Using Relations on Streams to Solve the RPC-Memory Specification Problem

Ketil Stølen*

Institut für Informatik, TU München, D-80290 München

Abstract. We employ a specification and refinement technique based on streams to solve the RPC-memory specification problem. Streams are used to represent the communication histories of channels. We distinguish between input and output streams. Each input stream represents the communication history of an input channel; each output stream represents the communication history of an output channel. Specifications are *relations* between input and output streams. These relations are expressed as logical formulas. Composition corresponds to logical *conjunction*. We distinguish between time-independent and time-dependent specifications. A time-independent specification is based on untimed streams; a time-dependent specification employs timed streams. Timed streams are needed to capture *timing* constraints and *causalities*. A specification *refines* (or alternatively, *implements*) another specification if any input/output behavior of the former is also an input/output behavior of the latter. This means that refinement corresponds to logical *implication*.

1 Introduction

We use streams and relations on streams to solve the the RPC-memory specification problem, as described in [BL]. We address all parts of the problem statement with one exception: we do not impose the hand-shake protocol on which [BL] is based. This means for example, a user may send a new call before the memory component issues a reply to the previous call sent by the same user. The hand-shake protocol is not considered because our approach is tailored towards asynchronous communication with *unbounded* buffering. More correctly, our approach is based on the hypotheses that asynchronous communication with unbounded buffering simplifies specifications and the verification of refinement steps — simplifies in the sense that it allows us to abstract from synchronization requirements needed in system descriptions based on bounded communication buffers. In our approach this kind of synchronization is first introduced when it is really needed, namely when a specification is mapped into its final implementation. How system specifications based on asynchronous communication with unbounded buffering can be refined into system specifications based on

* Address from September 1, 1996: Institute for Energy Technology, P.O.Box 173, 1751 Halden, Norway. Email:Ketil.Stoelen@hrp.no

hand-shake communication is explained in the appendix. More explicitly, in the appendix we show how the implementation of the memory component can be refined into an implementation which differs from the first in only one respect: the communication is conducted via hand-shakes. Thus, we do not ignore the hand-shake protocol because we cannot handle it. On the contrary, we can easily specify hand-shake communication. We ignore the hand-shake protocol because we see asynchronous communication with unbounded buffering as a helpful feature of our method.

Any sensible specification language allows specifications to be expressed in many different ways and styles, and this is of course also true for our approach. In this paper we have tried to find the right balance between readability and brevity. In particular, we characterize nondeterministic behavior in terms of *oracles*.

The rest of the paper is structured as follows. In Sect. 2 we introduce streams and the most basic operators for their manipulation. In Sect. 3 we specify the memory components. The RPC component is the subject of Sect. 4, and in Sect. 5 we show how it can be used to implement the memory component. Similarly, the lossy RPC component is specified in Sect. 6 and used to implement the RPC component in Sect. 7. In Sect. 8 we give a brief summary and draw some conclusions. Finally, as mentioned above, there is an appendix dealing with the introduction of hand-shake communication.

2 Streams and Operators on Streams

We use *streams* to model the communication histories of channels. A stream is a finite or infinite sequence of *messages*. The messages occur in the order they are transmitted. For any set of messages M, by M^∞, M^* and M^ω we denote respectively the set of all infinite streams over M, the set of all finite streams over M, and the set of all finite and infinite streams over M.

We now introduce the most basic operators for the manipulation of streams. $\mathbb{N}$ denotes the set of natural numbers; $\mathbb{N}_+$ denotes $\mathbb{N} \setminus \{0\}$; $\mathbb{N}_\infty$ denotes $\mathbb{N} \cup \{\infty\}$; $[1..n]$ denotes $\{1, .., n\}$; $\mathbb{B}$ denotes the Booleans. Let $r, s \in M^\omega$ and $j \in \mathbb{N}_\infty$:

- $\#r$ denotes the *length* of r. This means ∞, if r is infinite, and the number of messages in r otherwise.
- a^j denotes the stream consisting of exactly j *copies* of the message a.
- $\mathsf{dom}(r)$ denotes $\mathbb{N}_+$ if $\#r = \infty$, and $\{1, 2, \ldots, \#r\}$, otherwise.
- $r[j]$ denotes the jth message of r if $j \in \mathsf{dom}(r)$.
- $\langle a_1, a_2, .., a_n \rangle$ denotes the stream of length n whose first message is a_1, whose second message is a_2, and so on. As a consequence, $\langle\rangle$ denotes the *empty* stream.
- $r|_j$ denotes the *prefix* of r of length j if $0 \leq j < \#r$, and r otherwise. This means that $r|_\infty = r$.
- $r \frown s$ denotes the result of *concatenating* r and s. Thus, $\langle a, b \rangle \frown \langle c, d \rangle = \langle a, b, c, d \rangle$. If r is infinite we have that $r \frown s = r$.
- $r \sqsubseteq s$ holds if r is a *prefix* of or equal to s. In other words, if $\exists\, v \in M^\omega : r \frown v = s$.

For any n-tuple t, we use $\Pi_j.t$ to denote the jth component of t (counting from the left to the right). For example, $\Pi_3.(a, b, c) = c$. When convenient we use t_k as a short-hand for $\Pi_k.t$.

We also need a *filtration* operator Ⓢ for tuples of streams. For any n-tuple of streams t and natural number $j \in \mathbb{N}_+$ less than or equal to the shortest stream in t, by t/j we denote the n-tuple of messages whose kth component is equal to $(\Pi_k.t)[j]$. For example, if $t = (\langle a, b\rangle, \langle c, d\rangle)$ then $t/1 = (a, c)$ and $t/2 = (b, d)$. For any set of n-tuples of messages A and n-tuple of streams t, by $A Ⓢ t$ we denote the n-tuple of streams obtained from t by

- truncating each stream in t at the length of the shortest stream in t,
- selecting or deleting t/j depending on whether t/j is in A or not.

For example, if $n = 1$ we have that

$$\{a, b\} Ⓢ \langle a, b, d, c, d, a, d\rangle = \langle a, b, a\rangle$$

and if $n = 2$ we have that

$$\{(a, a), (b, b)\} Ⓢ (\langle a, b, a, b, a, b, a\rangle, \langle a, a, a, b\rangle) = (\langle a, a, b\rangle, \langle a, a, b\rangle)$$

Moreover, if $n = 3$ we have that

$$\{(a, b, c)\} Ⓢ (\langle a, a, a\rangle, \langle b, b, b\rangle, \langle a, b, c\rangle) = (\langle a\rangle, \langle b\rangle, \langle c\rangle)$$

3 Problem I: The Memory Component

In this section we specify the *reliable* and the *unreliable* memory components,[2] as described in Sect. 2 of the problem statement [BL]. To give a smooth introduction to our specification technique, we construct these specifications in a stepwise fashion. We first specify a simple memory component with a *sequential* interface. This component has exactly one input and one output channel. We refer to this memory component as the sequential memory component. Then we consider memory components with a *concurrent* interface. More explicitly, as indicated by Fig. 1, memory components which communicate with n user components in a *point-to-point* fashion. These components we call concurrent. We first explain how the specification of the sequential memory component can be generalized to handle a concurrent interface. This specification is then step-by-step generalized to capture the requirements to the reliable and unreliable memory components, as described in [BL].

[2] What we call the "unreliable memory component" corresponds to the "memory component" in [BL].

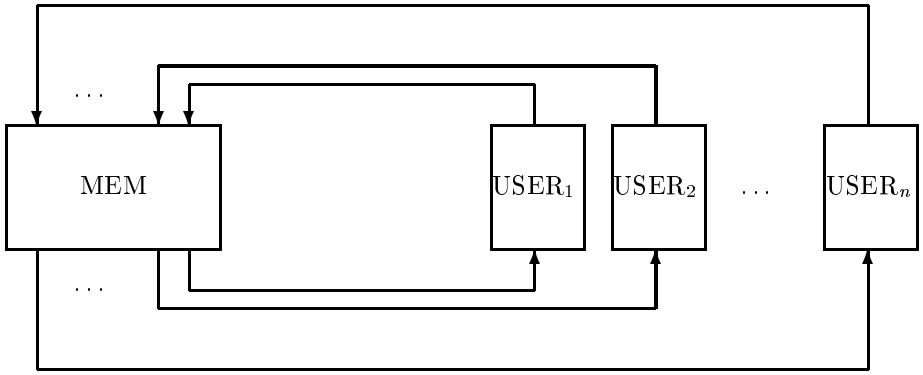


Fig. 1. Network with Concurrent Memory Component.

3.1 Basic Definitions

We first give some basic definitions.

$$
\begin{array}{ll}
\text{Call} & \stackrel{\text{def}}{=} \{\text{Read}(l) \mid l \in L\} \cup \{\text{Write}(l, v) \mid l \in L \wedge v \in V\} \\
\text{Rpl} & \stackrel{\text{def}}{=} \{\text{OkRd}(v) \mid v \in V\} \cup \{\text{OkWr}, \text{BadArg}, \text{MemFail}\} \\
\text{VldRd} & \stackrel{\text{def}}{=} \{\text{Read}(l) \mid l \in \text{MemLocs}\} \\
\text{Wr} & \stackrel{\text{def}}{=} \{\text{Write}(l, v) \mid l \in L \wedge v \in V\} \\
\text{VldWr}(l) & \stackrel{\text{def}}{=} \{\text{Write}(l, v) \mid v \in \text{MemVals}\} \\
\text{VldWr} & \stackrel{\text{def}}{=} \{\text{VldWr}(l) \mid l \in \text{MemLocs}\} \\
\text{VldCall} & \stackrel{\text{def}}{=} \text{VldRd} \cup \text{VldWr}
\end{array}
$$

L is a set of locations; V is a set of values. Call and Rpl are the sets of all possible calls and replies, respectively. VldRd and VldWr are the sets of all valid read and write calls, respectively. Wr is the set of all possible write calls, and VldWr(l) is the set of all write calls whose value is valid. VldCall is the set of all valid calls. We also introduce two auxiliary functions which are used to access the contents of calls.

$$\mathrm{Loc} \in \mathrm{Call} \to L$$

$$\mathrm{Loc}(\mathrm{Read}(l)) \stackrel{\mathsf{def}}{=} l$$

$$\mathrm{Loc}(\mathrm{Write}(l, v)) \stackrel{\mathsf{def}}{=} l$$

$$\mathrm{Val} \in \mathrm{VldWr} \to \mathrm{MemVals}$$

$$\mathrm{Val}(\mathrm{Write}(l, v)) \stackrel{\mathsf{def}}{=} v$$

By InitVal we denote the initial value of the memory locations. We assume that

$$\mathrm{InitVal} \in \mathrm{MemVals} \tag{1}$$

3.2 The Sequential Memory Component

As already mentioned, we start by specifying a simple sequential memory component. This component communicates with exactly one user component via exactly one input and one output channel. The component is reliable in the sense that each call results in exactly one memory access and that a call never fails. It can be specified as follows.

SeqMc — time_independent

in $i \in \mathrm{Call}^{\omega}$

out $o \in \mathrm{Rpl}^{\omega}$

$$\#o = \#i$$

$$\forall j \in \mathsf{dom}(o) :$$

$$\quad \mathsf{let}\ w = [\,\mathrm{VldWr}(\mathrm{Loc}(i[j]))\circledS(i|_j)\,]$$

$$\quad \mathsf{in}\ \mathrm{SeqMcBehavior}(i[j], o[j], w)$$

SeqMc is the *name* of the specification. The keyword time_independent is used to state that the specification is expressed in a *time-independent* setting. Thus, SeqMc does not impose any timing requirements. The keywords in and out separate the declarations of the *input streams* from the declarations of the *output streams*. In the specification above there is one input stream i of type Call^{ω} and one output stream o of type Rpl^{ω}. The input stream i models the communication history of the input channel. Similarly, the output stream o models the communication history of the output channel. We refer to these declarations as the specification's *syntactic interface*. The formula constituting the rest of the specification characterizes the required input/output behavior. We refer to this

formula as the I/O-relation. For any specification with name S, we use R_S to denote its I/O-relation.

Throughout this paper we often use line breaks to fix *scoping* and represent *conjunction*. We indicate scoping dependency by *indenting* the new line with respect to the previous one. For example, in the I/O-relation above, this technique is used to indicate that the let construct is in the scope of the universal quantifier. We indicate that a line break represents *conjunction* by not indenting the new line with respect to the previous one. For example, in the I/O-relation above the first line break represents a conjunction. The same does not hold for the third line break. The reason is of course that the third and fourth line do not represent formulas.

The I/O-relation of SeqMc has two main conjuncts. The first main conjunct requires that the number of output messages is equal to the number of input messages. This implies that exactly one reply is issued for each call. The second main conjunct determines the output messages as a function of the input messages. For any j, the let construct defines w to be the sub-stream of $i|_j$ of valid write calls to location $\mathrm{Loc}(i[j])$. The body of the let construct refers to an auxiliary function which is defined below.

SeqMcBehavior(i, o, w)

$$\mathrm{Call} \times \mathrm{Rpl} \times \mathrm{Wr}^* \to \mathbb{B}$$

$$i \in \mathrm{VldWr} \Rightarrow o = \mathrm{OkWr}$$

$$i \in \mathrm{VldRd} \Rightarrow$$

$$\quad w = \langle\rangle \Rightarrow o = \mathrm{OkRd}(\mathrm{InitVal})$$

$$\quad w \neq \langle\rangle \Rightarrow o = \mathrm{OkRd}(\mathrm{Val}(w[\#w]))$$

$$i \notin \mathrm{VldCall} \Rightarrow o = \mathrm{BadArg}$$

Note that the types of i, o differ from their types in SeqMc. The auxiliary function SeqMcBehavior has three main conjuncts. The first conjunct states that if the call is a valid write call, then the reply is an OkWr. The second conjunct states that if the call is a valid read call then the reply is OkRd(InitVal) if the actual memory location has not yet been updated (w is the empty stream); otherwise the reply is OkRd(v), where v is the value of the last update of this memory location (last message of w). The third conjunct states that if the call is not a valid call then BadArg is returned.

3.3 The Concurrent Memory Component

We now generalize SeqMc to handle a concurrent interface consisting of n input and n output channels. This new component, whose behavior is captured by the

specification ConMc below, communicates with n user components, as indicated by Fig. 1.

ConMc — time_independent

in $ip \in (\mathrm{Call}^\omega)^n$

out $op \in (\mathrm{Rpl}^\omega)^n$

$\exists\, p \in [1..n]^\infty, i \in \mathrm{Call}^\omega, o \in \mathrm{Rpl}^\omega$:

$\mathrm{Merged}(ip, i, p)$

$\mathrm{Merged}(op, o, p)$

$R_{\mathrm{SeqMc}(i,o)}$

ip and op are n-tuples of streams. The variable p is an *oracle* (or, alternatively, a *prophecy*) characterizing the order in which the input messages access the memory. Obviously, there are infinitely many interpretations of p. For each interpretation of p such that the first conjunct holds, the output history op is a function of the input history ip. Thus, the nondeterminism allowed by this specification is completely captured by p. Strictly speaking, also i and o are oracles. However, for any interpretation of p and input history ip their interpretations are fixed. The oracle i can be thought of as an internal buffer in which the input messages are placed in the same order as they access the memory — in other words, in accordance with p. Similarly, o can be seen as a buffer which stores the replies in the same order. This is expressed by the first two conjuncts. The third conjunct makes sure that i and o are related as in the sequential case. The auxiliary function Merged is defined below.

$\mathrm{Merged}(ap, a, p)$

$(\alpha^\omega)^n \times \alpha^\omega \times [1..n]^\infty \to \mathbb{B}$

$\forall\, k \in [1..n] : ap_k = \Pi_1.[\,(\alpha \times \{k\})\circledS(a, p)\,]$

α is a type variable. The filtration operator is used to make sure that the n streams in ap are merged into a in accordance with p.

3.4 The Repetitive Memory Component

The reliable memory component described in [BL] differs from ConMc in several respects. In particular, ConMc does not allow the same call to access the memory more than once. We now generalize ConMc to allow each call to result in an unbounded, finite, nonzero number of memory accesses. Note that we allow a read call to result in more than one memory access. Strictly speaking, this

is in conflict with the problem statement. However, we are only interested in specifying the externally observable behavior. In that case this deviation from the problem statement does not matter. Let

$$\mathrm{RpEx} \stackrel{\mathsf{def}}{=} \{\mathrm{Rep}, \mathrm{Exit}\}$$

The repetitive memory component can then be specified as below.

RepMc — time_independent

in	$ip \in (\mathrm{Call}^\omega)^n$
out	$op \in (\mathrm{Rpl}^\omega)^n$

$\exists\, p \in [1..n]^\infty, r \in \mathrm{RpEx}^\infty, i \in \mathrm{Call}^\omega, o \in \mathrm{Rpl}^\omega :$

$\mathrm{RepMerged}(ip, i, p, r)$

$\mathrm{RepMerged}(op, o, p, r)$

$\mathrm{Compatible}(i, p, r)$

$R_{\mathrm{SeqMc}(i,o)}$

Another oracle r has been introduced. It determines the number of memory accesses for each call. The final memory access for a call has Exit as its corresponding element in r; any other memory access corresponds to a Rep in r. As before the amount of nondeterminism is completely captured by the oracles. The output history op is a function of the input history ip for each choice of p and r such that the first and third conjuncts hold. The merge of the input and output histories is captured by the auxiliary function RepMerged which is defined as follows.

RepMerged(ap, a, p, r)

$(\alpha^\omega)^n \times \alpha^\omega \times [1..n]^\infty \times \mathrm{RpEx}^\infty \to \mathbb{B}$

$\forall\, k \in [1..n] : ap_k = \Pi_1.[\,(\alpha \times \{k\} \times \{\mathrm{Exit}\})Ⓢ(a, p, r)\,]$

It differs from the earlier merge function in that it considers only the entries whose element in r is Exit. In other words, we filter away those elements of i and o that correspond to Rep in r. The third conjunct in the I/O-relation of RepMc makes sure that also those entries that correspond to Rep in r are related to i in the correct way. In other words, that the memory accesses which correspond to Rep are compatible with the input streams, and that for each call the entry representing the last memory access has Exit as its element in r. This is captured by the auxiliary function defined below.

Compatible(i, p, r)

$$\text{Call}^{\omega} \times [1..n]^{\infty} \times \text{RpEx}^{\infty} \to \mathbb{B}$$

$$\forall\, j \in \mathsf{dom}(i) : r[j] = \text{Rep} \Rightarrow$$
$$\exists\, l \in \mathsf{dom}(i) :$$
$$l > j \wedge i[l] = i[j] \wedge p[l] = p[j] \wedge r[l] = \text{Exit}$$
$$\forall\, t \in \mathbb{N} : j < t < l \Rightarrow r[t] \neq \text{Exit} \vee p[t] \neq p[j]$$

Although RepMc is closer to the reliable memory component of [BL] than ConMc, some requirements are still missing. Firstly, as already mentioned in the introduction, we do not impose the hand-shake protocol on which [BL] is based. Our approach is based on asynchronous communication with unbounded buffering. Thus, we do not need the the environment assumption that a user never sends a new call before the memory component has issued a reply to its previous call.

However, this is not the only respect in which RepMc differs from the reliable memory component of [BL]; it also differs in the sense that it does not require the memory accesses to take place in the time interval between the arrival of the corresponding call and the issue of its reply. Moreover, RepMc does not say anything about the timing of the output with respect to the timing of the input. We now show how the missing causality and timing requirements can be imposed. To do so, we first have to explain what we mean by a timed stream.

3.5 Timed Streams and Operators on Timed Streams

To express *timing* constraints and also *causality* requirements between the input and output streams we use *timed streams*. A timed stream is a finite or infinite sequence of messages and *time ticks*. A time tick is represented by $\surd$. The interval between two consecutive ticks represents the least unit of time. A tick occurs in a timed stream at the end of each time unit.

An infinite timed stream represents a *complete* communication history; a finite timed stream represents a *partial* communication history. Since time never halts, any infinite timed stream is required to have *infinitely* many ticks. We do not want timed streams to end in the middle of a time unit. Thus, we insist that a timed stream is either empty, infinite or *ends* with a tick. For any set of messages M, by $M^{\underline{\infty}}$, $M^{\underline{*}}$ and $M^{\underline{\omega}}$ we denote respectively the set of all infinite timed streams over M, the set of all finite timed streams over M, and the set of all finite and infinite timed streams over M.

All the operators for untimed streams are also defined for timed streams. Their definitions are the the same as before given that $\surd$ is interpreted as an ordinary message. We also introduce some operators specially designed for timed streams. Let $r \in M^{\underline{\omega}}$ and $j \in \mathbb{N}_{\infty}$:

- $\overline{r}$ denotes the result of *removing* all ticks in r. Thus, $\overline{\langle a, \surd, b, \surd \rangle} = \langle a, b \rangle$. This operator is overloaded to tuples of timed streams in the obvious way.
- $\mathsf{tm}(r, j)$ denotes the time unit in which the jth message ($\surd$ is not a message) of r occurs if $j \in \mathsf{dom}(\overline{r})$. For example, if $r = \langle a, b, \surd, \surd, b, \surd \rangle$ then $\mathsf{tm}(r, 1) = \mathsf{tm}(r, 2) = 1$ and $\mathsf{tm}(r, 3) = 3$.
- $r{\downarrow_j}$ denotes the prefix of r characterizing the behavior until the end of time unit j. This means that $r{\downarrow_j}$ denotes r if j is greater than the number of ticks in r, and the shortest prefix of r containing j ticks, otherwise. Note that $r{\downarrow_\infty} = r$ and also that $r{\downarrow_0} = \langle\rangle$. Note also the way $\downarrow$ differs from $|$. For example, if $r = \langle a, b, \surd, c, \surd \rangle$ then $r{\downarrow_1} = \langle a, b, \surd \rangle$ and $r|_1 = \langle a \rangle$.

We also introduce a specially designed *filtration* operator. For any set of pairs of messages A ($\surd$ is not a message), timed infinite stream $t \in M^{\underline{\infty}}$, untimed infinite stream $s \in M^\infty$, let $A\circledR(t, s)$ denote the timed infinite stream such that

$$\overline{A\circledR(t, s)} = \Pi_1.[\, A\circledS(\overline{t}, s)\,]$$

$$\forall\, j \in \mathbb{N} : \#\overline{(A\circledR(t, s)){\downarrow_j}} = \#\{m \in \mathsf{dom}(\overline{t{\downarrow_j}}) \mid (\overline{t}[m], s[m]) \in A\}$$

Roughly speaking, $A\circledR(t, s)$ denotes the timed infinite stream obtained from t by removing the messages for which there are no corresponding pairs in A with respect to the untimed stream s. For example, if

$$A = \{(a, a), (b, b)\}$$

$$t = \langle a, b, \surd, a, b, \surd \rangle \frown \surd^\infty$$

$$s = \langle a, a \rangle \frown b^\infty$$

then

$$A\circledR(t, s) = \langle a, \surd, b, \surd \rangle \frown \surd^\infty$$

3.6 The Reliable Memory Component

As explained above, the component specified by RepMc is quite close to the reliable memory component described in [BL]. However, we still have to impose the requirement that any memory access takes place in the time interval between the transmission of the corresponding call and the issue of its reply, and that the reply is issued first after the call is transmitted. To do so, we introduce another oracle t. Informally speaking, it assigns a time stamp to each memory access. The specification of the reliable memory component is given below.

RelMc — time_dependent

in $ip \in (\mathrm{Call}^{\underline{\infty}})^n$

out $op \in (\mathrm{Rpl}^{\underline{\infty}})^n$

$\exists\, p \in [1..n]^\infty, r \in \mathrm{RpEx}^\infty, t \in \mathbb{N}^\infty, i \in \mathrm{Call}^\omega, o \in \mathrm{Rpl}^\omega :$

$\mathrm{RepMerged}(\overline{ip}, i, p, r)$

$\mathrm{RepMerged}(\overline{op}, o, p, r)$

$\mathrm{Compatible}(i, p, r)$

$\mathrm{Timed}(ip, op, p, r, t, i)$

$R_{\mathrm{SeqMc}(i,o)}$

In contrast to earlier the input and output streams are timed infinite streams. When this is the case we say that a specification is *time-dependent*. In a time-independent specification we represent the input and output histories by untimed streams; in a time-dependent specification we use timed infinite streams. Thus, strictly speaking, the keyword occurring in the frame is redundant; it is used to emphasize something that is already clear from the declarations of the input and output streams.

The reason why we allow the the input and output streams to be finite in the time-independent case is that a timed infinite stream with only finitely many ordinary messages degenerates to a finite stream when the time information is removed. For example $\overline{\surd^\infty} = \langle\rangle$. In fact, any time-independent specification S can be understood as syntactic sugar for a time-dependent specification S'; namely the time-dependent specification obtained from S by replacing the keyword by time-dependent; replacing any occurrence of $^\omega$ by $^{\underline{\infty}}$ in the declarations of the input and output streams; replacing any free occurrence of any input or output stream v in the I/O-relation by $\overline{v}$.

In the specification of the reliable memory component there is one "new" conjunct with respect to the RelMc. It requires the externally observable behavior to be *compatible* with an interpretation where any memory access takes place in the time interval between the transmission of the corresponding call and the issue of its reply. Moreover, it requires the reply to be delayed by at least one time unit with respect to the transmission of the call. The auxiliary function Timed is defined below.

Timed(ip, op, p, r, t, i)

$$(\text{Call}^{\underline{\infty}})^n \times (\text{Rpl}^{\underline{\infty}})^n \times [1..n]^\infty \times \text{RpEx}^\infty \times \mathbb{N}^\infty \times \text{Call}^\omega \to \mathbb{B}$$

$$\forall\, j \in \mathbb{N}_+ : t[j] \leq t[j+1]$$

$$\forall\, j \in \mathsf{dom}(i) :$$

$$\quad \mathsf{let}\ k = p[j]$$

$$\qquad w = \Pi_1.[\,(\text{RpEx} \times \{k\})\circledS(r|_j, p)\,]$$

$$\qquad m = \text{CallNumb}(w, \{\text{Exit}\})$$

$$\quad \mathsf{in}\ \mathsf{tm}(ip_k, m) < t[j] \leq \mathsf{tm}(op_k, m)$$

The first conjunct states that the infinite stream of time stamps is nondecreasing. The let construct introduces three variables: the channel number k of the considered memory access; the sub-stream w of r containing the entries for the channel k until this memory access; the number m of the call transmitted on the channel ip_k for which this memory access is performed. Thus, the second conjunct requires each memory access to take place between the transmission of the corresponding call and the issue of its reply. Note that this requirement also makes sure that there is a delay of at least one time unit between the transmission of a call and the issue of its reply. This is a sensible requirement under the assumption that the least unit of time is chosen sufficiently small.

The auxiliary function CallNumb is defined below.

CallNumb(w, A)

$$\alpha^\omega \times \mathbb{P}(\alpha) \to \mathbb{N}$$

$$\mathsf{let}\ n = \#(A\circledS w)\ \mathsf{in}\ \mathsf{if}\ w[\#w] \in A\ \mathsf{then}\ n\ \mathsf{else}\ n+1$$

α is a type variable. $\mathbb{P}(\alpha)$ denotes the power set of α.

3.7 The Unreliable Memory Component

The unreliable memory component (called the "memory component" in [BL]) differs from the reliable memory component in that calls may fail. Let

$$\text{OkFail} \stackrel{\mathsf{def}}{=} \{\text{Ok}, \text{Pfail}, \text{Tfail}\}$$

Once more an oracle is introduced. This time of type OkFail^∞. This new oracle q determines whether a memory access is successful or not. Any memory access whose corresponding element in q is Tfail has no effect on the memory. Thus, the difference between Pfail (partial failure) and Ok, on the one hand, and Tfail

(total failure), on the other hand, is that the latter has no effect on the memory locations. In addition, any call whose final memory access corresponds to a Pfail or a Tfail in q results in a MemFail. Note that a write call that fails may result in several memory updates, but it may also leave the memory unchanged.

UnrelMc — time_dependent

in $ip \in (\text{Call}^{\overline{\infty}})^n$

out $op \in (\text{Rpl}^{\overline{\infty}})^n$

$\exists\, p \in [1..n]^\infty, r \in \text{RpEx}^\infty, t \in \mathbb{N}^\infty, q \in \text{OkFail}^\infty, i \in \text{Call}^\omega, o \in \text{Rpl}^\omega :$

$\text{RepMerged}(\overline{ip}, i, p, r)$

$\text{RepMerged}(\overline{op}, o, p, r)$

$\text{Compatible}(i, p, r)$

$\text{Timed}(ip, op, p, r, t, i)$

$\text{UnrelSeqMc}(i, o, q)$

As before, for each interpretation of the oracles such that the first, third and fourth conjuncts hold, the output history is completely determined by the input history (if the timing of the output is ignored). The specification SeqMc is no longer sufficient to characterize the relationship between i and o. Instead we introduce an auxiliary function.

UnrelSeqMc(i, o, q)

$\text{Call}^\omega \times \text{Rpl}^\omega \times \text{OkFail}^\infty \to \mathbb{B}$

$\#o = \#i$

$\forall\, j \in \mathsf{dom}(o) :$

$q[j] = \text{Ok} \Rightarrow$

let $w = \Pi_1.[\,(\text{VldWr}(\text{Loc}(i[j])) \times \{\text{Ok}, \text{Pfail}\})\circledS(i|_j, q)\,]$

in $\text{SeqMcBehavior}(i[j], o[j], w)$

$q[j] \in \{\text{Pfail}, \text{Tfail}\} \Rightarrow o[j] = \text{MemFail}$

Note that w is defined to ignore memory accesses which correspond to Tfail. The rest is a straightforward adaptation of the SeqMc specification.

3.8 Implementation

In order to discuss whether RelMc is a *refinement* (or alternatively, an *implementation*) of UnrelMc or not, we have to define what we mean by refinement. Since any time-independent specification can be understood as syntactic sugar for an equivalent time-dependent specification, we consider only specifications written in the time-dependent format.

Let S and S' be specifications of the same syntactic interface. We define S' to be a *refinement* of S if the I/O-relation of S' implies the I/O-relation of S. More explicitly, if

$$R_{S'} \Rightarrow R_S$$

under the assumption that any free variable is typed in accordance with its declaration in the syntactic interface.

Since UnrelMc is equal to RelMc if q is fixed as Ok^∞, it follows that RelMc is a refinement of UnrelMc. It is also clear that UnrelMc allows an implementation which returns MemFail to any call — it is enough to fix q as Tfail^∞. It is straightforward to strengthen UnrelMc to avoid this. In fact the required behavior can be characterized by imposing additional constraints on the distribution of Ok's in q. This is the normal way of imposing fairness constraints in our method.

4 Problem II: The RPC Component

The second problem described in [BL] is to specify a *remote procedure call* component — from now on called the RPC component. As indicated by Fig. 2, the RPC component interfaces with two environment components, a sender and a receiver. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender.

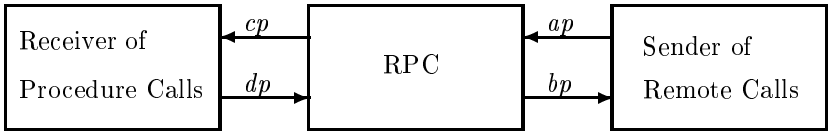


Fig. 2. The RPC Component and its Environment.

4.1 Basic Definitions

We first give some basic definitions.

$$
\begin{array}{ll}
\mathrm{PrCall} & \stackrel{\mathsf{def}}{=} \{p(a) \mid p \in \mathrm{Procs} \wedge \#a = \mathrm{ArgNum}(p)\} \\
\mathrm{ReCall} & \stackrel{\mathsf{def}}{=} \{\mathrm{RemoteCall}(p, a) \mid p \in \mathrm{P} \wedge a \in \mathrm{A}^*\} \\
\mathrm{VldReCall} & \stackrel{\mathsf{def}}{=} \{\mathrm{RemoteCall}(p, a) \mid p \in \mathrm{Procs} \wedge \#a = \mathrm{ArgNum}(p)\} \\
\mathrm{NotVldReCall} & \stackrel{\mathsf{def}}{=} \mathrm{ReCall} \setminus \mathrm{VldReCall} \\
\mathrm{ReRpl} & \stackrel{\mathsf{def}}{=} \mathrm{PrRpl} \cup \{\mathrm{RPCFail}, \mathrm{BadCall}\}
\end{array}
$$

P is a set of identifiers; A is a set of arguments. PrCall and ReCall are the sets of all possible procedure and remote calls, respectively. VldReCall is the set of all valid remote calls. PrRpl and ReRpl are the sets of all possible replies to procedure and remote calls, respectively. We assume that

$$\mathrm{PrRpl} \cap \{\mathrm{RPCFail}, \mathrm{BadCall}\} = \{\} \tag{2}$$

We also introduce three auxiliary functions.

$$
\begin{array}{l}
\mathrm{Map} \in \alpha^\omega \times (\alpha \to \beta) \to \beta^\omega \\
\quad \mathrm{Map}(a, m) \stackrel{\mathsf{def}}{=} b \\
\quad \mathsf{where}\ \#b = \#a \wedge \forall\, j \in \mathsf{dom}(b) : b[j] = m(a[j]) \\
\mathrm{Cl} \in \mathrm{VldReCall}^\omega \to \mathrm{PrCall}^\omega \\
\quad \mathrm{Cl}(e) \stackrel{\mathsf{def}}{=} \mathrm{Map}(e, m) \\
\quad \mathsf{where}\ m(\mathrm{RemoteCall}(p, a)) = p(a) \\
\mathrm{Rp} \in \mathrm{ReRpl}^\omega \to \mathrm{ReRpl}^\omega \\
\quad \mathrm{Rp}(e) \stackrel{\mathsf{def}}{=} \mathrm{Map}(e, m) \\
\quad \mathsf{where}\ m(r) = (\mathsf{if}\ r = \mathrm{RPCFail}\ \mathsf{then}\ \mathrm{MemFail}\ \mathsf{else}\ r)
\end{array}
$$

α, β are type variables.

4.2 The Sequential RPC Component

We first specify a *sequential* RPC component — in other words, an RPC component with respect to one user. As indicated by Fig. 3, the general (concurrent) RPC component can be thought of as a network of n sequential RPC components.

SeqRPC — time_dependent

in $a \in \mathrm{ReCall}^{\underline{\infty}}, d \in \mathrm{PrRpl}^{\underline{\infty}}$

out $b \in \mathrm{ReRpl}^{\underline{\infty}}, c \in \mathrm{PrCall}^{\underline{\infty}}$

$$\#\overline{d} = \#\overline{c}$$

$$\mathrm{Delayed}(c, d)$$

$$\Rightarrow$$

$$\exists\, q \in \mathrm{OkFail}^{\infty} :$$

$$\mathrm{RPCBehavior}(\overline{a}, \overline{d}, \overline{b}, \overline{c}, q)$$

$$\mathrm{RPCDelayed}(a, d, b, c, q)$$

Throughout the paper we use the convention that an indented implication operator on a separate line is the main operator of an I/O-relation. Thus, SeqRPC can be understood as an assumption/commitment specification. The antecedent characterizes the environment *assumption*, and the *commitment* is characterized by the consequent. If the environment behaves in accordance with the assumption then the specified component is required to behave in accordance with the commitment. The first conjunct of the assumption requires that the environment issues exactly one reply on d for each call received on c. The second imposes the constraint that the environment never sends a reply on d before at least one time unit after the call is transmitted on c. This is a sensible assumption given that the least unit of time is chosen sufficiently small. The auxiliary function Delayed is defined below.

Delayed(a, b)

$$\alpha^{\underline{\infty}} \times \alpha^{\underline{\infty}} \to \mathbb{B}$$

$$\forall\, j \in \mathbb{N} : \#\overline{(b{\downarrow_{j+1}})} \leq \#\overline{(a{\downarrow_{j}})}$$

α is a type variable. The function requires that for any j, the number of messages transmitted along b until time $j+1$ is less than or equal to the number of messages transmitted along a until time j.

The commitment employs an oracle q to determine whether a call terminates normally or with a BadCall exception (Ok); terminates with an RPCFail after having made the procedure call (Pfail); or terminates with an RPCFail without doing anything (Tfail). There are two main conjuncts; both represented by auxiliary functions. The first one is defined below.

RPCBehavior(a, d, b, c, q)

$\text{ReCall}^\omega \times \text{PrRpl}^\omega \times \text{ReRpl}^\omega \times \text{PrCall}^\omega \times \text{OkFail}^\infty \to \mathbb{B}$

$\forall\, k \in \mathsf{dom}(a) : q[k] = \text{Pfail} \Rightarrow a[k] \in \text{VldReCall}$

$\#b = \#a$

$\mathsf{let}\ (w, x) = (\text{VldReCall} \times \{\text{Ok}, \text{Pfail}\})\circledS(a, q)\ \mathsf{in}$

$c = \text{Cl}(w)$

$\text{PrRpl}\circledS b = \Pi_1.[\,(\text{PrRpl} \times \{\text{Ok}\})\circledS(d, x)\,]$

$\forall\, j \in \mathsf{dom}(b) :$

$q[j] \neq \text{Ok} \Rightarrow b[j] = \text{RPCFail}$

$q[j] = \text{Ok} \wedge a[j] \notin \text{VldReCall} \Rightarrow b[j] = \text{BadCall}$

Note that the types of a, d, b, c differ from their types in SeqRPC. There are four main conjuncts. The first conjunct makes sure that q is chosen in such a way that there is no invalid remote call in a whose corresponding element in q is Pfail. The second conjunct requires that a reply is issued for any remote call received.

The let construct of which the third conjunct consists defines two local variables w and x. w can be thought of as a buffer in which any remote call which leads to a procedure call is inserted. x contains the corresponding elements of q. Thus, $x \in \{\text{Ok}, \text{Pfail}\}^\omega$ since a valid call, whose corresponding element in p is Tfail, does not lead to a procedure call. The body of the let construct consists of two sub-conjuncts. The first sub-conjunct requires that the output along c is equal to the stream obtained by executing the call of every valid RemoteCall in w. The second sub-conjunct requires that the stream of PrRpl sent along b is equal to the stream of PrRpl received on d minus those replies which correspond to Pfail in x.

Also the fourth conjunct consists of two sub-conjuncts. The first sub-conjunct requires that any remote call, that fails, results in an RPCFail. The second sub-conjunct requires that any invalid RemoteCall, that does not fail, results in a BadCall.

The second main conjunct in the commitment of SeqRPC refers to the auxiliary function RPCDelayed. It makes sure that any response to a message is issued first one time unit after the message has been received. This function is defined below.

RPCDelayed(a, d, b, c, q)

$\mathrm{ReCall}^{\underline{\infty}} \times \mathrm{PrRpl}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{PrCall}^{\underline{\infty}} \times \mathrm{OkFail}^{\infty} \to \mathbb{B}$

$\mathrm{Delayed}((\mathrm{VldReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\})®(a, q), c)$

$\mathrm{Delayed}(a, b)$

$\mathrm{Delayed}(d, ((\mathrm{ReRpl} \setminus \{\mathrm{BadCall}\}) \times \{\mathrm{Ok}, \mathrm{Pfail}\})®(b, q))$

4.3 The RPC Component

The RPC component can be thought of as a network of n sequential RPC components (see Fig. 3). This network is specified as follows.

RPC — time_dependent

in	$ap \in (\mathrm{ReCall}^{\underline{\infty}})^n, dp \in (\mathrm{PrRpl}^{\underline{\infty}})^n$
out	$bp \in (\mathrm{ReRpl}^{\underline{\infty}})^n, cp \in (\mathrm{PrCall}^{\underline{\infty}})^n$

$\otimes_{j=1}^{n} \mathrm{SeqRPC}(ap_j, dp_j, bp_j, cp_j)$

The I/O-relation of RPC is equivalent to

$$\wedge_{j=1}^{n} R_{\mathrm{SeqRPC}(ap_j, dp_j, bp_j, cp_j)}$$

We use $\otimes$ instead of $\wedge$ to *emphasize* that we compose component specifications. The $\otimes$ operator is employed only when the specifications have mutually disjoint output alphabets — in other words, when for any pair of specifications the sets of identifiers representing output streams are disjoint. Since $\{bp_k, cp_k\}$ is disjoint from $\{bp_l, cp_l\}$ if $k \neq l$ this clearly holds in the case of RPC. We say that the specifications *interfere* if the sets of output identifiers are not mutually disjoint in this sense.

To implement such a *composite* specification it is enough to implement the n component specifications. These implementations can be designed independently of each other in a compositional manner. This is always the case when a network is specified by the $\otimes$ operator. Note the difference with respect to an ordinary *basic* specification based on auxiliary functions. For example, the five conjuncts of UnrelMc cannot be understood as independent component specifications. First of all, the oracles cannot easily be interpreted as local input and output channels. Secondly, the conjuncts interfere.

The RPC component as specified by us deviates from the informal specification in [BL]. Firstly, since we do not impose the hand-shake protocol, our specification does not disallow a sequential RPC component to transmit a new call before the environment has issued a reply to its previous call. Secondly, it

can be argued that the environment assumptions of the sequential RPC specifications are too strong. If the environment never replies to a call then we allow arbitrary behavior. However, since a component never knows in advance whether a reply eventually comes or not, it is not possible for a component to exploit this. After all, computer programs cannot predict the future. Thus, we do not see this as a problem.

5 Problem III: Implementing the Unreliable Memory Component

The unreliable memory component is implemented by combining the RPC component with the reliable memory component and a driver. We refer to the latter as the *clerk*. As in the case of the RPC component the clerk is divided into n sequential clerk components. The overall network is pictured in Fig. 3.

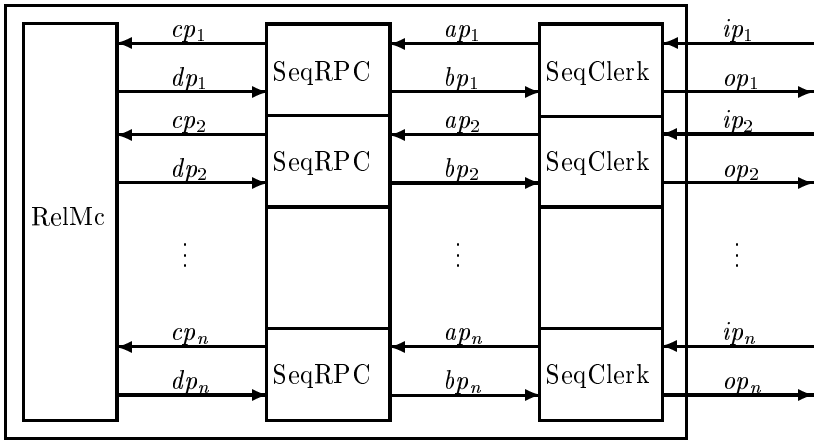


Fig. 3. Network Implementing the Unreliable Memory Component.

5.1 Basic Assumptions

In the following we assume that

$$\text{Call} \subseteq \text{PrCall} \tag{3}$$

$$\text{Rpl} \subseteq \text{PrRpl} \tag{4}$$

$$\{\text{Read}, \text{Write}\} \subseteq \text{Procs} \tag{5}$$

$$\text{ArgNum}(\text{Read}) = 1 \tag{6}$$

$$\mathrm{ArgNum}(\mathrm{Write}) = 2 \tag{7}$$

5.2 The Sequential Clerk Component

We first specify the sequential clerk component. It has the same main structure as the specification of the sequential RPC component.

SeqClerk — time_dependent

in $i \in \mathrm{Call}^{\underline{\infty}}, b \in \mathrm{ReRpl}^{\underline{\infty}}$

out $o \in \mathrm{ReRpl}^{\underline{\infty}}, a \in \mathrm{ReCall}^{\underline{\infty}}$

$\#\overline{b} = \#\overline{a}$

$\mathrm{Delayed}(a, b)$

$\Rightarrow$

$\exists\, r \in \mathrm{RpEx}^{\infty}$:

$\mathrm{ClerkBehavior}(\overline{i}, \overline{b}, \overline{o}, \overline{a}, r)$

$\mathrm{ClerkDelayed}(i, b, o, a, r)$

The environment assumption corresponds to that for SeqRPC. For each remote call sent along a the environment is assumed to send exactly one reply along b delayed by at least one time unit. In the commitment the oracle r is used to determine whether a call which results in an RPCFail should be retried or not. There are two sub-conjuncts. Both refer to auxiliary functions. The first one is defined below.

$\mathrm{ClerkBehavior}(i, b, o, a, r)$

$\mathrm{Call}^{\omega} \times \mathrm{ReRpl}^{\omega} \times \mathrm{ReRpl}^{\omega} \times \mathrm{ReCall}^{\omega} \times \mathrm{RpEx}^{\infty} \to \mathbb{B}$

$\forall\, j \in \mathsf{dom}(b) : r[j] = \mathrm{Rep} \Rightarrow$

$b[j] = \mathrm{RPCFail}$

$\exists\, l \in \mathsf{dom}(b) : l > j \wedge r[l] = \mathrm{Exit} \wedge \forall\, t \in \mathbb{N} : j < t \leq l \Rightarrow a[t] = a[j]$

let $(w, z) = \Pi_{1,2}.[\,(\mathrm{ReCall} \times \mathrm{ReRpl} \times \{\mathrm{Exit}\})\textcircled{S}(a, b, r)\,]$ in

$\mathrm{Cl}(w) = i$

$o = \mathrm{Rp}(z)$

Note that the types of i, b, o, a differ from their types in SeqClerk. There are two main conjuncts. The first conjunct requires that r is chosen in such a way

that each occurrence of a Rep in $r|_{\#b}$ corresponds to an RPCFail in b. It also makes sure that any remote call is repeated only a finite number of times.

The second conjunct consists of a let construct that defines w to be the stream of remote calls which correspond to Exit in r, and z to contain the corresponding replies. The body consists of two sub-conjuncts. The first requires i to be equal to w if each remote call in w is replaced by the corresponding call. The second requires that the stream of replies sent along o is equal to z given that each RPCFail is replaced by MemFail.

The second conjunct in the commitment of SeqClerk refers to an auxiliary function ClerkDelayed. In the same way as RPCDelayed, it captures that there is a delay of at least one time unit between input and output.

ClerkDelayed(i, b, o, a, r)

$$\mathrm{Call}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{ReCall}^{\underline{\infty}} \times \mathrm{RpEx}^{\infty} \to \mathbb{B}$$

$$\mathrm{Delayed}((\mathrm{ReRpl} \times \{\mathrm{Exit}\})\textcircled{R}(b, r), o)$$

$$\forall\, j \in \mathsf{dom}(\overline{a}) : \mathsf{let}\ m = \mathrm{CallNumb}(r|_j, \{\mathrm{Exit}\})\ \mathsf{in}\ \mathsf{tm}(a, j) > \mathsf{tm}(i, m)$$

5.3 The Clerk Component

As already explained, the clerk can be understood as a network of n sequential clerk components. It is specified as follows.

Clerk — time_dependent

in	$ip \in (\mathrm{Call}^{\underline{\infty}})^n, bp \in (\mathrm{ReRpl}^{\underline{\infty}})^n$
out	$op \in (\mathrm{ReRpl}^{\underline{\infty}})^n, ap \in (\mathrm{ReCall}^{\underline{\infty}})^n$

$$\otimes_{j=1}^{n} \mathrm{SeqClerk}(ip_j, bp_j, op_j, ap_j)$$

5.4 The Implementation

The network pictured in Fig. 3 can now be specified as follows.

McImpl — time_dependent

in	$ip \in (\mathrm{Call}^{\underline{\infty}})^n$
out	$op \in (\mathrm{Rpl}^{\underline{\infty}})^n$
loc	$cp \in (\mathrm{Call}^{\underline{\infty}})^n, dp \in (\mathrm{Rpl}^{\underline{\infty}})^n, ap \in (\mathrm{ReCall}^{\underline{\infty}})^n, bp \in (\mathrm{ReRpl}^{\underline{\infty}})^n$

$$\mathrm{RelMc}(cp, dp) \otimes \mathrm{RPC}(ap, dp, bp, cp) \otimes \mathrm{Clerk}(ip, bp, op, ap)$$

cp, dp, ap, bp represent tuples of local channels. This motivates the keyword loc. We have already explained that $\otimes$ corresponds to conjunction. The logical interpretation of the loc declarations is existential quantification. Thus, the composite specification McImpl is equivalent to a basic specification with the same external interface and the following I/O-relation

$$\exists\, cp \in (\mathrm{Call}^{\overline{\infty}})^n, dp \in (\mathrm{Rpl}^{\overline{\infty}})^n, ap \in (\mathrm{ReCall}^{\overline{\infty}})^n, bp \in (\mathrm{ReRpl}^{\overline{\infty}})^n :$$

$$R_{\mathrm{RelMc}(cp,dp)} \wedge R_{\mathrm{RPC}(ap,dp,bp,cp)} \wedge R_{\mathrm{Clerk}(ip,bp,op,ap)}$$

5.5 Verification

We now prove that McImpl is a refinement of UnrelMc. Remember that for any specification S, we use R_S to denote its I/O-relation. If the I/O-relation has $\Rightarrow$ as its main operator, we use A_S to denote its antecedent (assumption) and C_S to denote its consequent (commitment). Although in most cases this is not pointed out explicitly, the validity of the many deductions depends on the type definitions listed in Sect. 3.1, 4.1, and also the assumptions 1-7 listed in Sect. 3.1, 4.1, 5.1. In the following we assume that

$$ip \in (\mathrm{Call}^{\overline{\infty}})^n \tag{8}$$

$$cp \in (\mathrm{Call}^{\overline{\infty}})^n \tag{9}$$

$$op \in (\mathrm{Rpl}^{\overline{\infty}})^n \tag{10}$$

$$dp \in (\mathrm{Rpl}^{\overline{\infty}})^n \tag{11}$$

$$ap \in (\mathrm{ReCall}^{\overline{\infty}})^n \tag{12}$$

$$bp \in (\mathrm{ReRpl}^{\overline{\infty}})^n \tag{13}$$

$$j \in [1..n] \tag{14}$$

It must be shown that

$$R_{\mathrm{RelMc}(cp,dp)} \wedge R_{\mathrm{RPC}(ap,dp,bp,cp)} \wedge R_{\mathrm{Clerk}(ip,bp,op,ap)} \Rightarrow R_{\mathrm{UnrelMc}(ip,op)} \tag{15}$$

From the definitions of RepMerged and Timed it follows straightforwardly that

$$R_{\mathrm{RelMc}(cp,dp)} \Rightarrow A_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)} \tag{16}$$

From the second conjunct of RPCBehavior and the second conjunct of RPCDelayed it follows that

$$A_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)} \wedge C_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)} \Rightarrow A_{\mathrm{SeqClerk}(ip_j,bp_j,op_j,ap_j)} \tag{17}$$

16, 17 imply it is enough to show that

$$R_{\mathrm{RelMc}(cp,dp)} \wedge (\wedge_{k=1}^{n} C_{\mathrm{SeqRPC}(ap_k,dp_k,bp_k,cp_k)} \wedge C_{\mathrm{SeqClerk}(ip_k,bp_k,op_k,ap_k)}) \Rightarrow R_{\mathrm{UnrelMc}(ip,op)} \tag{18}$$

Let ip, cp, op, dp, ap, bp be such that

$$R_{\mathrm{RelMc}(cp,dp)} \tag{19}$$

$$C_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)} \tag{20}$$

$$C_{\mathrm{SeqClerk}(ip_j,bp_j,op_j,ap_j)} \tag{21}$$

It must be shown that

$$R_{\mathrm{UnrelMc}(ip,op)} \tag{22}$$

19 implies there are $p' \in [1..n]^\infty$, $r' \in \mathrm{RpEx}^\infty$, $t' \in \mathbb{N}^\infty$, $i' \in \mathrm{Call}^\omega$, $o' \in \mathrm{Rpl}^\omega$ such that

$$\mathrm{RepMerged}(\overline{cp}, i', p', r') \tag{23}$$

$$\mathrm{RepMerged}(\overline{dp}, o', p', r') \tag{24}$$

$$\mathrm{Compatible}(i', p', r') \tag{25}$$

$$\mathrm{Timed}(cp, dp, p', r', t', i') \tag{26}$$

$$\mathrm{SeqMc}(i', o') \tag{27}$$

20 implies there are $q_1, .., q_n \in \mathrm{OkFail}^\infty$ such that

$$\mathrm{RPCBehavior}(\overline{ap_j}, \overline{dp_j}, \overline{bp_j}, \overline{cp_j}, q_j) \tag{28}$$

$$\mathrm{RPCDelayed}(ap_j, dp_j, bp_j, cp_j, q_j) \tag{29}$$

21 implies there are $r_1, .., r_n \in \mathrm{RpEx}^\infty$ such that

$$\mathrm{ClerkBehavior}(\overline{ip_j}, \overline{bp_j}, \overline{op_j}, \overline{ap_j}, r_j) \tag{30}$$

$$\mathrm{ClerkDelayed}(ip_j, bp_j, op_j, ap_j, r_j) \tag{31}$$

22 follows if we can find $p \in [1..n]^\infty$, $r \in \mathrm{RpEx}^\infty$, $t \in \mathbb{N}^\infty$, $q \in \mathrm{OkFail}^\infty$, $i \in \mathrm{Call}^\omega$, $o \in \mathrm{Rpl}^\omega$ such that

$$\mathrm{RepMerged}(\overline{ip}, i, p, r) \tag{32}$$

$$\mathrm{RepMerged}(\overline{op}, o, p, r) \tag{33}$$

$$\mathrm{Compatible}(i, p, r) \tag{34}$$

$$\mathrm{Timed}(ip, op, p, r, t, i) \tag{35}$$

$$\mathrm{UnrelSeqMc}(i, o, q) \tag{36}$$

The remaining of the proof is structured as follows. We first argue the existence of oracles p, r, t, q, i, o satisfying a number of useful properties. Then we show the validity of 32-36.

Definition of oracles: We start by defining the new oracles p, t, i. Informally speaking, they are constructed from the oracles $p', r', t', i', q_1, .., q_n$ and the communication history ap by inserting one new entry for each call that reaches the RPC component but is not forwarded to the reliable memory component — in other words, for each call received on ap that is not forwarded along cp. It follows from 28 that the only calls that reach the jth sequential RPC component without being forwarded to the reliable memory component are those calls

- whose corresponding element in q_j is Tfail,
- that are contained in NotVldReCall and whose corresponding element in q_j is Ok.

3, 5, 6, 7, 8, 30 imply that each remote call is a valid remote call. Thus, it is enough to insert new entries for those elements of ap_j whose corresponding elements in q_j are Tfail. Let $p \in [1..n]^\infty$, $t \in \mathbb{N}^\infty$, $i \in \text{Call}^\omega$, $y \in \text{RpExIns}^\infty$, where $\text{RpExIns} \stackrel{\text{def}}{=} \{\text{Ins}, \text{Rep}, \text{Exit}\}$, be such that

$$i' = \Pi_1.[\,(\text{Call} \times \text{RpEx})Ⓢ(i, y)\,] \tag{37}$$

$$(p', r', t') = ([1..n] \times \text{RpEx} \times \mathbb{N})Ⓢ(p, y, t) \tag{38}$$

The idea is that any inserted call has Ins as its corresponding element in y. Any other call corresponds to Exit or Rep in y depending on whether the call's corresponding element in r' is Exit or Rep. Since 37, 38 hold if $p = p', t = t', i = i', y = r'$ it follows that such oracles exist. The fact that 37, 38 do not constrain entries whose element in y is Ins imply that this is still the case if we also impose the requirement that

$$\begin{aligned}&\text{Cl}(\Pi_1.[\,(\text{ReCall} \times \{\text{Tfail}\})Ⓢ(\overline{ap_j}, q_j)\,]) = \\ &\quad \Pi_1.[\,(\text{Call} \times \{j\} \times \{\text{Ins}\})Ⓢ(i, p, y)\,]\end{aligned} \tag{39}$$

39 allows us to insert the new entries at arbitrary positions in the old oracles as long as the order of the new entries is maintained. This is obviously too liberal. Give that $\text{ExIns} \stackrel{\text{def}}{=} \{\text{Exit}, \text{Ins}\}$, we therefore also require that

$$\begin{aligned}&\forall\, l \in \mathbb{N}_+ : \\ &\quad \textsf{let } k \;= p[l] \\ &\qquad\; w \;= \Pi_1.[\,(\text{RpExIns} \times \{k\})Ⓢ(y|_l, p)\,] \\ &\qquad\; m \;= \text{CallNumb}(w, \text{ExIns}) \\ &\quad \textsf{in } y[l] = \text{Ins} \Rightarrow q_k[m] = \text{Tfail}\end{aligned} \tag{40}$$

40 guarantees that each Ins entry is correctly ordered with respect to the Exit entries in r'. To make sure that the Ins entries are correctly ordered with respect to the old Rep entries we also assume that

$$\forall\, l \in \mathbb{N}_+ : \tag{41}$$

$$\begin{array}{l}\mathsf{let}\ k\ = p[l]\\ \quad w = \mathit{\Pi}_1.[\,(\mathrm{RpExIns} \times \{k\})Ⓢ(y|_{l-1}, p)\,]\\ \mathsf{in}\ y[l] = \mathrm{Ins} \Rightarrow w = \langle\rangle \vee w[\#w] \neq \mathrm{Rep}\end{array}$$

Thus, an Ins entry does not interfere with the memory accesses belonging to the next call by the same user. 26 implies

$$\begin{array}{l}\forall\, l \in \mathsf{dom}(i') :\\ \quad \mathsf{let}\ k\ = p'[l]\\ \quad\quad w\ = \mathit{\Pi}_1.[\,(\mathrm{RpEx} \times \{k\})Ⓢ(r'|_l, p')\,]\\ \quad\quad m\ = \mathrm{CallNumb}(w, \{\mathrm{Exit}\})\\ \quad \mathsf{in}\ \mathsf{tm}(cp_k, m) < t'[l] \leq \mathsf{tm}(dp_k, m)\end{array} \tag{42}$$

37, 38, 42 imply

$$\begin{array}{l}\forall\, l \in \mathsf{dom}(i) : y[l] \neq \mathrm{Ins} \Rightarrow\\ \quad \mathsf{let}\ k\ = p[l]\\ \quad\quad w\ = \mathit{\Pi}_1.[\,(\mathrm{RpExIns} \times \{k\})Ⓢ(y|_l, p)\,]\\ \quad\quad m\ = \mathrm{CallNumb}(w, \{\mathrm{Exit}\})\\ \quad \mathsf{in}\ \mathsf{tm}(cp_k, m) < t[l] \leq \mathsf{tm}(dp_k, m)\end{array} \tag{43}$$

29 implies

$$\mathrm{Delayed}((\mathrm{VldReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(ap_j, q_j), cp_j) \tag{44}$$

$$\mathrm{Delayed}(dp_j, ((\mathrm{ReRpl} \setminus \{\mathrm{BadCall}\}) \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(bp_j, q_j)) \tag{45}$$

3, 5, 6, 7, 8, 30 imply that any remote call is a valid remote. Thus, 44, 45 imply

$$\mathrm{Delayed}((\mathrm{ReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(ap_j, q_j), cp_j) \tag{46}$$

$$\mathrm{Delayed}(dp_j, (\mathrm{ReRpl} \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(bp_j, q_j)) \tag{47}$$

39, 40, 43, 46, 47 imply

$$\begin{array}{l}\forall\, l \in \mathsf{dom}(i) : y[l] \neq \mathrm{Ins} \Rightarrow\\ \quad \mathsf{let}\ k\ = p[l]\\ \quad\quad w\ = \mathit{\Pi}_1.[\,(\mathrm{RpExIns} \times \{k\})Ⓢ(y|_l, p)\,]\\ \quad\quad m\ = \mathrm{CallNumb}(w, \mathrm{ExIns})\\ \quad \mathsf{in}\ \mathsf{tm}(ap_k, m) < t[l] \leq \mathsf{tm}(bp_k, m)\end{array} \tag{48}$$

26, 37, 38 imply

$$\forall\, k, l \in \mathbb{N}_+ : k < l \wedge \{y[k], y[l]\} \subseteq \mathrm{RpEx} \Rightarrow t[k] \leq t[l] \tag{49}$$

28 and 29 guarantee that the replies to the calls received on ap_j are output along bp_j in the FIFO manner and with a delay of at least one time unit. Moreover, 41 implies that the inserted Ins entries do not interfere with the memory accesses of the other calls by the same user. Thus, since neither of the requirements imposed on the new oracles so far say anything about the entries in t that correspond to Ins in y, it follows from 48, 49 that we may also assume that

$$\forall\, l \in \mathbb{N}_+ : t[l] \leq t[l+1] \tag{50}$$

$$\forall\, l \in \mathsf{dom}(i) : \tag{51}$$

$$\begin{aligned}
&\mathsf{let}\ k\ = p[l] \\
&\qquad w\ = \Pi_1.[\,(\mathrm{RpExIns} \times \{k\})Ⓢ(y|_l, p)\,] \\
&\qquad m\ = \mathrm{CallNumb}(w, \mathrm{ExIns}) \\
&\mathsf{in}\ \mathsf{tm}(ap_k, m) < t[l] \leq \mathsf{tm}(bp_k, m)
\end{aligned}$$

We now chose $q \in \mathrm{OkFail}^\infty$ and $r \in \mathrm{RpEx}^\infty$ such that

$$\Pi_1.[\,(\mathrm{OkFail} \times \mathrm{ExIns} \times \{j\})Ⓢ(q, y, p)\,] \sqsubseteq q_j \tag{52}$$

$$\Pi_1.[\,(\mathrm{OkFail} \times \{\mathrm{Rep}\})Ⓢ(q, y)\,] \sqsubseteq \mathrm{Ok}^\infty \tag{53}$$

$$\Pi_1.[\,(\mathrm{RpEx} \times \mathrm{ExIns} \times \{j\})Ⓢ(r, y, p)\,] \sqsubseteq r_j \tag{54}$$

$$\Pi_1.[\,(\mathrm{RpEx} \times \{\mathrm{Rep}\})Ⓢ(r, y)\,] \sqsubseteq \mathrm{Rep}^\infty \tag{55}$$

Since 37-51 do not constrain q, r, they are clearly not in conflict with 52-55. Moreover, since 52, 53 do not constrain r, and 54, 55 do not constrain q, it follows that 52, 53 are not in conflict with 54, 55. Finally, since 52, 54 only constrain those entries that are marked by Exit or Ins in y, and since 53, 55 only constrain those entries that are marked by Rep in y, it follows that 52, 54 are not in conflict with 53, 55. Finally, we define $o \in \mathrm{Rpl}^\omega$ to be such that

$$\mathrm{Rp}(\overline{bp_j}) = \Pi_1.[\,(\mathrm{Rpl} \times \{j\} \times \mathrm{ExIns})Ⓢ(o, p, y)\,] \tag{56}$$

$$(\mathrm{Rpl} \times \{\mathrm{Rep}\})Ⓢ(o', r') = (\mathrm{Rpl} \times \{\mathrm{Rep}\})Ⓢ(o, y) \tag{57}$$

Since the earlier constraints do not refer to o, and since 56 and 57 clearly are not in conflict with each other, we have shown that there are type correct oracles p, r, t, q, i, o, y such that 37-57 hold.
It remains to prove that 32-36 hold.

Proof of 32: 23, 37, 38 imply

$$\overline{cp_j} = \Pi_1.[\,(\text{Call} \times \{j\} \times \{\text{Exit}\})Ⓢ(i,p,y)\,] \tag{58}$$

28 implies

$$\overline{cp_j} = \text{Cl}(\Pi_1.[\,(\text{VldReCall} \times \{\text{Ok},\text{Pfail}\})Ⓢ(\overline{ap_j},q_j)\,]) \tag{59}$$

3, 5, 6, 7, 8, 30 imply that each remote call is a valid remote call. Thus, 59 implies

$$\overline{cp_j} = \text{Cl}(\Pi_1.[\,(\text{ReCall} \times \{\text{Ok},\text{Pfail}\})Ⓢ(\overline{ap_j},q_j)\,]) \tag{60}$$

39, 40, 58, 60 imply

$$\text{Cl}(\overline{ap_j}) = \Pi_1.[\,(\text{Call} \times \{j\} \times \text{ExIns})Ⓢ(i,p,y)\,] \tag{61}$$

61 implies

$$\Pi_1.[\,(\text{Call} \times \{\text{Exit}\})Ⓢ(\text{Cl}(\overline{ap_j}),r_j)\,] = \tag{62}$$
$$\Pi_1.[\,(\text{Call} \times \{\text{Exit}\})Ⓢ(\Pi_1.[\,(\text{Call} \times \{j\} \times \text{ExIns})Ⓢ(i,p,y)\,],r_j)\,]$$

Clearly

$$\Pi_1.[\,(\text{Call} \times \{\text{Exit}\})Ⓢ(\text{Cl}(\overline{ap_j}),r_j)\,] = \tag{63}$$
$$\text{Cl}(\Pi_1.[\,(\text{ReCall} \times \{\text{Exit}\})Ⓢ(\overline{ap_j},r_j)\,])$$

30 implies

$$\overline{ip_j} = \text{Cl}(\Pi_1.[\,(\text{ReCall} \times \{\text{Exit}\})Ⓢ(\overline{ap_j},r_j)\,]) \tag{64}$$

62, 63, 64 imply

$$\overline{ip_j} = \tag{65}$$
$$\Pi_1.[\,(\text{Call} \times \{\text{Exit}\})Ⓢ(\Pi_1.[\,(\text{Call} \times \{j\} \times \text{ExIns})Ⓢ(i,p,y)\,],r_j)\,]$$

54, 55, 65 imply

$$\overline{ip_j} = \Pi_1.[\,(\text{Call} \times \{j\} \times \{\text{Exit}\})Ⓢ(i,p,r)\,] \tag{66}$$

65 implies 32.

Proof of 33: 30 implies

$$\overline{op_j} = \text{Rp}(\Pi_1.[\,(\text{ReRpl} \times \{\text{Exit}\})Ⓢ(\overline{bp_j},r_j)\,]) \tag{67}$$

56, 67 imply

$$\overline{op_j} = \Pi_1.[\,(\text{Rpl} \times \{\text{Exit}\})Ⓢ \tag{68}$$
$$(\Pi_1.[\,(\text{Rpl} \times \{j\} \times \text{ExIns})Ⓢ(o,p,y)\,],r_j)\,]$$

54, 55, 68 imply

$$\overline{op_j} = \Pi_1.[\,(\text{Rpl} \times \{j\} \times \{\text{Exit}\})Ⓢ(o,p,r)\,] \tag{69}$$

69 implies 33.

Proof of 34: 25, 37, 38, 41 imply

$$\forall\, k \in \mathsf{dom}(i) : y[k] = \mathrm{Rep} \Rightarrow \tag{70}$$
$$\quad \exists\, l \in \mathsf{dom}(i) :$$
$$\quad\quad l > k \wedge i[l] = i[k] \wedge p[l] = p[k] \wedge y[l] = \mathrm{Exit}$$
$$\quad\quad \forall\, t \in \mathbb{N} : k < t < l \Rightarrow y[t] \neq \mathrm{Exit} \vee p[t] \neq p[k]$$

28, 30 imply

$$\forall\, k \in \mathsf{dom}(\overline{ap_j}) : r_j[j] = \mathrm{Rep} \Rightarrow \tag{71}$$
$$\quad \exists\, l \in \mathsf{dom}(\overline{ap_j}) : l > k \wedge r_j[l] = \mathrm{Exit}$$
$$\quad\quad \forall\, t \in \mathbb{N} : k < t \leq l \Rightarrow \overline{ap_j}[t] = \overline{ap_j}[k]$$

54, 55, 61, 70, 71 imply

$$\forall\, k \in \mathsf{dom}(i) : r[k] = \mathrm{Rep} \Rightarrow \tag{72}$$
$$\quad \exists\, l \in \mathsf{dom}(i) :$$
$$\quad\quad l > k \wedge i[l] = i[k] \wedge p[l] = p[k] \wedge r[l] = \mathrm{Exit}$$
$$\quad\quad \forall\, t \in \mathbb{N} : k < t < l \Rightarrow y[t] \neq \mathrm{Exit} \vee p[t] \neq p[k]$$

72 implies 34.

Proof of 35: 31 implies

$$\mathrm{Delayed}((\mathrm{ReRpl} \times \{\mathrm{Exit}\})®(bp_j, r_j), op_j) \tag{73}$$
$$\forall\, k \in \mathsf{dom}(\overline{ap_j}) : \tag{74}$$
$$\quad \mathsf{let}\ m = \mathrm{CallNumb}(r_j|_k, \{\mathrm{Exit}\})\ \mathsf{in}\ \mathsf{tm}(ap_j, k) > \mathsf{tm}(ip_j, m)$$

51, 54, 55, 73, 74 imply

$$\forall\, l \in \mathsf{dom}(i) : \tag{75}$$
$$\quad \mathsf{let}\ k\ = p[l]$$
$$\quad\quad w\ = \Pi_1.[\,(\mathrm{RpEx} \times \{k\})Ⓢ(r|_l, p)\,]$$
$$\quad\quad m\ = \mathrm{CallNumb}(w, \{\mathrm{Exit}\})$$
$$\quad \mathsf{in}\ \mathsf{tm}(ip_k, m) < t[l] \leq \mathsf{tm}(op_k, m)$$

50, 75 imply 35.

Proof of 36: 28 implies

$$\#\overline{ap_j} = \#\overline{bp_j} \tag{76}$$

37, 38, 56, 57, 61, 76 imply

$$\#o = \#i \tag{77}$$

27 implies

$$\forall\, l \in \mathsf{dom}(o') : \tag{78}$$

$$\quad \mathsf{let}\ w = [\,\mathrm{VldWr}(\mathrm{Loc}(i'[l]))Ⓢ(i'|_l)\,]$$

$$\quad \mathsf{in}\ \mathrm{SeqMcBehavior}(i'[l], o'[l], w)$$

We want to prove that

$$\forall\, k \in \mathsf{dom}(o) : \tag{79}$$

$$\quad q[k] = \mathrm{Ok} \Rightarrow$$

$$\quad\quad \mathsf{let}\ w = \Pi_1.[\,(\mathrm{VldWr}(\mathrm{Loc}(i[k])) \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓢ(i|_k, q)\,]$$

$$\quad\quad \mathsf{in}\ \mathrm{SeqMcBehavior}(i[k], o[k], w)$$

$$\quad q[k] \in \{\mathrm{Pfail}, \mathrm{Tfail}\} \Rightarrow o[k] = \mathrm{MemFail}$$

Let

$$k \in \mathsf{dom}(o) \tag{80}$$

$$l = p[k] \tag{81}$$

$$w = \Pi_1.[\,(\mathrm{RpExIns} \times \{l\})Ⓢ(y|_k, p)\,] \tag{82}$$

$$m = \mathrm{CallNumb}(w, \mathrm{ExIns}) \tag{83}$$

39, 40, 52, 53, 77 imply

$$\forall\, t \in \mathsf{dom}(o) : q[t] = \mathrm{Tfail} \Leftrightarrow y[t] = \mathrm{Ins} \tag{84}$$

There are three cases to consider.

Case 1: Assume

$$y[k] = \mathrm{Rep} \tag{85}$$

37 implies

$$i[k] = i'[k - \#(\{\mathrm{Ins}\}Ⓢ(y|_k))] \tag{86}$$

38, 57 imply

$$o[k] = o'[k - \#(\{\mathrm{Ins}\}Ⓢ(y|_k))] \tag{87}$$

53 implies

$$q[k] = \mathrm{Ok} \tag{88}$$

78, 84, 86, 87, 88 imply 79.

Case 2: Assume

$$y[k] = \mathrm{Ins} \tag{89}$$

52, 81, 82, 83, 84, 89 imply

$$q[k] = q_l[m] = \mathrm{Tfail} \tag{90}$$

28, 56, 90 imply 79.

Case 3: Assume

$$y[k] = \text{Exit} \tag{91}$$

52, 81, 82, 83, 84, 91 imply

$$(q[k] = q_l[m] = \text{Pfail}) \vee (q[k] = q_l[m] = \text{Ok}) \tag{92}$$

If the first disjunct holds, then 28, 56 imply 79. Assume

$$q[k] = q_l[m] = \text{Ok} \tag{93}$$

37, 91 imply

$$i[k] = i'[k - \#(\{\text{Ins}\}\circledS(y|_k))] \tag{94}$$

28, 38, 56, 91, 93 imply

$$o[k] = o'[k - \#(\{\text{Ins}\}\circledS(y|_k))] \tag{95}$$

78, 84, 93, 94, 95 imply 79.

6 Problem IV: The Lossy RPC Component

We now specify the *lossy* RPC component. We consider only the *sequential* case — in other words, a lossy RPC component with respect to one user. We refer to this specification as SeqLossyRPC. In the same way as we specified the RPC component by the conjunction of n SeqRPC specifications, we may specify the lossy RPC component by the conjunction of n SeqLossyRPC specifications. However, this is trivial and therefore left out.

6.1 The Sequential Lossy RPC Component

The specification of the sequential lossy RPC component is given below.

SeqLossyRPC — time_dependent

in $r \in \text{ReCall}^{\underline{\infty}}, d \in \text{PrRpl}^{\underline{\infty}}$

out $s \in \text{ReRpl}^{\underline{\infty}}, c \in \text{PrCall}^{\underline{\infty}}$

$\#\overline{d} = \#\overline{c}$

$\text{Delayed}(c, d)$

$\Rightarrow$

$\exists\, q \in \text{OkFail}^{\infty}, s' \in \text{ReRpl}^{\underline{\infty}}:$

$s = ((\text{ReRpl} \cup \{\surd\}) \setminus \{\text{RPCFail}\})\circledS s'$

$\text{RPCBehavior}(\overline{r}, \overline{d}, \overline{s'}, \overline{c}, q)$

$\text{RPCDelayed}(r, d, s', c, q)$

$\text{RPCBounded}(r, d, s', c, q)$

As for the sequential RPC component specified in Sect. 4.2, we assume that exactly one reply is received on d for each call sent along c. Moreover, we assume that each reply is received with a delay of at least one time unit with respect to the transmission of the call. The commitment is split into four conjuncts. The second and third conjunct correspond to the two conjuncts in the commitment of SeqRPC with the exception that s' is substituted for s. This means that s' contains an RPCFail for each remote call that fails. The informal description of the lossy RPC component disallows the output of RPCFail exceptions, which is why the first conjunct in the commitment of SeqLossyRPC defines s to be equal to s' minus the occurrences of RPCFail. The fourth conjunct refers to an auxiliary function which is defined below.

RPCBounded(r, d, s', c, q)

$$\mathrm{ReCall}^{\underline{\infty}} \times \mathrm{PrRpl}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{PrCall}^{\underline{\infty}} \times \mathrm{OkFail}^{\infty} \to \mathbb{B}$$

let

$$A = \mathrm{VldReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\}$$

$$B = (\mathrm{ReCall} \times \mathrm{OkFail}) \setminus A$$

$$C = \{\mathrm{BadCall}, \mathrm{RPCFail}\} \times \{\mathrm{Ok}, \mathrm{Tfail}\}$$

$$D = (\mathrm{ReRpl} \setminus \{\mathrm{BadCall}\}) \times \{\mathrm{Ok}, \mathrm{Pfail}\}$$

in

$$\mathrm{Bounded}(A®(r, q), c, \delta)$$

$$\mathrm{Bounded}(B®(r, q), C®(s', q), \delta)$$

$$\mathrm{Bounded}(d, D®(s', q), \delta)$$

For any j RPCBounded compares what has been sent along the output streams s' and c until time $j + \delta$ with what has been received on the input streams r and d until time j. δ stands for the number of time units that corresponds to the upper bound on the response time imposed by the problem statement.

The first conjunct makes sure that if a remote call leads to a procedure call then this procedure call takes place within δ time units after the transmission of the remote call.

The second conjunct makes sure that if a remote call leads to a BadCall then this reply is output within δ time units of the transmission of the remote call.

The third conjunct makes sure that if any other value than BadCall is returned then this takes place within δ time units of the reply to the corresponding procedure call.

Note that the second and third conjuncts also impose response time constraints on the occurrences of RPCFail in s'. These constraints have no effect on

the externally observable behavior of SeqLossyRPC. However, they simplify the proof in Sect. 7.3.

The auxiliary function Bounded is defined below.

Bounded(a, b, δ)

$\alpha^{\underline{\infty}} \times \alpha^{\underline{\infty}} \times \mathbb{N} \to \mathbb{B}$

$\forall j \in \mathbb{N} : \#(\overline{a{\downarrow_j}}) \leq \#(\overline{b{\downarrow_{j+\delta}}})$

α is a type variable.

7 Problem V: Implementing the RPC Component

The next task is to implement the sequential RPC component with the sequential lossy RPC component. We first specify the required driver, which we refer to as the sequential RPC clerk. It is connected to the lossy RPC component in accordance with Fig. 4. In the following we assume that $\sigma = 2\delta + \epsilon$, where ϵ

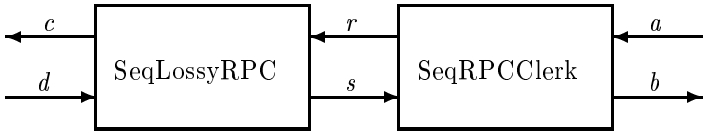


Fig. 4. Network Implementing the RPC Component.

is the number of time units corresponding to the upper bound on the response time of the environment assumed by the problem statement.

7.1 The Sequential RPC Clerk Component

The sequential RPC Clerk component is specified below.

SeqRPCClerk — time_dependent

in $a \in \text{ReCall}^{\underline{\infty}}, s \in \text{ReRpl}^{\underline{\infty}}$

out $b \in \text{ReRpl}^{\underline{\infty}}, r \in \text{ReCall}^{\underline{\infty}}$

RPCClerkBehavior(a, s, b, r)

RPCClerkDelayed(a, s, b, r)

The I/O-relation consists of two conjuncts captured by two auxiliary functions. The first one is defined below.

RPCClerkBehavior(a, s, b, r)

$$\mathrm{ReCall}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{ReRpl}^{\underline{\infty}} \times \mathrm{ReCall}^{\underline{\infty}} \to \mathbb{B}$$

$$\overline{r} = \overline{a}$$

$$(\mathrm{ReRpl} \setminus \{\mathrm{RPCFail}\})\circledS\overline{b} = \overline{s}$$

$$\#\overline{b} = \mathrm{NbRp}(s, r)$$

$$\forall\, k \in \mathsf{dom}(\overline{r}) :$$

$$\quad \mathsf{let}\ m = \mathrm{NbRp}(s{\downarrow_{\mathsf{tm}(r,k)}}, r) + 1$$

$$\quad \mathsf{in}\ s{\downarrow_{\mathsf{tm}(r,k)}} \frown \surd^{\sigma} \sqsubseteq s \Rightarrow \overline{b}[m] = \mathrm{RPCFail}$$

$$\forall\, k, l \in \mathsf{dom}(\overline{r}) : k < l \Rightarrow$$

$$\quad \mathsf{tm}(r, l) - \mathsf{tm}(r, k) > \sigma \vee \#\overline{s{\downarrow_{\mathsf{tm}(r,l)-1}}} - \#\overline{s{\downarrow_{\mathsf{tm}(r,k)}}} > 0$$

There are five main conjuncts. The first requires that any message received on a is forwarded along r, and that no other message is sent along r.

The second conjunct requires that if the RPCFail exceptions are ignored then the output along b is exactly the input received on s.

The third conjunct requires that the number of messages in $\overline{b}$ is equal to the number of messages in $\overline{s}$ plus the number of *time-outs* — in other words, the number of messages in r for which no reply is received on s within σ time units.

The fourth conjunct requires that if a message is sent along r at some time k and no reply is received on s within the next σ time units then the mth message output along b is RPCFail, where $m - 1$ is the number of messages (time-outs included) received on s until time $\mathsf{tm}(r, k)$.

The fifth conjunct requires that the component always waits for a reply to the previous call before it makes a new call. This means that the next call comes at least $\sigma + 1$ time units after the previous call or at least one time unit after the transmission of a reply to the previous call. The auxiliary function NbRp is defined below.

NbRp(s, r)

$$\mathrm{ReRpl}^{\underline{\omega}} \times \mathrm{ReCall}^{\underline{\infty}} \to \mathbb{N}$$

$$\#\overline{s} + \#\{m \in \mathsf{dom}(\overline{r}) \mid s{\downarrow_{\mathsf{tm}(r,m)}} \frown \surd^{\sigma} \sqsubseteq s\}$$

It yields the number of replies in s including the time-outs with respect to r.

The second main conjunct of the I/O-relation of SeqRPCClerk refers to the auxiliary function defined below.

RPCClerkDelayed(a, s, b, r)

$\text{ReCall}^{\underline{\infty}} \times \text{ReRpl}^{\underline{\infty}} \times \text{ReRpl}^{\underline{\infty}} \times \text{ReCall}^{\underline{\infty}} \to \mathbb{B}$

$\text{Delayed}(a, r)$

$\forall j \in \mathbb{N} : \#(\overline{b\downarrow_{j+1}}) \leq \text{NbRp}(s\downarrow_j, r)$

The first conjunct makes sure that the messages received on a are forwarded along r with a delay of at least one time unit. The second conjunct imposes a similar requirement with respect to s and b. It looks a bit different since it not only considers the ordinary messages received on s, but also the time-outs with respect to r.

7.2 The Implementation

The network pictured in Fig. 4 is specified as follows.

SeqRPCImpl — time_dependent

in $a \in \text{ReCall}^{\underline{\infty}}, d \in \text{PrRpl}^{\underline{\infty}}$

out $b \in \text{ReRpl}^{\underline{\infty}}, c \in \text{PrCall}^{\underline{\infty}}$

loc $r \in \text{ReCall}^{\underline{\infty}}, s \in \text{ReRpl}^{\underline{\infty}}$

$\text{SeqLossyRPC}(r, d, s, c) \otimes \text{SeqRPCClerk}(a, s, b, r)$

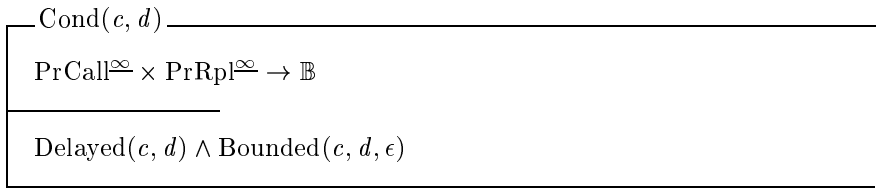
7.3 Verification

In order to prove that SeqRPCImpl implements SeqRPC we need a more general concept of refinement — namely what we refer to as conditional refinement. Let S and S' be time-dependent specifications of the same syntactic interface, and let B be a formula whose free variables are either input or output streams according to the declarations in the syntactic interface. We define S' to be a *conditional refinement* of S modulo the condition B if

$$B \wedge R_{S'} \Rightarrow R_S$$

under the assumption that any free variable is typed in accordance with its declaration in the syntactic interface.

We now prove that SeqRPCImpl is a conditional refinement of SeqRPC modulo the *condition* $\text{Cond}(c, d)$ defined below.



Remember that for any specification S, we use R_S to denote its I/O-relation. If the I/O-relation has $\Rightarrow$ as its main operator, we use A_S to denote its antecedent (assumption) and C_S to denote its consequent (commitment). Although this is normally not pointed out explicitly, the validity of the many deductions below depends on the type definitions listed in Sect. 3.1, 4.1, and also the assumptions 1-7 listed in Sect. 3.1, 4.1, 5.1.

In the following we assume that

$$a \in \mathrm{ReCall}^{\underline{\infty}} \tag{96}$$

$$r \in \mathrm{ReCall}^{\underline{\infty}} \tag{97}$$

$$b \in \mathrm{ReRpl}^{\underline{\infty}} \tag{98}$$

$$s \in \mathrm{ReRpl}^{\underline{\infty}} \tag{99}$$

$$c \in \mathrm{PrCall}^{\underline{\infty}} \tag{100}$$

$$d \in \mathrm{PrRpl}^{\underline{\infty}} \tag{101}$$

We want to prove that

$$\mathrm{Cond}(c,d) \wedge R_{\mathrm{SeqLossyRPC}(r,d,s,c)} \wedge R_{\mathrm{SeqRPCClerk}(a,s,b,r)} \Rightarrow R_{\mathrm{SeqRPC}(a,d,b,c)} \tag{102}$$

Since

$$\mathrm{Cond}(c,d) \Rightarrow A_{\mathrm{SeqLossyRPC}(r,d,s,c)} \wedge A_{\mathrm{SeqRPC}(a,d,b,c)} \tag{103}$$

it follows that 102 holds if

$$\mathrm{Cond}(c,d) \wedge C_{\mathrm{SeqLossyRPC}(r,d,s,c)} \wedge R_{\mathrm{SeqRPCClerk}(a,s,b,r)} \Rightarrow C_{\mathrm{SeqRPC}(a,d,b,c)} \tag{104}$$

It remains to prove 104. Let a, r, b, s, c, d be such that

$$\mathrm{Cond}(c,d) \tag{105}$$

$$C_{\mathrm{SeqLossyRPC}(r,d,s,c)} \tag{106}$$

$$R_{\mathrm{SeqRPCClerk}(a,s,b,r)} \tag{107}$$

It must be shown that

$$C_{\mathrm{SeqRPC}(a,d,b,c)} \tag{108}$$

106 implies there are $q \in \text{OkFail}^\infty$, $s' \in \text{ReRpl}^{\overline{\infty}}$ such that

$$s = ((\text{ReRpl} \cup \{\surd\}) \setminus \{\text{RPCFail}\})Ⓢs' \tag{109}$$

$$\text{RPCBehavior}(\overline{r}, \overline{d}, \overline{s'}, \overline{c}, q) \tag{110}$$

$$\text{RPCDelayed}(r, d, s', c, q) \tag{111}$$

$$\text{RPCBounded}(r, d, s', c, q) \tag{112}$$

107 implies

$$\text{RPCClerkBehavior}(a, s, b, r) \tag{113}$$

$$\text{RPCClerkDelayed}(a, s, b, r) \tag{114}$$

108 follows if we can show that

$$\text{RPCBehavior}(\overline{a}, \overline{d}, \overline{b}, \overline{c}, q) \tag{115}$$

$$\text{RPCDelayed}(a, d, b, c, q) \tag{116}$$

We first prove the following two lemmas

$$\text{Delayed}(s', b) \tag{117}$$

$$\overline{s'} = \overline{b} \tag{118}$$

Then we use 117, 118 to deduce 115, 116.

Proof of 117: 111 implies

$$\text{Delayed}(r, s') \tag{119}$$

2, 105, 112 imply

$$\text{Bounded}(r, s', \sigma) \tag{120}$$

119, 120 imply

$$\forall j \in \mathbb{N} : \#(\overline{s'\downarrow_{j+1}}) \leq \#(\overline{r\downarrow_j}) \leq \#(\overline{s'\downarrow_{j+\sigma}}) \tag{121}$$

113 implies

$$\forall k, l \in \mathsf{dom}(\overline{r}) : k < l \Rightarrow \tag{122}$$
$$\mathsf{tm}(r, l) - \mathsf{tm}(r, k) > \sigma \vee \#\overline{s\downarrow_{\mathsf{tm}(r,l)-1}} - \#\overline{s\downarrow_{\mathsf{tm}(r,k)}} > 0$$

109, 122 imply

$$\forall k, l \in \mathsf{dom}(\overline{r}) : k < l \Rightarrow \tag{123}$$
$$\mathsf{tm}(r, l) - \mathsf{tm}(r, k) > \sigma \vee \#\overline{s'\downarrow_{\mathsf{tm}(r,l)-1}} - \#\overline{s'\downarrow_{\mathsf{tm}(r,k)}} > 0$$

121, 123 imply

$$\forall k, l \in \mathsf{dom}(\overline{r}) : k < l \Rightarrow \#\overline{s'\downarrow_{\mathsf{tm}(r,l)-1}} - \#\overline{s'\downarrow_{\mathsf{tm}(r,k)}} > 0 \tag{124}$$

119, 120, 124 imply

$$\forall\, k, l \in \mathsf{dom}(\overline{r}) : k < l \Rightarrow \quad (125)$$
$$\mathsf{tm}(r,k) < \mathsf{tm}(s',k) < \mathsf{tm}(r,l) < \mathsf{tm}(s',l)$$

109, 120, 125 imply

$$\forall\, j \in \mathbb{N} : \mathrm{NbRp}(s{\downarrow_j}, r) \leq \#\overline{s'{\downarrow_j}} \quad (126)$$

114 implies

$$\forall\, j \in \mathbb{N} : \#(\overline{b{\downarrow_{j+1}}}) \leq \mathrm{NbRp}(s{\downarrow_j}, r) \quad (127)$$

126, 127 imply 117.

Proof of 118: 113 implies

$$(\mathrm{ReRpl} \setminus \{\mathrm{RPCFail}\})Ⓢ\overline{b} = \overline{s} \quad (128)$$
$$\#\overline{b} = \mathrm{NbRp}(s, r) \quad (129)$$
$$\forall\, k \in \mathsf{dom}(\overline{r}) : \quad (130)$$
$$\mathsf{let}\ m = \mathrm{NbRp}(s{\downarrow_{\mathsf{tm}(r,k)}}, r) + 1$$
$$\mathsf{in}\ s{\downarrow_{\mathsf{tm}(r,k)}} \frown \surd^{\sigma} \sqsubseteq s \Rightarrow \overline{b}[m] = \mathrm{RPCFail}$$

109, 128 imply

$$(\mathrm{ReRpl} \setminus \{\mathrm{RPCFail}\})Ⓢ\overline{b} = (\mathrm{ReRpl} \setminus \{\mathrm{RPCFail}\})Ⓢ\overline{s'} \quad (131)$$

109, 122, 125 imply

$$\forall\, j \in \mathsf{dom}(\overline{s'}) : \overline{s'}[j] = \mathrm{RPCFail} \Rightarrow s{\downarrow_{\mathsf{tm}(r,j)}} \frown \surd^{\sigma} \sqsubseteq s \quad (132)$$

109, 122, 125, 130, 132 imply

$$\forall\, j \in \mathsf{dom}(\overline{s'}) : \overline{s'}[j] = \mathrm{RPCFail} \Rightarrow \overline{b}[j] = \mathrm{RPCFail} \quad (133)$$

109, 129, 131, 133 imply 118.

Proof of 115: 113 implies

$$\overline{a} = \overline{r} \quad (134)$$

110, 118, 134 imply 115.

Proof of 116: 111 implies

$$\mathrm{Delayed}((\mathrm{VldReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(r, q), c) \quad (135)$$
$$\mathrm{Delayed}(r, s') \quad (136)$$
$$\mathrm{Delayed}(d, ((\mathrm{PrRpl} \setminus \{\mathrm{BadCall}\}) \times \{\mathrm{Ok}, \mathrm{Pfail}\})Ⓡ(s', q)) \quad (137)$$

113, 114 imply

$$\overline{a} = \overline{r} \tag{138}$$

$$\mathrm{Delayed}(a, r) \tag{139}$$

135, 138, 139 imply

$$\mathrm{Delayed}((\mathrm{VldReCall} \times \{\mathrm{Ok}, \mathrm{Pfail}\})®(a, q), c) \tag{140}$$

117, 136, 139 imply

$$\mathrm{Delayed}(a, b) \tag{141}$$

117, 118, 137 imply

$$\mathrm{Delayed}(d, ((\mathrm{PrRpl} \setminus \{\mathrm{BadCall}\}) \times \{\mathrm{Ok}, \mathrm{Pfail}\})®(b, q)) \tag{142}$$

140, 141, 142 imply 116.

8 Conclusions

We have employed a specification and refinement technique based on *streams* to solve the RPC-memory specification problem. We have emphasized the use of *oracles* to capture time-independent nondeterministic behavior. We find oracles intuitive. Moreover, they allow us to deal with nondeterminism in a structured way. The commitments of our specifications are all written in the form

$$\exists\ \mathit{Oracle\_Decls} : \mathit{Oracle\_Constraint} \wedge \mathit{Behavior\_Constraint}$$

Oracle_Decls is a list of oracle declarations. *Oracle_Constraint* is a formula imposing additional constraints on the oracles. *Oracle_Constraint* is typically used to impose fairness requirements or to make sure that the oracles satisfy certain compatibility conditions. *Behavior_Constraint* is formula which for each oracle interpretation such that *Oracle_Constraint* holds, determines the (time-independent) output history as a function of the input history. If the timing is ignored *Behavior_Constraint* can easily be replaced by a functional program viewing the oracles as additional input streams. Thus, with respect to the time-independent behavior, our specifications are quite close to what one might think of as "nondeterministic" functional programs.

We have carried out detailed proofs. They are all relatively straightforward. Our proofs are not formal. However, they seem easy to formalize. The main obstacle on the road to formalization is the definitions of the new oracles in Section 5.5. They must be restated in a constructive manner. This is not technically difficult, but nevertheless quite tedious.

The use of streams to model dataflow networks was first proposed in [Kah74]. Timed streams and relations on timed streams are also well-known from the literature [Par83], [Kok87]. The same holds for oracles [Kel78], [AL88] (called prophecy variables in the latter). The presented approach is based on [BS96] which can be seen as a complete rework/relational reformulation of [BDD$^+$93]. There are also some obvious links to Z [Spi88].

9 Acknowledgements

The author has benefited from discussions with Manfred Broy on this and related topics. A very constructive referee report by Leslie Lamport led to many improvements and simplifications. A second anonymous referee report also provided some useful remarks. Øystein Haugen read a more recent version of the paper. His detailed comments were very helpful.

References

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, Digital, SRC, Palo Alto, 1988.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17:507–533, 1995.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to Focus (revised version). Technical Report SFB 342/2/92 A, Technische Universität München, 1993.

[BL] M. Broy and L. Lamport. The rpc-memory specification problem. This volume.

[BS96] M. Broy and K. Stølen. Focus — a method for the development of interactive systems. Book manuscript, June 1996.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. Information Processing*, pages 471–475. North-Holland, 1974.

[Kel78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In *Proc. Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[Kok87] J. N. Kok. A fully abstract semantics for data flow nets. In *Proc. PARLE, Lecture Notes in Computer Science 259*, pages 351–368. Springer, 1987.

[Par83] D. Park. The "fairness" problem and nondeterministic computing networks. In *Proc. Foundations of Computer Science, Mathematical Centre Tracts 159*, pages 133–161. Mathematisch Centrum Amsterdam, 1983.

[Spi88] J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[Stø96a] K. Stølen. Assumption/commitment rules for data-flow networks — with an emphasis on completeness. In *Proc. ESOP, Lecture Notes in Computer Science 1058*, pages 356–372. Springer, 1996.

[Stø96b] K. Stølen. Refinement principles supporting the transition from asynchronous to synchronous communication. *Science of Computer Programming*, 26:255–272, 1996.

A Specifying the Hand-Shake Protocol

As pointed out in the introduction to this paper, we have not specified the hand-shake protocol on which [BL] is based. The protocol has not been left out because it cannot be expressed in our formalism. In fact, as shown below, we can

easily specify the hand-shake protocol. We have ignored the hand-shake protocol because our formalism is tailored towards asynchronous communication with *unbounded* buffering. Our approach is based on the idea that the use of unbounded buffering is helpful when a system is described and reasoned about during its design — helpful because it allows us to abstract from the synchronization protocols needed in system descriptions based on bounded buffering.

Of course, at some point in a system development boundedness constraints have to be introduced — after all, any real system is based on bounded resources. To show that our approach supports the introduction of hand-shake communication, we now use conditional refinement, as defined in Sect. 7.3, to refine McImpl into a specification SynMcImpl that differs from McImpl in only one respect: the hand-shake protocol is imposed. See [Stø96b] for a detailed discussion of refinement principles supporting the introduction of synchronization in a compositional manner. The new overall specification is given below.

SynMcImpl — time_dependent

in	$ip \in (\text{Call}^{\underline{\infty}})^n$
out	$op \in (\text{Rpl}^{\underline{\infty}})^n$
loc	$cp \in (\text{Call}^{\underline{\infty}})^n$, $dp \in (\text{Rpl}^{\underline{\infty}})^n$, $ap \in (\text{ReCall}^{\underline{\infty}})^n$, $bp \in (\text{ReRpl}^{\underline{\infty}})^n$

$$\text{SynRelMc}(cp, dp) \otimes \text{SynRPC}(ap, dp, bp, cp) \otimes \text{SynClerk}(ip, bp, op, ap)$$

To simplify the specifications of the three components we introduce an auxiliary function HandShaked.

HandShaked(a, b, j)

$$(\alpha^{\underline{\infty}})^n \times (\alpha^{\underline{\infty}})^n \times \mathbb{N}_\infty \to \mathbb{B}$$

$$\forall\, k \in [1..n], l \in [1..j] : \#(\overline{a_k{\downarrow}_l}) \leq \#(\overline{b_k{\downarrow}_{l-1}}) + 1$$

α is a type variable. HandShaked(a, b, ∞) holds if for any k, l the number of messages transmitted on the channel a_k until time l is maximum one greater than the number of messages transmitted on the channel b_k until time $l-1$. The component specification SynRelMc is adapted from RelMc as follows.

SynRelMc — time_dependent

in $cp \in (\mathrm{Call}^{\underline{\infty}})^n$

out $dp \in (\mathrm{Rpl}^{\underline{\infty}})^n$

$$\forall j \in \mathbb{N}_\infty : \mathrm{HandShaked}(cp, dp, j) \Rightarrow$$
$$\exists cp' \in (\mathrm{Call}^{\underline{\infty}})^n, dp' \in (\mathrm{Rpl}^{\underline{\infty}})^n :$$
$$R_{\mathrm{RelMc}(cp', dp')}$$
$$\forall k \in [1..n] : cp_k{\downarrow_j} \sqsubseteq cp'_k \wedge dp_k{\downarrow_{j+1}} \sqsubseteq dp'_k$$

The component characterized by SynRelMc is required to behave in accordance with RelMc at least one time unit longer than its environment behaves in accordance with HandShaked. This is sufficient to ensure hand-shake behavior, since RelMc requires that a reply is never output before at least one time unit after the call is transmitted. Note the relationship to the interpretation of assumption/commitment specifications in [AL95] (see also [Stø96a]).

In exactly the same way as the specifications RPC and Clerk are defined as the conjunctions of n sub-specifications, we define SynRPC and SynClerk as the conjunctions of n SeqSynRPC and SeqSynClerk specifications, respectively. We give only the specifications of the sequential components. The following auxiliary function is useful.

RplDelayed(a, b, j)

$$(\alpha^{\underline{\infty}})^n \times (\alpha^{\underline{\infty}})^n \times \mathbb{N}_\infty \to \mathbb{B}$$

$$\forall k \in [1..n] :$$
$$\forall l \in [1..j] : \#(\overline{b_k{\downarrow_l}}) \leq \#(\overline{a_k{\downarrow_{l-1}}})$$
$$j = \infty \Rightarrow \#\overline{b_k} = \#\overline{a_k}$$

RplDelayed is used to capture that for any message sent on a exactly one reply is received on b with a delay of at least one time unit. The sequential synchronous RPC component is specified as follows.

SeqSynRPC — time_dependent

in $a \in \text{ReCall}^{\underline{\infty}}, d \in \text{PrRpl}^{\underline{\infty}}$

out $b \in \text{ReRpl}^{\underline{\infty}}, c \in \text{PrCall}^{\underline{\infty}}$

$\forall j \in \mathbb{N}_\infty : \text{HandShaked}(a, b, j) \wedge \text{RplDelayed}(c, d, j) \Rightarrow$

$\exists\, a' \in \text{ReCall}^{\underline{\infty}}, d' \in \text{PrRpl}^{\underline{\infty}}, b' \in \text{ReRpl}^{\underline{\infty}}, c' \in \text{PrCall}^{\underline{\infty}} :$

$A_{\text{SeqRPC}(a', d', b', c')}$

$C_{\text{SeqRPC}(a', d', b', c')}$

$a{\downarrow_j} \sqsubseteq a' \wedge d{\downarrow_j} \sqsubseteq d' \wedge b{\downarrow_{j+1}} \sqsubseteq b' \wedge c{\downarrow_{j+1}} \sqsubseteq c'$

Remember that for any specification S, whose I/O-relation has $\Rightarrow$ as its main operator, we use A_S and C_S to denote its antecedent (assumption) and consequent (commitment), respectively. SeqSynRPC requires the specified component to behave in accordance with the commitment of SeqRPC at least one time unit longer than the environment of SeqSynRPC behaves in accordance with the antecedent (environment assumption) of SeqSynRPC. The sequential synchronous clerk component is specified below.

SeqSynClerk — time_dependent

in $i \in \text{Call}^{\underline{\infty}}, b \in \text{ReRpl}^{\underline{\infty}}$

out $o \in \text{ReRpl}^{\underline{\infty}}, a \in \text{ReCall}^{\underline{\infty}}$

$\forall j \in \mathbb{N}_\infty : \text{HandShaked}(i, o, j) \wedge \text{RplDelayed}(a, b, j) \Rightarrow$

$\exists\, i' \in \text{Call}^{\underline{\infty}}, b' \in \text{ReRpl}^{\underline{\infty}}, o' \in \text{ReRpl}^{\underline{\infty}}, a' \in \text{ReCall}^{\underline{\infty}} :$

$A_{\text{SeqClerk}(i', b', o', a')}$

$C_{\text{SeqClerk}(i', b', o', a')}$

$\text{Delayed}(\langle \text{RPCFail}, \surd \rangle \frown b', \langle \surd \rangle \frown a')$

$i{\downarrow_j} \sqsubseteq i' \wedge b{\downarrow_j} \sqsubseteq b' \wedge o{\downarrow_{j+1}} \sqsubseteq o' \wedge a{\downarrow_{j+1}} \sqsubseteq a'$

This specification has exactly the same structure as SeqSynRPC with the exception that an additional causality constraint is introduced to make sure that the clerk never sends a new call along a before it receives a reply on b to its previous call. Note that we could have replaced RPCFail by any other type correct message (remember that $\surd$ is not a message).

We now prove that SynMcImpl is a refinement of McImpl under the condition that a user never makes another call before it has received a reply to its previous

call. In the following we assume that

$$ip \in (\mathrm{Call}^{\underline{\infty}})^n \tag{143}$$

$$cp \in (\mathrm{Call}^{\underline{\infty}})^n \tag{144}$$

$$op \in (\mathrm{Rpl}^{\underline{\infty}})^n \tag{145}$$

$$dp \in (\mathrm{Rpl}^{\underline{\infty}})^n \tag{146}$$

$$ap \in (\mathrm{ReCall}^{\underline{\infty}})^n \tag{147}$$

$$bp \in (\mathrm{ReRpl}^{\underline{\infty}})^n \tag{148}$$

$$j \in [1..n] \tag{149}$$

It must be shown that

$$\mathrm{HandShaked}(ip, op, \infty) \wedge R_{\mathrm{SynMcImpl}(ip,op)} \Rightarrow R_{\mathrm{McImpl}(ip,op)} \tag{150}$$

We start by proving that the sub-specifications of SynMcImpl are conditional refinements of the corresponding sub-specifications of McImpl. More explicitly, conditional refinements in the following sense

$$\mathrm{HandShaked}(cp, dp, \infty) \wedge R_{\mathrm{SynRelMc}(cp,dp)} \Rightarrow R_{\mathrm{RelMc}(cp,dp)} \tag{151}$$

$$\mathrm{HandShaked}(ap_j, bp_j, \infty) \wedge R_{\mathrm{SeqSynRPC}(ap_j,dp_j,bp_j,cp_j)} \Rightarrow \tag{152}$$

$$R_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)}$$

$$\mathrm{HandShaked}(ip_j, op_j, \infty) \wedge R_{\mathrm{SeqSynClerk}(ip_j,bp_j,op_j,ap_j)} \Rightarrow \tag{153}$$

$$R_{\mathrm{SeqClerk}(ip_j,bp_j,op_j,ap_j)}$$

151 holds trivially. Since

$$A_{\mathrm{SeqRPC}(ap_j,dp_j,bp_j,cp_j)} \Rightarrow \mathrm{RplDelayed}(cp_j, dp_j, \infty) \tag{154}$$

$$A_{\mathrm{SeqClerk}(ip_j,bp_j,op_j,ap_j)} \Rightarrow \mathrm{RplDelayed}(ap_j, bp_j, \infty) \tag{155}$$

it is also clear that 152, 153 hold. Let ip, op be such that

$$\mathrm{HandShaked}(ip, op, \infty) \tag{156}$$

$$R_{\mathrm{SynMcImpl}(ip,op)} \tag{157}$$

150 follows if we can show that

$$R_{\mathrm{McImpl}(ip,op)} \tag{158}$$

157 implies there are cp, dp, ap, bp such that

$$R_{\mathrm{SynRelMc}(cp,dp)} \tag{159}$$

$$R_{\mathrm{SynRPC}(ap,dp,bp,cp)} \tag{160}$$

$$R_{\mathrm{SynClerk}(ip,bp,op,ap)} \tag{161}$$

151, 152, 153, 156, 159, 160, 161 imply that 158 follows if we can show that

$$\text{HandShaked}(ap, bp, \infty) \wedge \text{HandShaked}(cp, dp, \infty) \tag{162}$$

For any $k \in \mathbb{N}$, let

$$I_k \stackrel{\text{def}}{=} \text{HandShaked}(ap, bp, k) \wedge \text{HandShaked}(cp, dp, k) \tag{163}$$
$$\text{RplDelayed}(ap, bp, k) \wedge \text{RplDelayed}(cp, dp, k)$$

163 implies

$$I_0 \tag{164}$$

156, 159, 160, 161, 163 imply

$$\forall\, k \in \mathbb{N} : I_k \Rightarrow I_{k+1} \tag{165}$$

164, 165 and induction on k imply

$$\forall\, k \in \mathbb{N} : I_k \tag{166}$$

163, 166 imply 162. This ends our proof.

In Sect. 5.5 we have shown that the composite specification McImpl(ip, op) is a refinement of UnrelMc(ip, op). Above we have shown that SynMcImpl(ip, op) is a conditional refinement of McImpl(ip, op) modulo HandShaked(ip, op, ∞). By the definitions of refinement it follows that SynMcImpl(ip, op) is a conditional refinement of UnrelMc(ip, op) modulo HandShaked(ip, op, ∞).