RPC(s', y, s, y')$$

The verification condition includes the assumption that the calls issued on the channel $s$ are returned within $\epsilon$ time ticks.

## 11 Conclusion

We have given a complete treatment of the problems posed in [Broy, Lamport]. Although these problems were posed mainly in terms of an abstract design we have given requirement specifications and not design specifications. Of course, in cases where the informal problem description is given in terms of an abstract design a design specification using some abstract implementation model may be easier to relate to the informal description.

We demonstrate in appendix B how such a specification of the memory component using an abstract design can be given by functional modeling techniques. Another extreme of a specification is shown in appendix C. There we give a pure black box specification in terms of traces where no auxiliary construct is used such as a simple memory component or an internal access stream.

## Acknowledgment

It is a pleasure to thank Stephan Merz and Ketil Stølen for discussions and Birgit Schieder for comments. I am grateful to Peter Scholz and Ursula Hinkel for careful comments on a draft version.

## A Mathematical Basis

Throughout this paper interactive systems are supposed to communicate asynchronously through unbounded FIFO channels. Streams are used to denote histories of communications on channels. Given a set $M$ of messages, a *stream* over $M$ is a finite or infinite sequence of elements from $M$. By $M^*$ we denote the finite sequences over $M$. $M^*$ includes the empty stream that is denoted by $\langle\rangle$.

By $M^\infty$ we denote the infinite streams over the set $M$. $M^\infty$ can be represented by the total mappings from the natural numbers $\mathbb{N}$ into $M$. We denote the set of all streams over the set $M$ by $M^\omega$. Formally we have

$$M^\omega = M^* \cup M^\infty.$$

We introduce a number of functions on streams that are useful in system descriptions.

A classical operation on streams is the concatenation that we denote by ˆ. The concatenation is a function that takes two streams (say $s$ and $t$) and produces a stream $s\hat{}t$ as result, starting with the stream $s$ and continuing with the stream $t$. Formally the concatenation has the following functionality:

$$.\hat{}. : M^\omega \times M^\omega \to M^\omega.$$

If the stream $s$ is infinite, then concatenating the stream $s$ with a stream $t$ yields the stream $s$ again:

$$s \in M^\infty \Rightarrow s\hat{}t = s.$$

Concatenation is associative and has the empty stream $\langle\rangle$ as its neutral element:

$$r\hat{}(s\hat{}t) = (r\hat{}s)\hat{}t, \qquad \langle\rangle\hat{}s = s = s\hat{}\langle\rangle.$$

For any message $m \in M$ we denote by $\langle m\rangle$ the one element stream consisting of the element $m$.

On the set $M^\omega$ of streams we define a *prefix ordering* $\sqsubseteq$. We write $s \sqsubseteq t$ for streams $s$ and $t$ if $s$ is a prefix of $t$. Formally we have

$$s \sqsubseteq t \text{ iff } \exists r \in M^\omega : s\hat{\ }r = t.$$

The prefix ordering defines a partial ordering on the set $M^\omega$ of streams. If $s \sqsubseteq t$, then we also say that s is an *approximation* of $t$. The set of streams ordered by $\sqsubseteq$ is complete in the sense that every directed set $S \subseteq M^\omega$ of streams has a *least upper bound* denoted by $lub\ S$. A nonempty subset $S$ of a partially ordered set is called *directed*, if

$$\forall x, y \in S : \exists z \in S : x \sqsubseteq z \wedge y \sqsubseteq z.$$

By least upper bounds of directed sets of finite streams we may describe infinite streams. Infinite streams are also of interest as (and can also be described by) fixpoints of prefix monotonic functions. The streams associated with feedback loops in interactive systems correspond to such fixpoints.

A *stream processing function* is a function

$$f : M^\omega \to N^\omega$$

that is *prefix monotonic* and *continuous*. The function $f$ is called prefix monotonic, if for all streams $s$ and $t$ we have

$$s \sqsubseteq t \Rightarrow f.s \sqsubseteq f.t.$$

For better readability we often write for the function application $f.x$ instead of $f(x)$. A prefix monotonic function $f$ is called prefix continuous, if for all directed sets $S \subseteq M^\omega$ of streams we have

$$f.lub\ S = lub\ \{f.s : s \in S\}.$$

If a function is prefix continuous, then its results for infinite input can be already determined from its results on all finite approximations of the input.

By $\perp$ we denote the pseudo element that represents the result of diverging computations. We write $M^\perp$ for $M \cup \{\perp\}$. Here we assume that $\perp$ is not an element of $M$. On $M^\perp$ we define also a simple partial ordering called the flat ordering as follows:

$$x \sqsubseteq y \quad \text{iff} \quad x = y \vee x = \perp$$

We use the following functions on streams

$$ft : M^\omega \to M^\perp,$$

$$rt : M^\omega \to M^\omega.$$

The function $ft$ selects the first element of a nonempty stream. The function $rt$ deletes the first element of a nonempty stream.

For keeping our notation simple we extend the concatenation ˆ also to elements of the message set $M$ (treating them like one element sequences) and to tuples of streams (by concatenating the streams elementwise). For the special element $\perp$ we specify $\perp\hat{}s = \langle\rangle$. This equation reflects the fact that there cannot be any further message on a channel after trying to send a message that is to be generated by a diverging (and therefore never ending) computation.

The properties of the introduced functions can be expressed by the following equations (let $m \in M, s \in M^\omega$):

$$ft.\langle\rangle = \perp, \quad rt.\langle\rangle = \langle\rangle, \quad ft(m\hat{}s) = m, \quad rt(m\hat{}s) = s.$$

All the introduced concepts and functions such as the prefix ordering and the concatenation carry over to tuples of streams by pointwise application. Similarly the prefix ordering induces a partial ordering on functions with streams and tuples of streams as range.

We denote the function space of $(n, m)$-ary prefix continuous stream processing functions by

$$[(M^\omega)^n \sqsubseteq (M^\omega)^m]$$

The operations $ft$ and $rt$ are prefix monotonic and continuous, whereas concatenation ˆ as defined above is prefix monotonic and continuous only in its second argument.

# B State Transition Specification of the Memory Component

As pointed out in the introduction, an alternative to the functional specification for the description of the unreliable memory is a state transition specification. For the state transition specification we work with a state space. We show how to use state transition techniques to define history relations. A state of the unreliable memory is characterized by a value for each memory location and at most one active call for each process id. For each process at most one call may be active in the unreliable memory. If a call is active we store an optional memory return message from RetMem for that process; otherwise the default value nonactive is stored.

We use the set State to represent the state space of the unreliable memory. It is defined as follows:

$$State = (PrIds \to RetMem \cup \{nonactive\}) \times (MemLocs \to MemVals)$$

For a state $t \in State$ and a process $p \in PrIds$ we denote by $t.p$ the status (the optional return message or nonactive if the process is not active) of the process in the state $t$ and for a memory location $e \in MemLocs$ we denote by $t.e$ the value stored under that location. By $(t.p).2$ we denote the second component of the returned message, that is the returned value.

Each memory location holds a memory value. Initially the memory locations contain the value InitVal and all processes are not active.

We define a function that associates with every state of the memory component a behavior relation:

$$MCS : State \rightarrow [(Calls^{\underline{\omega}} \times RetMem^{\underline{\omega}}) \rightarrow Bool]$$

We specify the behavior relation $MCS$ for each state $t$ with the help of the state transition relations that associates input streams and output streams by the following formula:

$$\begin{array}{l}
MCS(t) \equiv (s \in Calls^{\underline{\omega}}, y \in RetMem^{\underline{\omega}}) : \\
\qquad \exists t' \in State : \\
\qquad (ft.y \neq \surd \wedge out(ft.y, t, t') \wedge MCS(t').(s, rt.y)) \vee \\
\qquad (ft.s \neq \surd \wedge ft.y = \surd \wedge ((in(ft.s, t, t') \wedge MCS(t').(rt.s, y)) \vee \\
\qquad \qquad \qquad \qquad t.PrIds(ft.s) \neq nonactive) \vee \\
\qquad (ft.s = \surd \wedge ft.y = \surd \wedge MCS(t).(rt.s, rt.y))
\end{array}$$

We require that MCS is the weakest relation that fulfills the equation above. We specify the state transition relations called in and out as follows:

$$\begin{array}{ll}
in((p, u, a), t, t') & \equiv SR(t[p := (c, MemFail)], t') \\
out(((p, u, a), z), t, t') & \equiv SR(t[p := nonactive], t') \\
 & \quad \wedge (u = read \Rightarrow z = (t.p).2) \\
 & \quad \wedge (u = write \Rightarrow z = (t.p).2)
\end{array}$$

$SR$ denotes a relation between the states of the memory component. It is specified below. Here we write

$$t[p := v]$$

to express that the state $t$ is updated selectively by changing the entry for process $p$ to $v$.

$$(t[p := v]).p = v$$

$$(t[p := v]).q = t.q \quad \Leftarrow \quad q \neq p$$

We always give an optional return value for each active process that is $MemFail$ in the beginning. In transitions this value may be changed.

In every state transition all active calls may change the memory locations. This is expressed by the relation $SR$ that specifies the relation between the old and the new state. The relation $SR$ is specified as follows:

$$
\begin{array}{l}
SR(t,t') \equiv \\
\forall p \in PrIds,\ e \in MemLocs, z \in RetVals, \\
\qquad\qquad u \in Procs, a \in Args : \exists z' \in RetVals : \\
\quad \begin{array}{lcl}
(t.p = nonactive & \Rightarrow & t'.p = nonactive) \quad \wedge \\
(t.p = ((p,u,a),z) & \Rightarrow & t'.p = ((p,u,a),z') \quad \wedge \\
(u = read & \Rightarrow & z' = t'.a \\
 & \vee & z' = MemFail \\
 & \vee & (z' = BadArg \wedge a \notin MemLocs)) \quad \wedge \\
(u = write & \Rightarrow & (z' = Ack \wedge CHD(MemLoc(a),t,t')) \\
 & \vee & z' = z \\
 & \vee & (z' = BadArg \wedge a \notin MemLocs \times MemVals)) \quad \wedge \\
(t.e = t'.e \vee CHD(e,t,t')) & & 
\end{array}
\end{array}
$$

The proposition $CHD(e,t,t')$ expresses that the memory location $e$ is accessed by one write action in the transition from state $t$ to state $t'$.

$$
\begin{array}{c}
CHD(e,t,t') \equiv \exists p \in PrIds, z \in \{Ack, MemFail\}, v \in MemVals : \\
t.p = ((p, write, (e,v)), z) \wedge t'.e = v
\end{array}
$$

Note that this specification does not include any liveness specifications. Therefore, by this specification, it is not guaranteed that every call eventually returns. However, it is not difficult to add this as an additional constraint. The same holds for the timing condition.

## C A Trace Solution to the Memory Specification Problem

A pure trace specification may be the most appropriate approach for a history-based specification of the unreliable memory. We give such a specification in the following. We use the following syntactic interface:

$$MC : Action^{\underline{\omega}} \to Bool$$

where the set action includes both the input and output actions of the unreliable memory:

$$Action = Calls \cup RetMem$$

Again we write an assumption/commitment specification. The assumption that for each process identifier at most one call is active can easily be formulated. We specify the commitment predicate of a trace by three conditions:

(1) Every call eventually returns.
(2) Every returned memory message fits to its call.
(3) A return message for a read call c returns a value written by one of the writers active while the call c is active or a value written by a write call that terminates before c is active provided there are no successful read or write calls that starts after the write call terminates and terminates before the read call starts.

Conditions (1) and (2) are obvious. Condition (3) is more sophisticated and requires a more detailed justification. According to the behavior of the memory a write call may be executed many times while it is active (while it is issued but has not returned yet). A read call $c$ may access the memory any time while it was active. Therefore it may return any value written by a write call to the respective location that overlaps in its activity interval with that of the read call $c$. Besides that, it may read a value that was written before the read call $c$ is issued if this value is stored in the memory at the time the read call $c$ is issued.

A value written by a write call $w$ (or the initial value, if there are no write calls) can only be the value stored in the memory at the time the read call $c$ is issued if there does not exist an output that indicates that the value was definitely overwritten. This is indicated by another read call $c'$ that starts after the write call $w$ for the respective location has been completed and returns a different value and ends before the read call $c$ starts or by a write call $w'$ that starts after the write call $w$ was finished and ends successfully. In both cases the value written by the write call $w$ has been definitely overwritten. If there exists such a write call $w$ and if read calls $c'$ and write calls $w'$ with the specified properties do not exist then there is always a possibility that the written value is still stored and can be read.

This analysis shows that a value returned by a read call either is a value written by a write call with an overlap in the activity interval or it is the last written value.

The trace predicate is specified in the assumption/commitment format again by the following formula:

$MC \equiv (t \in Action^{\underline{\omega}}):$

$(\forall s : s \sqsubseteq t \Rightarrow (\forall p \in PrIds : \#C(p) \copyright s \leq 1 + \#RP(p) \copyright s)) \Rightarrow$

$\forall c \in Calls:$

(1) $\#c \copyright t = \#R(c) \copyright t \wedge \forall i \in [1 : \#t]:$

(2) $t.i \in RetMem \Rightarrow Fit(t.i) \wedge$

(3) $t.i \in Calls \times MemVals \Rightarrow val(t.i) \in active(t, k, i, e) \cup posval(t, k)$

**where**

$e = Loc(t.i)$

$j = \mathit{Call\_to\_Return}(t, i)$

$$\begin{aligned}
&k = \max\{\mathit{Call\_to\_Return}(t,a) : a < j \ \wedge \\
&\qquad\qquad\qquad\qquad t.a \text{ is a successful write or read for location } e\} \\
&\mathit{Call\_to\_Return}(t,i) = \text{ index of the call to the return action } t.i \text{ in trace } t \\
&\mathit{Loc}(a) = \text{ Location referenced in action } a \\
&\mathit{active}(t,k,i,e) = \{\mathit{val}(t.w) : \mathit{Loc}(t.w) = e \wedge \mathit{Proc}(t.w) = \mathit{Write} \wedge k \leq w \leq i\} \\
&\mathit{posval}(t,k) = \{\mathit{val}(t.w) : \mathit{Loc}(t.w) = e \wedge \mathit{Proc}(t.w) = \mathit{Write} \wedge \mathit{free}(w,j)\} \\
&\qquad\qquad\quad \cup\{\mathit{InitVal} : \mathit{free}(0,j)\} \\
&\mathit{free}(w,j) \ \ = \forall z : t.z \in \mathit{RetMem} \wedge \\
&\qquad\qquad\quad w < \mathit{Call\_to\_Return}(t,z) \wedge \\
&\qquad\qquad\quad z < j \wedge \\
&\qquad\qquad\quad \mathit{Loc}(t.z) = e \Rightarrow \\
&\qquad\qquad\quad (t.z \notin \mathit{Calls} \times \{\mathit{Ack}\} \wedge t.z \notin \mathit{Calls} \times \mathit{MemVals} \backslash \{\mathit{val}(t.i)\}
\end{aligned}$$

According to our assumption we can find the unique index j of the call action for which a return message is issued.

The procedure Fit is specified as in section 4.

By this trace specification we obtain a purely extensional specification. Without referring to a simple memory component or to an internal access stream we specify the properties of a trace of the unreliable memory. The same style of specification can be used for the relational component model since all the used concepts can also be expressed, there.

Of course this specification could only be written after a careful analysis of the informal description, understanding all the data dependencies. However, such an analysis is useful and necessary, anyhow in a well-organized development method. The trace specification is very interesting if we only intend to write specifications. It is not so easy to deal with if we want to compose specifications.

## D Proof of the Verification Conditions.

In this appendix we give the proof for the first of our two basic correctness theorems. It claims that the composed system CI is a refinement of the memory component $MC$.

**Theorem:**
The system $CI$ is a refinement of the unreliable memory component $MC$:

$$CI(s', y') \Rightarrow MC(s', y')$$

**Proof:** We may use $CI(s', y')$ as our logical assumptions to prove $MC(s', y')$. Unfolding $CI(s', y')$ yields:

$$\exists x, s, r, y : CLK(s', r, x, y') \land RPC(x, y, s, r) \land RMC(s, y)$$

Furthermore, we may add the assumption in the specification of $MC(s', y')$

$$ProcAssumption(s', y')$$

to our assumptions. Unfolding yields (we drop all outermost universal quantifiers):

(A') $\quad C(p)\ddagger(s' \downarrow i+1) \leq 1 + RP(p)\ddagger(y' \downarrow i)$

We have to prove that there exists an internal access stream $z \in Calls^{\underline{\omega}}$ such that the following properties hold:

(U1) $\quad c\ddagger s' = R(c)\ddagger y' \land$

(U2) $\quad Fit(d.i) \land$

(U3) $\quad \forall k \in [1 : c\ddagger z] : \exists j \in [1 : c\ddagger s'] : active(j, (c, \surd \copyright z)^{TM} k) \land$

(U4) $\quad (d.i = (c, Ack) \Rightarrow \exists k : active(i, z^{TM} k) \land z.k = c) \land$

(U5) $\quad (d.i = (c, v) \Rightarrow \exists k : active(i, z^{TM} k) \land v = last(z : k, MemLoc(c)))$

**where** $\quad b = \{c, \surd\} \copyright s', d = (R(c) \cup \{\surd\}) \copyright y',$

$\quad active(i, t) \equiv b^{TM} i \leq t \leq d^{TM} i$

We unfold the specification $CLK$, $RPC$ and $RMC$ that are used in the description of $CI$. For the clerk specification $CLK(s', r, x, y')$ we obtain the following properties:

$RPCAssumption(s', y', x, r) \Rightarrow \forall c \in Calls, v \in MemVals :$

(C1) $\quad (c, BadArg)\ddagger y' = (c, BadCall)\ddagger r + (c, BadArg)\ddagger r \land$

(C2) $\quad (c, MemFail)\ddagger y' \geq (c, MemFail)\ddagger r \land$

(C3) $\quad (c, MemFail)\ddagger y' + c\ddagger x = c\ddagger s' + (c, RPCFailure)\ddagger r + (c, MemFail)\ddagger r \land$

(C4) $\quad (c, v)\ddagger y' = (c, v)\ddagger r \land$

(C5) $\quad R(c)\ddagger y' = c\ddagger s'$

For $RPC(x, y, s, r)$ we obtain the following properties:

$RPCAssumption(x, r, s, y) \Rightarrow \forall c \in Calls, v \in RetVals :$

(R0) $\quad (IsBadCall(c) \Rightarrow c\ddagger x = (c, BadCall)\ddagger r \land c\ddagger s = 0) \land$

(R1) $\quad c\ddagger s \leq c\ddagger x \land$

(R2) $(c,v)\ddagger r \le (c,v)\ddagger y \wedge$

(R3) $R(c)\ddagger r = c\ddagger x$

For $RMC(s,y)$ we get that there exists an internal access stream $z' \in Calls^{\underline{\omega}}$ such that the following properties hold:

$ProcAssumption(s,y) \Rightarrow \forall v \in MemVals, c \in Calls, i \in [1 : c\ddagger s]$ :

(M1) $c\ddagger s = R(c)\ddagger y$

(M2) $Fit(d.i)$

(M3) $\forall k \in [1 : c\ddagger z'] : \exists j \in [1 : c\ddagger s] : active(j, (\{c, \surd\}\copyright z')^{TM} k)$

(M4) $(d.i = (c, Ack) \Rightarrow \exists k : active(i, {z'}^{TM} k) \wedge z'.k = c)$

(M5) $(d.i = (c,v) \Rightarrow \exists k : active(i, {z'}^{TM} k) \wedge v = last(z' : k, MemLoc(c)))$

**where** $b = \{c, \surd\}\copyright s, d = (R(c) \cup \{\surd\})\copyright y,$

$active(i,t) \equiv b^{TM} i \le t \le d^{TM} i$

To be able to make use of the input assumptions of the components above we first show that all the component assumptions hold provided the input assumption for $MC(s', y')$ holds. These assumptions are:

$$\begin{array}{l} ProcAssumption(s,y) \\ RPCAssumption(s', y', x, r) \\ RPCAssumption(x, r, s, y) \end{array}$$

Unfolding yields the assumption for the three components (which is abbreviated by $As(y', x, s, y, r, s')$):

(A) $C(p)\ddagger(s \downarrow i+1) \le 1 + RP(p)\ddagger(y \downarrow i) \wedge$

(C'1) $C(p)\ddagger(s' \downarrow i+1) \le 1 + RP(p)\ddagger(y' \downarrow i) \wedge$

(C'2) $c\ddagger x \downarrow i \ge R(c)\ddagger r \downarrow i+1 \wedge$

(C'3) $R(c)\ddagger r \ge c\ddagger x \wedge$

(R'1) $C(p)\ddagger(x \downarrow i+1) \le 1 + RP(p)\ddagger(r \downarrow i) \wedge$

(R'2) $c\ddagger s \downarrow i \ge R(c)\ddagger y \downarrow i+1 \wedge$

(R'3) $R(c)\ddagger y \ge c\ddagger s$

We get the following commitment for the three components (which is abbreviated by $Co(y', x, s, y, r, s')$):

(C1) - (C5), (R0) - (R3), (M1) - (M5)

We structure our proof as follows.

(1) We prove that every tuple of streams (y', x, s, y, r, s') which fulfills the formula

$$As(y', x, s, y, r, s') \Rightarrow Co(y', x, s, y, r, s')$$

fulfills the predicate $As(y', x, s, y, r, s')$ provided it fulfills the input assumption $ProcAssumption(s', y')$.

(2) We prove from $As(y', x, s, y, r, s') \wedge Co(y', x, s, y, r, s')$ the commitment of the unreliable memory.

The step (1) is structured into two steps.

(1a) We prove that if an assumption of a component contains a safety and a liveness part, by the time delay property we can assume that every family of streams that fulfills the safety part of the assumption till time point $i$ fulfills the safety part of the commitment till time point $i$.

(1b) We prove that every family of streams that fulfills the specification and the safety part of the assumption fulfills the liveness part of the assumption.

Proof of (1): The assumption contains only two liveness properties (C3) and (R3).
Proof of (1a): We prove that from

$$As(y', x, s, y, r, s') \Rightarrow Co(y', x, s, y, r, s')$$

we can deduce the safety assumption $As_S(y', x, s, y, r, s')$ given by the formulas:

(A) $C(p)\ddagger(s \downarrow i+1) \leq 1 + RP(p)\ddagger(y \downarrow i)$

(C'1) $C(p)\ddagger(s' \downarrow i+1) \leq 1 + RP(p)\ddagger(y' \downarrow i)$

(C'2) $c\ddagger x \downarrow i \geq R(c)\ddagger r \downarrow i+1$

(R'1) $C(p)\ddagger(x \downarrow i+1) \leq 1 + RP(p)\ddagger(r \downarrow i)$

(R'2) $c\ddagger s \downarrow i \geq R(c)\ddagger y \downarrow i+1$

for $i = n + 1$ from

$$\forall i \in I\!N : i \leq n \Rightarrow As_S(y', x, s, y, r, s') \wedge Co_S(y', x, s, y, r, s')$$

and from the safety part of the commitments.
We may assume the safety assumptions for $n = i$:

(A') $C(p)\ddagger(s' \downarrow n+1) \leq 1 + RP(p)\ddagger(y' \downarrow n)$

(A) $C(p)\ddagger(s \downarrow n+1) \leq 1 + RP(p)\ddagger(y \downarrow n)$

(C'1) $C(p)\ddagger(s' \downarrow n+1) \leq 1 + RP(p)\ddagger(y' \downarrow n)$

(C'2) $c\ddagger x \downarrow n \geq R(c)\ddagger r \downarrow n+1$

(R'1) $C(p)\ddagger(x \downarrow n+1) \leq 1 + RP(p)\ddagger(r \downarrow n)$

(R'2) $c\ddagger s \downarrow n \geq R(c)\ddagger y \downarrow n+1$

and the safety commitments that hold because of the delay property till time point $i = n+1$:

(C1) $(c, BadArg)\ddagger y' \downarrow n+2 \leq (c, BadCall)\ddagger r \downarrow n+1 + (c, BadArg)\ddagger r \downarrow n+1$

(C3) $(c, MemFail)\ddagger y' \downarrow n+2 + c\ddagger x \downarrow n+2 \leq$

$c\ddagger s' \downarrow n+1 + (c, RPCFailure)\ddagger r \downarrow n+1 + (c, MemFail)\ddagger r \downarrow n+1$

(C4) $(c, v)\ddagger y' \downarrow n+2 \leq (c, v)\ddagger r \downarrow n+1$

(C5) $R(c)\ddagger y' \downarrow n+2 \leq c\ddagger s' \downarrow n+1$

(R0) $(IsBadCall(c) \Rightarrow c\ddagger x \downarrow n+1 \geq (c, BadCall)\ddagger r \downarrow n+2 \wedge c\ddagger s = 0)$

(R1) $c\ddagger s \downarrow n+2 \leq c\ddagger x \downarrow n+1$

(R2) $(c, v)\ddagger r \downarrow n+2 \leq (c, v)\ddagger y \downarrow n+1$

(R3) $R(c)\ddagger r \downarrow n+2 \leq c\ddagger x \downarrow n+1$

(M1) $c\ddagger s \downarrow n+1 \geq R(c)\ddagger y \downarrow n+2$

(M2) $Fit(d.n+1)$

(M3) $\forall k \in [1 : c\ddagger z] : \exists j \in [1 : c\ddagger s] : active(j, (c, \surd \copyright z')^{TM} k)$

(M4) $(d.n+1 = (c, Ack) \Rightarrow \exists k : active(n+1, z'^{TM} k) \wedge z'.k = c)$

(M5) $(d.n+1 = (c, v) \Rightarrow \exists k : active(n+1, z'^{TM} k) \wedge (c, v) = last(z' : k, c))$

**where** $b' = c, \surd \copyright s, d' = (R(c) \cup \{\surd\}) \copyright y,$

$active(n+1, t) \equiv b'^{TM} n+1 \leq t \leq d'^{TM} n+1$

We have to prove:

(A) $C(p)\ddagger(s \downarrow n+2) \leq 1 + RP(p)\ddagger(y \downarrow n+1)$

(C'1) $C(p)\ddagger(s' \downarrow n+2) \leq 1 + RP(p)\ddagger(y' \downarrow n+1)$

(C'2) $c\ddagger x \downarrow n+1 \geq R(c)\ddagger r \downarrow n+2$

(R'1) $C(p)\ddagger(x \downarrow n+2) \leq 1 + RP(p)\ddagger(r \downarrow n+1)$

(R'2) $c\ddagger s\downarrow n+1 \geq R(c)\ddagger y\downarrow n+2$

We give only an informal proof outline and do not carry out all the formal steps of the proof. The assumption (C'2) for the clerk follows immediately from (R3). The assumption (C'1) for the clerk follows immediately from the assumption (A'). The assumption (R'1) follows from the commitments by straightforward arithmetic manipulation.
Proof of (1b): We show that from

$$As(y',x,s,y,r,s') \Rightarrow Co(y',x,s,y,r,s')$$

and the safety assumption $As_S(y',x,s,y,r,s')$ we can deduce the liveness part of the assumption

(C'3) $R(c)\ddagger r \geq c\ddagger x$

(R'3) $R(c)\ddagger y \geq c\ddagger s$

Proof of (2): We assume the properties $As(y',x,s,y,r,s') \wedge Co(y',x,s,y,r,s')$ and prove the commitment of the unreliable buffer. We can assume therefore the following properties:

(A) $C(p)\ddagger(s\downarrow i+1) \leq 1+RP(p)\ddagger(y\downarrow i) \quad \wedge$

(C'1) $C(p)\ddagger(s'\downarrow i+1) \leq 1+RP(p)\ddagger(y'\downarrow i) \quad \wedge$

(C'2) $c\ddagger x\downarrow i \geq R(c)\ddagger r\downarrow i+1 \quad \wedge$

(C'3) $R(c)\ddagger r \geq c\ddagger x \quad \wedge$

(R'1) $C(p)\ddagger(x\downarrow i+1) \leq 1+RP(p)\ddagger(r\downarrow i) \quad \wedge$

(R'2) $c\ddagger s\downarrow i \geq R(c)\ddagger y\downarrow i+1 \quad \wedge$

(R'3) $R(c)\ddagger y \geq c\ddagger s \quad \wedge$

(C1) $(c,BadArg)\ddagger y' = (c,BadCall)\ddagger r + (c,BadArg)\ddagger r \quad \wedge$

(C2) $(c,MemFail)\ddagger y' \geq (c,MemFail)\ddagger r \quad \wedge$

(C3) $(c,MemFail)\ddagger y' + c\ddagger x = c\ddagger s' + (c,RPCFailure)\ddagger r + (c,MemFail)\ddagger r \quad \wedge$

(C4) $(c,v)\ddagger y' = (c,v)\ddagger r \quad \wedge$

(C5) $R(c)\ddagger y' = c\ddagger s' \quad \wedge$

(R0) $(IsBadCall(c) \Rightarrow c\ddagger x = (c,BadCall)\ddagger r \wedge c\ddagger s = 0) \quad \wedge$

(R1) $c\ddagger s \leq c\ddagger x \quad \wedge$

(R2) $(c,v)\ddagger r \leq (c,v)\ddagger y \quad \wedge$

(R3) $R(c)\ddagger r = c\ddagger x \quad \wedge$

(M1) $c\ddagger s = R(c)\ddagger y \quad \wedge$

(M2) $Fit(d'.i) \quad \wedge$

(M3) $\forall k \in [1 : c\ddagger z] : \exists j \in [1 : c\ddagger s] : active(j, (c, \surd\copyright z')^{TM}k) \quad \wedge$

(M4) $(d.i = (c, Ack) \Rightarrow \exists k : active(i, {z'}^{TM}k) \wedge z'.k = c) \quad \wedge$

(M5) $(d.i = (c, v) \Rightarrow \exists k : active(i, {z'}^{TM}k) \wedge (c, v) = last(z' : k, c))$

where $b' = c, \surd\copyright s, d' = (R(c) \cup \{\surd\})\copyright y,$

$active(i, t) \equiv {b'}^{TM}i \leq t \leq {d'}^{TM}i$

Based on these assumptions we have to prove that there exists $z \in Calls^{\underline{\omega}}$ such that:

(U1) $c\ddagger s' = R(c)\ddagger y' \quad \wedge$

(U2) $Fit(d.i) \quad \wedge$

(U3) $\forall k \in [1 : c\ddagger z] : \exists j \in [1 : c\ddagger s'] : active(j, (c, \surd\copyright z)^{TM}k) \quad \wedge$

(U4) $(d.i = (c, Ack) \Rightarrow \exists k : active(i, z^{TM}k) \wedge z.k = c) \quad \wedge$

(U5) $(d.i = (c, v) \Rightarrow \exists k : active(i, z^{TM}k) \wedge (c, v) = last(z : k, c))$

**where** $b = c, \surd\copyright s', d = (R(c) \cup \{\surd\})\copyright y',$

$active(i, t) \equiv b^{TM}i \leq t \leq d^{TM}i$

The statements (U1) and (U2) are logical consequences of our assumptions. (U1) is exactly (C5). (U2) follows from (M2) by (R2) and (R0) and by (C4) and (C2). To prove the properties (U3)-(U5) we have to construct a stream $z$ that has the required properties. We do this based on the stream $z'$.

Every entry in $z$ corresponds to an entry in $z'$. We define this correspondence by

$$z.i = \max\{j \in I\!N : \exists k \in I\!N : (z' \downarrow k).j = z.i \;\wedge\; R(c) \neq (y' \downarrow k) = i \\ \wedge\; c \neq (s' \downarrow k) = i\}$$

By this definition $z.i$ is the last reply to the iterated sending of the call $z'.i$ to the memory component. With the help of this definition we can prove the properties (U3) - (U5) by straightforward arithmetic manipulation.

## References

[Broy 94a] M. Broy: Specification and Refinement of a Buffer of Length One. Marktoberdorf Summer School 1994

[Broy 94b] M. Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style. Technische Universität München, Institut für Informatik, TUM-I9417, June 1994

[Broy, Lamport] M. Broy, L. Lamport: The RPC-Memory Specification Problem. This volume.

[Broy, Stølen 94] M. Broy, K. Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach. In: Langmaack, H. and de Roever, W.-P. and Vytopil, J. (eds): Proc. FTRTFT'94, Lecture Notes in Computer Science 863, 1994, 247-267

[Stølen et al. 93] K. Stølen, F. Dederichs, R. Weber: Assumption/Commitment Rules for Networks of Agents. Technische Universität München, Institut für Informatik, TUM-I9302