Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Tackling the Dagstuhl'94 specification problem with I/O automata

J.M.T. Romijn

Computer Science/Department of Software Technology

CS-R9617 1996

Report CS-R9617
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Tackling the Dagstuhl'94 Specification Problem with I/O Automata

Judi Romijn
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
judi@cwi.nl

Abstract

An I/O automata solution to the problem posed by Broy & Lamport at the Dagstuhl Workshop on Reactive Systems is presented. The problem, which concerns components that communicate by means of a procedure interface, consists of an untimed and a timed part. In this paper, both parts are solved completely.

AMS Subject Classification (1991): 68Q05, 68Q22, 68Q25, 68Q60.
CR Subject Classification (1991): C.2.2, D.2.4, F.1.1, F.3.1.
Keywords & Phrases: Concurrency, protocol verification, I/O automata, fairness, liveness, real time.
Note: The results reported in this paper have been obtained as part of the research project "Specification, Testing and Verification of Software for Technical Applications", which is being carried out by the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps.

1. Introduction

An example of an distributed system specification problem was stated at the Workshop on Reactive Systems, held in Dagstuhl, Germany in September 1994. The problem concerned the specification of a memory component and a procedure interface component, and the implementation of both. The specification problem is stated in full in [7]. In the remainder of this paper, we assume that the reader is familiar with that document.

The workshop's main intention was to compare different formalisms by applying them to this example, in order to understand the similarities and differences of the various approaches, as well as their strengths and weaknesses. The problem has been solved completely in [2, 6, 8, 11, 13, 14, 21]. Other papers on this topic are [3, 4, 5, 10, 12, 22] which only solve the untimed part, and [19] which simplifies the problem to a situation with only one sender and one receiver.

This paper is the result of a successful attempt to model and verify the Dagstuhl problem with the I/O automata model [9, 16, 17, 20]. The next two sections dwell on the obstacles that were encountered during the birth of this paper, and on the merits of I/O automata.

1.1 Notes on the problem specification

Ambiguities The informal descriptions of the Memory component in Problem 1 and the RPC component in Problem 2 are slightly ambiguous. It is not clear whether these components may issue a failure when a bad call is received. In both cases we have chosen to allow this, because it yields a more general specification. For the Memory component this decision conforms with the implementation proposed in Problem 3.

Observable versus internal behaviour Problem 3 requires to prove that a composition of components implements the Memory component. The Memory component can perform at most one internal read action between call and return. The proposed implementation, however, can do this an arbitrary (but finite!) number of times. The proof for the implementation relation is simplified substantially if one assumes that the Memory component can perform an arbitrary number of internal read actions instead of at most one. The solution of Abadi, Lamport & Merz [2] uses such a more convenient Memory component, and apparently adopts the assumption that the two Memory components are observationally equivalent. We prove formally that this assumption is correct, which requires a backward simulation proof of about four pages.

In the solution of Hooman [11] the correctness of this assumption is also proved, with seemingly much less effort. This is due to a difference in view on executions. Hooman introduces safety restrictions on the set of all possible executions. In this manner, unwanted behaviour is avoided. His approach also allows executions with an infinite number of internal actions between two external actions. Our executions are built in an operational manner by concatenating states and transitions. Hence safety restrictions are posed only on single actions, and not on executions. Besides, since each execution contains at most a countable number of actions, there is at most a finite number of actions between any two actions. We feel that the operational view is more natural and closer to any true implementation of this problem specification.

Fairness and real time In Problem 5, a timed implementation is compared with an untimed specification. The untimed behaviour is restricted by fairness, whereas the timed behaviour is completely determined by timing constraints. To be able to compare these behaviours, we defined the *fair timed I/O automaton*. This ad hoc notion is explained in Section 7.2.

1.2 Notes on the I/O automata model

Benefits I/O automata provide a natural way to describe processes with an input/output behaviour. Most distributed systems can be specified in this way. The specifications are highly readable, and can be explained without too much trouble to most non-experts.

The Dagstuhl problem includes a lot of rather exotic data types. The I/O automata model can handle such extensive use of data.

In the untimed part of our solution, simulation relations provide the major part of proofs for implementation relations, the rest is taken care of by inclusion of fairness properties. All these are standard ingredients of verifications with I/O automata.

Real time aspects of specifications are also captured in I/O automata quite easily. When comparing timed specifications, simulation relations prove implementation relations in a straightforward way.

Imperfections When reasoning about an I/O automaton with over five state variables and over five locally controlled actions, proofs for safety properties involve an enormous amount of tedious detail, and are prone to typos and more serious errors. The amount of paper needed to get these proofs done in a semi-readable way is terrifying, whereas in general the properties being proved seem so trivial and intuitively correct. However, we are not aware of the existence of a similar formalism without this problem.

An inconvenient gap in current I/O automata theory is that it is not possible to impose restrictions on the behaviour of the environment. Especially when using timed I/O automata, one sometimes needs to assume that certain events will happen within certain time bounds. There is no formal framework yet for assumptions of this kind.

What we added to the classic model The Dagstuhl problem requires strong fairness restrictions on the behaviour of the proposed implementation of the Memory component in Problem 3, but the I/O automata model proposed by Lynch & Tuttle [15] only deals with weak fairness. Secondly, the problem holds a parameter whose cardinality is unknown, namely the number of calling processes for a Memory or RPC component. Well-known results for liveness with respect to fairness conditions deal with at most a countable number of fairness sets or actions, and cannot be applied to this problem.

We overcome both difficulties by using the *fair I/O automaton* [20]. This is a slight variant of the I/O automaton in [15], and a special case of the live I/O automaton in [9] provided that two conditions hold. These conditions require that each reachable state enables at most a countable number of fairness sets, and that input actions do not disturb the enabledness of these sets. In this paper, each specification is proved to be a live I/O automaton by checking these two conditions.

In the solution of Abadi, Lamport & Merz [2] fairness properties are used quite frequently. In a preliminary version of that paper, liveness was proved for each specification with fairness properties. However, the fine point on cardinality, which we mentioned above, was overlooked. In the last version of [2] the liveness proofs have been omitted.

1.3 Further remarks

The outline of this paper is as follows. Section 2 lists some preliminaries which are necessary for a good understanding of the specifications, as well as the proofs. Sections 3 to 7 solve parts 1 to 5 of the problem consecutively. Appendix A lists the basics of the I/O automata model.

Since endless listings of highly detailed proofs guarantee a boring paper instead of a higher degree of understanding, we have omitted unnecessary detailed proofs and replaced some by sketches. The full formal proofs can be obtained by e-mail request to the author.

Acknowledgements Frits Vaandrager put me on the Dagstuhl problem to 'get to know the field of protocol verification'. We both thought that it would take much less time and energy than it did. Yet I have learned so much about protocol verification in general and more specifically about I/O automata, that I have almost developed a taste for obstacles. While I was working on this paper, enlightening e-mail correspondence has taken place with Jozef Hooman, Leslie Lamport and Stephan Merz.

2. Preliminaries

2.1 Fair I/O automata

The set-up of specification and verifications is as follows. All untimed specifications use the *fair I/O automata* model from [20]. The basics of this model are listed in Appendix A. The model is a generalization from the classic model by Lynch & Tuttle [15], and, under two restrictions, a special case of the live I/O automaton model by Gawlick et al. [9].

The timed specifications use the *timed I/O automata* model in [17].

We prove an implementation relation between two fair I/O automata A and B by proving that $\mathit{fairtraces}(A) \subseteq \mathit{fairtraces}(B)$. To ease this proof, we mostly start out by proving inclusion on the ordinary and quiescent traces of A and B using refinements and simulations.

Since the only difference between the fair and classic I/O automata model lies in the fairness properties, all results in the latter that do not concern fairness carry over to the fair I/O automata model. This is used when proving ordinary and quiescent trace inclusion.

2.2 Design and presentation of the fair I/O automata

All fair I/O automata are designed as follows.

Each action is indexed with the process, for which this action is performed. Some of the state variables are also indexed with a process. The state space is roughly partitioned by the value of the *program counters*, the state variables pc_P. These variables keep track of what the automaton should be doing for process P. All automata initially wait for some action by the environment, and each pc_P has a value that expresses this waiting condition. As soon as input is received for process P, pc_P changes accordingly, and each next input for P is discarded (the state is not changed), if pc_P does not satisfy the waiting condition. For each internal action, the precondition requires pc_P to have a specific value in order to ensure that the right actions are taken at the right moment. After the input for some process P has been handled, pc_P is set to the waiting condition again.

To give the values of each program counter the right meaning, we assume that the interpretation of the domain of each program counter is free, in the sense that different constants symbols are mapped to different elements in its domain ("no confusion"), and each element in the domain is denoted by some constant symbol ("no junk").

In the presentation of fair I/O automata, we use the following conventions:

- We omit the precondition of an input action (since this equals true by definition).
- In the effect part of transition types we omit assignments of the form $x := x$.
- We write if c then $[z_1 := f_1, \ldots, z_k := f_k]$ as an abbreviation for

$$\begin{array}{lcl} z_1 & := & \text{if } c \text{ then } f_1 \text{ else } z_1 \\ & \vdots & \\ z_k & := & \text{if } c \text{ then } f_k \text{ else } z_k \end{array}$$

- We write $x \in \{A, B, C\}$ for $x{=}A \vee x{=}B \vee x{=}C$, etc.
- To improve readability we often use Lamport's list notation for conjunction or disjunction. Thus we write

$$\begin{array}{ll} \wedge & b_1 \\ \wedge & b_2 \\ & \vdots \\ \wedge & b_n \end{array}$$

for $b_1 \wedge b_2 \cdots \wedge b_n$.

3. Specifications and verifications for Problem 1

3.1 Problem 1(a): Specification of two Memory Components

In this section, we present the formal specification of the given components. First we give the fair I/O automaton for the Memory component *Memory*, then for the Reliable Memory component *RelMemory*.

3.1.1 Data types We start the specification with a description of the various data types that play a role. We assume a typed signature Σ_1 and a Σ_1-algebra $\mathcal{A}_1$ which consist of the following components:

- a type **Bool** of booleans with constant symbols true and false, and a standard repertoire of function symbols ($\wedge$, $\vee$, $\neg$, $\rightarrow$), all with the standard interpretation over the booleans. Also, we require, for all types **S** in Σ, an equality, inequality, and if-then-else function symbol, with the usual interpretation:

$$\begin{array}{rcl} .=. & : & \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ .\neq. & : & \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ \mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ . & : & \mathbf{Bool}\times\mathbf{S}\times\mathbf{S} \rightarrow \mathbf{S} \end{array}$$

  Note the (harmless) overloading of the constants and function symbols of type **Bool** with the propositional connectives used in formulas. We will frequently view boolean valued expressions as formulas, i.e., we use b as an abbreviation of b=true.

- a type **Process** of process identifiers. We frequently use the variable P ranging over **Procs** as a subscript.

- a type **MemLocs** of legal memory locations.

- a type **MemVals** of legal memory values, with constant symbol InitVal. None of the memory values is equal to BadArg.

- a type **Locs** of memory locations, such that **MemLocs** $\subseteq$ **Locs**, and a function memloc : **Locs** $\rightarrow$ **Bool**, telling us whether an element of **Locs** is also an element of **MemLocs**.

- a type **Vals** of memory values, such that **MemVals** $\subseteq$ **Vals**, and a function memval : **Vals** $\rightarrow$ **Bool**, telling us whether an element of **Vals** is also an element of **MemVals**.

- a type **Ack** of acknowledgement values, such that **Ack** = **MemVals** $\cup$ WriteOk.

- a type **Memory** of functions from **MemLocs** to **MemVals** We need two functions to actually access the memory: find : **MemLocs**×**Memory** → **MemVals** and change : **MemLocs**×**MemVals**×**Memory** → **Memory**. These operations are fully characterized by the axioms:
  $\mathsf{find}(l, m) = m(l)$
  $\mathsf{change}(l, v, m) = m'$ where $m'(l){=}v \wedge \forall l' : (l' \neq l \rightarrow m'(l){=}m(l))$
  (where l, l' are variables of type **MemLocs**, v is a variable of type **MemVals**, and m, m' are variables of type **Memory**)

- a type **Mpc** of program counter values of the Memory component, with constant symbols WC, R and W. The intended meaning of these constants will be explained further on in this section.

3.1.2 The Memory component We will now present the fair I/O automaton *Memory*, which models a Memory component. The state variable pc_P of *Memory*, gives the current value of the program counter of the Memory component for calling process P. Note that there are as many program counters as calling processes. Each of them may have one of the following values:

- WC: Wait for a $READ_P$ or $WRITE_P$ call,
- R: Reading memory,
- W: Writing to memory.

Initially, the program counter value is WC for every process P.

Every possible action of *Memory* is indexed with the process that issued the call leading to this action. Since the state variables are also indexed in this manner (except for *memory*!), we can determine in any situation what is going on for each process P.

$READ_P$ and $WRITE_P$ model an incoming read or write call from a process P. They do not change the state when *Memory* is still handling a previous call from the same process. In this case, we call the input action *discarded*. If *Memory* is ready for handling an incoming call, its state will be updated according to the parameter(s) of the call.

GET_P actions model an atomic read operation, PUT_P actions model an atomic write operation. Reading is allowed only once between call and return, writing is allowed for an arbitrary number of times.

A $MEM\_FAILURE_P$ action can occur in any 'busy' state.

BAD_ARG_P is the only action enabled if the parameters of the call from process P were not legal. $RETURN_P$ delivers the requested memory value or a general WriteOk acknowledgement, after $performed_P$ has been set to true by a GET_P or PUT_P action. The fact that PUT_P actions are in another weak fairness set than $RETURN_P$ and $MEM\_FAILURE_P$, ensures that writing will stop at some point.

The code for *Memory* is listed in figure 1.

Input: $READ_P$, $WRITE_P$
Output: $RETURN_P$, BAD_ARG_P, $MEM\_FAILURE_P$
Internal: GET_P, PUT_P
WFair: $\bigcup_P \{\{GET_P, PUT_P\}, \{BAD\_ARG_P, MEM\_FAILURE_P, RETURN_P\}\}$
SFair: $\emptyset$

State Variables:

pc_P:	**Mpc**	
loc_P:	**Locs**	
val_P:	**Vals**	
$memory$:	**Memory**	
$performed_P$:	**Bool**	
$legal\_args_P$:	**Bool**	

Initialization: $\bigwedge_P\ pc_P$=WC
$\bigwedge_l$ find$(l, memory)$=InitVal

$READ_P(l : \mathbf{Locs})$
Effect:
if pc_P=WC then $[loc_P := l$
$performed_P :=$ false
$legal\_args_P :=$ memloc(l)
$pc_P :=$ R$]$

$WRITE_P(l : \mathbf{Locs}, v : \mathbf{Vals})$
Effect:
if pc_P=WC then $[loc_P := l$
$val_P := v$
$performed_P :=$ false
$legal\_args_P :=$ memloc$(l) \wedge$ memval(v)
$pc_P :=$ W$]$

GET_P
Precondition:
pc_P=R $\wedge\ legal\_args_P \wedge \neg performed_P$
Effect:
$val_P :=$ find$(loc_P, memory)$
$performed_P :=$ true

PUT_P
Precondition:
pc_P=W $\wedge\ legal\_args_P$
Effect:
$memory :=$ change$(loc_P, val_P, memory)$
$performed_P :=$ true

$RETURN_P(a : \mathbf{Ack})$
Precondition:
$pc_P \in \{$R, W$\} \wedge performed_P \wedge a$=if pc_P=R then val_P else WriteOk
Effect:
$pc_P :=$ WC

BAD_ARG_P
Precondition:
$pc_P \in \{$R, W$\} \wedge \neg legal_args_P$
Effect:
$pc_P :=$ WC

$MEM\_FAILURE_P$
Precondition:
$pc_P \in \{$R, W$\}$
Effect:
$pc_P :=$ WC

Figure 1: Fair I/O automaton *Memory*

Memory is live We will now show that fair I/O automaton *Memory* is a live I/O automaton in the sense of [9]. To do this, we have to check that *Memory* satisfies two conditions. After this, Theorem 1 from [20] applies immediately.

The next lemma checks a restriction of one of the two conditions.

Lemma 3.1 *Each reachable state in Memory enables at most finitely many locally controlled actions.*

Proof The initial states enable only input actions.

Suppose state s enables n locally controlled actions. It is trivial to see that for each transition $s \xrightarrow{a} s'$, s' enables at most $n+2$ locally controlled actions. ⊠

Proposition 3.2 *live*(*Memory*) *is a live I/O automaton.*

Proof We can apply Theorem 1 in [20] if we can show that (1) each reachable state of *Memory* enables at most countably many weak and strong fairness sets, and (2) each set in *sfair*(*Memory*) is input resistant.

Condition (1) is satisfied by Lemma 3.1, since each locally controlled action is in exactly one weak fairness set. Condition (2) is trivially satisfied, since there are no strong fairness sets. ⊠

3.1.3 The Reliable Memory component We will now present the fair I/O automaton *RelMemory*, which models a *Reliable* Memory component. This component behaves exactly like the Memory component, except for $MEM\_FAILURE_P$ actions, which cannot occur.

Since the code for *RelMemory* can be obtained from the code for *Memory* by omitting the $MEM\_FAILURE_P$ action, *wfair*(*RelMemory*) becomes

$\bigcup_P \{\{GET_P, PUT_P\}, \{BAD\_ARG_P, RETURN_P\}\}$

RelMemory is live Knowing that *Memory* is a live I/O automaton, it is easy to prove that *RelMemory* is also a live I/O automaton.

Proposition 3.3 *live*(*RelMemory*) *is a live I/O automaton.*

Proof The proof is almost identical to the proof of Theorem 3.2, since the only difference between *Memory* and *RelMemory* is the absence of $MEM\_FAILURE_P$ actions. ⊠

3.2 Problem 1(b): RelMemory implements Memory

We will show that *fairtraces*(*RelMemory*) $\subseteq$ *fairtraces*(*Memory*), using the properties safety and deadlock freeness.

3.2.1 Safety Since *RelMemory* and *Memory* are so very much alike, a weak refinement appears the most natural construction for proving safety.

Theorem 3.4 *The function* REF, *which is the identity function on state variables with the same name, is a weak refinement from RelMemory to Memory, with respect to the reachable states in both RelMemory and Memory.*

Proof The requirements in [18] are trivially fulfilled, since REF is the identity function, and the actions in *RelMemory* form a subset of those in *Memory*. ⊠

Corollary 3.5 *RelMemory is safe with respect to Memory.*

Proof Directly from Theorem 3.4 and [18]'s Theorem 6.2. ⊠

3.2.2 Deadlock freeness

Theorem 3.6 *For each reachable and quiescent state s of RelMemory,* REF(s) *is a quiescent state of Memory.*

Proof Suppose s is a quiescent state of *RelMemory*. Observing the preconditions of *RelMemory*, we see that $s \models \bigwedge_P RelMemory.pc_P = \mathsf{WC}$.
Clearly, REF(s) $\models \bigwedge_P Memory.pc_P = \mathsf{WC}$, hence REF($s$) is quiescent. ⊠

Corollary 3.7 *RelMemory is deadlock free with respect to Memory.*

Proof By Theorems 3.4 and 3.6 we can, for each quiescent execution of *RelMemory*, construct a corresponding quiescent execution of *Memory* with the same trace. ⊠

3.2.3 Implementation

Theorem 3.8 *RelMemory implements Memory.*

Proof

Assume that $\beta \in fairtraces(RelMemory)$. We must prove $\beta \in fairtraces(Memory)$.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be a fair execution of *RelMemory* with trace β.

If α is finite then α is quiescent and it follows by Corollary 3.7 that *Memory* has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in fairtraces(Memory)$. So we may assume without loss of generality that α is infinite.

Using the fact that REF is a weak refinement (Theorem 3.4) we can easily construct an execution $\alpha' = t_0 a_1 t_1 a_2 t_2 \cdots$ of *Memory* with trace β if we let each t_i = REF(s_i). It remains to prove that α' is fair.

The only difficulty is caused by an infinite suffix of α', in which $MEM\_FAILURE_P$ is enabled continuously, and no action from $\{RETURN_P, BAD\_ARG_P, MEM\_FAILURE_P\}$ is performed. In this case, α must contain an infinite suffix in which PUT_P occurs infinitely many times and is enabled continuously continuously. Since α is weakly fair, this is impossible.

The interpretation of all the other actions are equal in both automata, even with respect to to the weak fairness sets, so the weak fairness requirements for α' are satisfied by the weak fairness requirements for α.

Since *Memory* has no strong fairness sets, the above shows that α' is fair. ⊠

3.3 Problem 1(c): Nothing but $MEM\_FAILURE_P$ actions?

We can construct a very trivial automaton that implements *Memory*, and does nothing but raise $MEM\_FAILURE_P$ actions. It can have the same state variables as *Memory*, but only actions $READ_P$, $WRITE_P$ and $MEM\_FAILURE_P$. A weak refinement like REF will provide us safety and deadlock freeness results. Such a refinement is even enough to show that this

automaton implements *Memory*, since each fair execution in this automaton can be imitated by a fair execution in *Memory*, using the refinement.

Is it reasonable that such an implementation is possible? Since the specification of the Memory component is presented as a black box that does not remember success nor failure, it is reasonable to think of it as a dice, harbouring the same chances at success with every throw. So while one can expect such a Memory component to yield the right return at some time in an infinite sequence of trials, the possibility of infinitely many failures exists and must therefore be included in the specification we have presented here.

4. Specifications and verifications for Problem 2

4.1 Problem 2: Specification of the RPC component

4.1.1 Data types We assume a typed signature Σ_2 and a Σ_2-algebra $\mathcal{A}_2$ which consist of the following components:

- the type **Bool** as defined in Section 3.1.1
- a type **Nat** of natural numbers
- a type **Procs** of procedure names
- a type **Names**, such that **Procs** $\subseteq$ **Names**, and a function legal_proc : **Names** $\rightarrow$ **Bool**, telling us whether a given name is a legal procedure name (that is, an element of **Procs**), and a function arg_num : **Names** $\rightarrow$ **Nat**, giving the expected number of arguments for each name.
- a type **Args** of function arguments, and a function num : **Args** $\rightarrow$ **Nat**, giving the number of actual arguments.
- a type **ReturnVal** of possible return values. All exceptions raised by remote procedure calls are expected to be included in this type.
- a type **Rpc** of program counter values of the RPC component, with constant symbols WC, IC, WR and IR.

4.1.2 Specification We will now present the fair I/O automaton *RPC*, which models an RPC component. RPC stands for Remote Procedure Call. *RPC*'s program counters may have one of the following values:

- WC: Wait for remote calls from the sender
- IC: Issue a call to the receiver or an exceptional return to the sender
- WR: Wait for a return from the receiver
- IR: Issue a return (possibly exceptional) to the sender

Initially, the program counter value is WC for every process P.

As in the solution to Problem 1, every possible action of *RPC* is indexed with the process that issued the call leading to this action.

Input: REM_CALL_P, I_RETURN_P
Output: $I\_CALL_P, REM\_RETURN_P, BAD\_CALL_P, RPC\_FAILURE_P$
WFair: $\bigcup_P \{\{I\_CALL_P, REM\_RETURN_P, BAD\_CALL_P, RPC\_FAILURE_P\}\}$
SFair: $\emptyset$

State Variables: pc_P: **Rpc**; $proc_P$: **Names**; $args_P$: **Args**; $legal\_call_P$: **Bool**; $return_P$: **ReturnVal**

Initialization: $\bigwedge_P\ pc_P$=WC

$REM\_CALL_P(p : \textbf{Names}, a : \textbf{Args})$
Effect:
if pc_P=WC then [$proc_P := p$
$args_P := a$
$legal\_call_P :=$ legal_proc(p) $\wedge$ (num(a)=arg_num(p))
$pc_P :=$ IC]

$I\_CALL_P(p : \textbf{Names}, a : \textbf{Args})$
Precondition:
pc_P=IC $\wedge$ $legal\_call_P$ $\wedge$ p=$proc_P$ $\wedge$ a=$args_P$
Effect:
$pc_P :=$ WR

$I\_RETURN_P(r : \textbf{ReturnVal})$
Effect:
if pc_P=WR then [$pc_P :=$ IR
$return_P := r$]

$REM\_RETURN_P(r : \textbf{ReturnVal})$
Precondition:
pc_P=IR $\wedge$ r=$return_P$
Effect:
$pc_P :=$ WC

BAD_CALL_P
Precondition:
pc_P=IC $\wedge$ $\neg legal\_call_P$
Effect:
$pc_P :=$ WC

$RPC\_FAILURE_P$
Precondition:
$pc_P \in$ {IC, IR}
Effect:
$pc_P :=$ WC

Figure 2: Fair I/O automaton RPC

The code for *RPC* is listed in figure 2.

We will now show that fair I/O automaton *RPC* is a live I/O automaton, as before. The next lemma checks a restriction of one of the two necessary conditions.

Lemma 4.1 *Each reachable state in RPC enables at most finitely many locally controlled actions.*

Proof The initial states enable only input actions. Suppose state s enables n locally controlled actions. It is trivial to see that for each transition $s \xrightarrow{a} s'$, s' enables at most $n + 2$ locally controlled actions. ⊠

Proposition 4.2 *live(RPC) is a live I/O automaton.*

Proof As before, we apply Theorem 1 in [20] after showing that (1) each reachable state of *RPC* enables at most countably many weak and strong fairness sets, and (2) each set in *sfair*(*RPC*) is input resistant.

Condition (1) is satisfied by Lemma 4.1, since each locally controlled action is in exactly one weak fairness set. Condition (2) is trivially satisfied, since there are no strong fairness sets. ⊠

5. Specifications and verifications for Problem 3

5.1 Problem 3: Specification of the composition

5.1.1 Data types We start the specification with a description of the various data types that play a role. We assume a typed signature Σ_3 and a Σ_3-algebra $\mathcal{A}_3$ which imports $\mathcal{A}_1$ (section 3.1) and $\mathcal{A}_2$ (section 4.1) in such a way that:

- Read and Write are different constants of type **Procs** (and therefore also of type **Names**)
- arg_num(Read) = 1, and arg_num(Write) = 2
- the domain of **ReturnVal** is equal to the domain of **Ack**, plus an extra constant BadArg
- for each l of type **Locs** and v of type **Vals**, (l) and (l, v) are elements of type **Args**, num$((l))$ = 1 and num$((l, v))$ = 2.

5.1.2 A front end for the RPC component We need another component to make the RPC component retry a call to the reliable memory component. This component is a clerk, which can translate incoming calls to *RPC*'s format, and reissue such a call if *RPC* should fail. Therefore we present the fair I/O automaton *ClerkRPC*, which models a front end to the RPC component *RPC*. The program counters of *ClerkRPC* are of type **Rpc**, and therefore have the same possibilities as the program counters of *RPC*. Initially, the program counter value is WC for every process P.

The code for *ClerkRPC* is listed in figure 3.

We will now show that fair I/O automaton *ClerkRPC* is a live I/O automaton, as before. The next lemma checks a restriction of one of the two necessary conditions.

Lemma 5.1 *Each reachable state in ClerkRPC enables at most finitely many locally controlled actions.*

Input: $READ_P, WRITE_P, REM\_RETURN_P, BAD\_CALL_P, RPC\_FAILURE_P$
Output: $REM\_CALL_P, RETURN_P, BAD\_ARG_P, MEM\_FAILURE_P$
WFair: $\bigcup_P\{\{REM\_CALL_P, RETURN_P, BAD\_ARG_P, MEM\_FAILURE_P\}\}$
SFair: $\bigcup_P\{\{MEM\_FAILURE_P\}\}$
State Variables: pc_P: **Rpc** **Initialization:** $\bigwedge_P\ pc_P$=WC
$proc_P$: **Names**
loc_P: **Locs**
val_P: **Vals**
$mf\_allowed_P$: **Bool**
$return_P$: **ReturnVal**

$READ_P(l : \textbf{Locs})$
Effect:
if pc_P=WC then [$proc_P$:= Read
$loc_P := l$
$mf\_allowed_P$:= false
pc_P := IC]

$WRITE_P(l : \textbf{Locs}, v : \textbf{Vals})$
Effect:
if pc_P=WC then [$proc_P$:= Write
$loc_P := l$
$val_P := v$
$mf\_allowed_P$:= false
pc_P := IC]

$REM\_CALL_P(p : \textbf{Names}, a : \textbf{Args})$
Precondition:
pc_P=IC $\wedge$ p=$proc_P$ $\wedge$ a=if $proc_P$=Read then (loc_P) else (loc_P, val_P)
Effect:
pc_P := WR

$REM\_RETURN_P(r : \textbf{ReturnVal})$
Effect:
if pc_P=WR then [$return_P := r$
pc_P := IR]

BAD_CALL_P
Effect:
if pc_P=WR then [$return_P$:= BadArg
pc_P := IR]

$RPC\_FAILURE_P$
Effect:
if pc_P=WR then [$mf\_allowed_P$:= true
pc_P := IC]

$MEM\_FAILURE_P$
Precondition:
pc_P=IC $\wedge$ $mf\_allowed_P$
Effect:
pc_P := WC

$RETURN_P(r : \textbf{ReturnVal})$
Precondition:
pc_P=IR $\wedge$ $return_P \neq$ BadArg $\wedge$ r=$return_P$
Effect:
pc_P := WC

BAD_ARG_P
Precondition:
pc_P=IR $\wedge$ $return_P$=BadArg
Effect:
pc_P := WC

Figure 3: Fair I/O automaton *ClerkRPC*

Proof The initial states enable only input actions. Suppose state s enables n locally controlled actions. It is trivial to see that for each transition $s \xrightarrow{a} s'$, s' enables at most $n+2$ locally controlled actions. ⊠

Proposition 5.2 *live(ClerkRPC) is a live I/O automaton.*

Proof As before, we apply Theorem 1 in [20] after showing that (1) each reachable state of *RPC* enables at most countably many weak and strong fairness sets, and (2) each set in *sfair(ClerkRPC)* is input resistant.

Condition (1) is satisfied by Lemma 5.1, since each locally controlled action is in exactly one weak fairness set.

Condition (2) relies upon the input resistance of action *MEM_FAILURE*. Suppose that $MEM\_FAILURE_P$ is enabled in the reachable state s. Clearly, $s \models ClerkRPC.pc_P{=}\mathsf{IC}$. If an input action a for P occurs in s, by definition of *ClerkRPC* the transition $s \xrightarrow{a} s$ is taken, and $MEM\_FAILURE_P$ is still enabled. If an input action a for another P' occurs in s, the transition taken does not affect $ClerkRPC.pc_P$. Hence $MEM\_FAILURE_P$ is input resistant and the second condition is satisfied. ⊠

5.1.3 Renaming component RelMemory

Defining the front end *ClerkRPC* is not enough to establish the intended implementation. We also need to rename *RelMemory*, to avoid name clashing, and to get the proper synchronization. So we define a new fair I/O automaton $RRMemory \triangleq \mathsf{rename}(RelMemory)$, where for every P:

$$
\begin{array}{lll}
\mathsf{rename}(READ_P(l)) & = I\_CALL_P(\mathsf{Read}, (l)) & \\
\mathsf{rename}(WRITE_P(l,v)) & = I\_CALL_P(\mathsf{Write}, (l,v)) & \\
\mathsf{rename}(RETURN_P(a)) & = I\_RETURN_P(a) & \\
\mathsf{rename}(BAD\_ARG_P) & = I\_RETURN_P(\mathsf{BadArg}) & \\
\mathsf{rename}(x) & = x & \text{otherwise}
\end{array}
$$

(l is a variable of type **Locs**, v is a variable of type **Vals**, a is a variable of type **Ack**, and x is a action variable)

The code for *RRMemory* is listed in figure 4.

It is easily shown that *RRMemory* is a live I/O automaton.

Proposition 5.3 *live(RRMemory) is a live I/O automaton.*

Proof Trivially, $live(RRMemory) = rename(live(RelMemory))$. Combining this with Theorem 3.3 and [9]'s Proposition 3.23, we obtain that *live(RRMemory)* is a live I/O automaton. ⊠

5.1.4 The implementation *MemoryImp* is defined as the parallel composition of I/O automata *ClerkRPC*, *RPC* and *RRMemory*, with all communication between those components hidden:

$$MemoryImp \triangleq \mathsf{HIDE}\ I\ \mathsf{IN}\ (ClerkRPC \| RPC \| RRMemory)$$

Input: I_CALL_P
Output: I_RETURN_P
Internal: GET_P, PUT_P
WFair: $\bigcup_P \{\{GET_P, PUT_P\}, \{I\_RETURN_P\}\}$
SFair: $\emptyset$

State Variables:
pc_P: **Mpc**
loc_P: **Locs**
val_P: **Vals**
$memory$: **Memory**
$performed_P$: **Bool**
$legal\_args_P$: **Bool**

Initialization: $\bigwedge_P\ pc_P$=WC
$\bigwedge_l$ find$(l, memory)$=InitVal

I_CALL_P(Read, $(l$: **Locs**$))$
Effect:
if pc_P=WC then $[loc_P := l$
$performed_P :=$ false
$legal\_args_P :=$ memloc(l)
$pc_P :=$ R]

I_CALL_P(Write, $(l$: **Locs**, v : **Vals**$))$
Effect:
if pc_P=WC then $[loc_P := l$
$val_P := v$
$performed_P :=$ false
$legal\_args_P :=$ memloc$(l) \wedge$ memval(v)
$pc_P :=$ W]

GET_P
Precondition:
pc_P=R $\wedge\ legal\_args_P \wedge \neg performed_P$
Effect:
$val_P :=$ find$(loc_P, memory)$
$performed_P :=$ true

PUT_P
Precondition:
pc_P=W $\wedge\ legal\_args_P$
Effect:
$memory :=$ change$(loc_P, val_P, memory)$
$performed_P :=$ true

$I\_RETURN_P(a$: **Ack**$)$
Precondition:
$pc_P \in \{$R, W$\} \wedge performed_P \wedge a$=if pc_P=R then val_P else WriteOk
Effect:
$pc_P :=$ WC

I_RETURN_P(BadArg)
Precondition:
$pc_P \in \{$R, W$\} \wedge \neg legal_args_P$
Effect:
$pc_P :=$ WC

Figure 4: Fair I/O automaton *RRMemory*

where $I \triangleq \bigcup_P \{REM\_CALL_P(p,a), REM\_RETURN_P(r), BAD\_CALL_P, RPC\_FAILURE_P,$
$I\_CALL_P(p,a), I\_RETURN_P(r), GET_P, PUT_P$
$\mid p$ in domain **Names**, a in domain **Args**, r in domain **ReturnVal**$\}$.

Proposition 5.4 *live(MemoryImp) is a live I/O automaton.*

Proof Using Propositions 4.2, 5.2, 5.3, we can apply [9]'s Proposition 3.28 and obtain that $live(ClerkRPC) \| live(RPC) \| live(RRMemory)$ is a live I/O automaton. By applying [20]'s Theorem 2 twice, we obtain that $live(ClerkRPC \| RPC \| RRMemory)$ is a live I/O automaton. Now [9]'s Proposition 3.22 shows us that HIDE I IN $live(ClerkRPC \| RPC \| RRMemory)$ is a live I/O automaton, and this automaton is trivially equal to $live(MemoryImp)$. ⊠

5.2 Set-up for the verification

A direct proof of trace inclusion between *MemoryImp* and *Memory* is not very straightforward. This stems from the fact that *Memory* can only read its memory once for every read call. However, by *MemoryImp*'s fail retry-mechanism, it is able to read multiple times for one read call.

An intermediate automaton To show trace inclusion, we are apparently forced to use a forward backward simulation. However, since this is rather complicated, and [18]'s Theorem 4.1 states that we can just as well look for an intermediate automaton, we will keep things clear by constructing an intermediate automaton, which we allow to read its memory multiple times for one read call. This intermediate automaton will be called *Memory**, the $*$ indicating the possibility of multiple reads instead of one. The code for *Memory** is obtained from *Memory* as follows. The precondition for GET_P is weakened, and a new state variable $hist_P$ is added, in which the value of val_P is stored each time a return is issued. Figure 5 lists the code for fair I/O automaton *Memory**. Boxes highlight the places where the code for *Memory** differs from *Memory*.

A forward simulation establishes trace inclusion between *MemoryImp* and *Memory**; a backward simulation does the same for *Memory** and *Memory*. The new state variable $Memory^*.hist_P$ substantially simplifies the backward simulation and also makes it image-finite.

Fair I/O automaton *Memory** is now shown to be a live I/O automaton, as before. The next lemma checks a restriction of one of the two necessary conditions.

Lemma 5.5 *Each reachable state in Memory* enables at most finitely many locally controlled actions.*

Proof The initial states enable only input actions. Suppose state s enables n locally controlled actions. It is trivial to see that for each transition $s \xrightarrow{a} s'$, s' enables at most $n+2$ locally controlled actions. ⊠

Proposition 5.6 *live(Memory*) is a live I/O automaton.*

Proof As before, we apply Theorem 1 in [20] after showing that (1) each reachable state of *Memory** enables at most countably many weak and strong fairness sets, and (2) each set in $sfair(Memory^*)$ is input resistant.

Input: $READ_P$, $WRITE_P$
Output: $RETURN_P$, BAD_ARG_P, $MEM\_FAILURE_P$
Internal: GET_P, PUT_P
WFair: $\bigcup_P \{\{GET_P, PUT_P\}, \{BAD\_ARG_P, MEM\_FAILURE_P, RETURN_P\}\}$
SFair: $\emptyset$

State Variables:

pc_P:	**Mpc**
loc_P:	**Locs**
val_P:	**Vals**
$memory$:	**Memory**
$performed_P$:	**Bool**
$legal\_args_P$:	**Bool**
$\boxed{hist_P}$:	**Vals**

Initialization: $\bigwedge_P\ pc_P$=WC
$\bigwedge_l$ find$(l, memory)$=InitVal
$\boxed{\bigwedge_P\ hist_P = val_P}$

$READ_P(l : \textbf{Locs})$
Effect:
if pc_P=WC then $[loc_P := l$
$performed_P :=$ false
$legal\_args_P :=$ memloc(l)
$pc_P :=$ R$]$

$WRITE_P(l : \textbf{Locs}, v : \textbf{Vals})$
Effect:
if pc_P=WC then $[loc_P := l$
$val_P := v$
$performed_P :=$ false
$legal\_args_P :=$ memloc$(l) \wedge$ memval(v)
$pc_P :=$ W$]$

GET_P
Precondition:
$\boxed{pc_P = \text{R} \wedge legal\_args_P}$
Effect:
$val_P :=$ find$(loc_P, memory)$
$performed_P :=$ true

PUT_P
Precondition:
$pc_P = \text{W} \wedge legal\_args_P$
Effect:
$memory :=$ change$(loc_P, val_P, memory)$
$performed_P :=$ true

$RETURN_P(a : \textbf{Ack})$
Precondition:
$pc_P \in \{\text{R}, \text{W}\} \wedge performed_P \wedge a$=if pc_P=R then val_P else WriteOk
Effect:
$pc_P :=$ WC
$\boxed{hist_P := val_P}$

BAD_ARG_P
Precondition:
$pc_P \in \{\text{R}, \text{W}\} \wedge \neg legal\_args_P$
Effect:
$pc_P :=$ WC
$\boxed{hist_P := val_P}$

$MEM\_FAILURE_P$
Precondition:
$pc_P \in \{\text{R}, \text{W}\}$
Effect:
$pc_P :=$ WC
$\boxed{hist_P := val_P}$

Figure 5: Fair I/O automaton *Memory**

Condition (1) is satisfied by Lemma 5.5, since each locally controlled action is in exactly one weak fairness set.

Condition (2) is trivially satisfied, since there are no strong fairness sets. ⊠

5.3 Problem 3: MemoryImp implements Memory

Section 5.3.1 shows that *Memory** implements *Memory*, Section 5.3.2 shows that *MemoryImp* implements *Memory**. Both results are reached via safety and deadlock freeness. Transitivity of the implementation relation yields the desired result in Section 5.3.3.

5.3.1 Memory implements Memory* We need an invariant to show that between the previous output action and the next internal action, *Memory**'s history variable $hist_P$ is up to date with respect to val_P for each P.

Lemma 5.7 *The following property* Inv1 *is an invariant of Memory*.*

$$\bigwedge_P \ (pc_P \in \{\mathsf{WC}, \mathsf{R}\} \wedge \neg performed_P) \rightarrow val_P{=}hist_P$$

The next invariant expresses that *Memory** will not read or write if it has received illegal arguments.

Lemma 5.8 *The following property* Inv2 *is an invariant of Memory*.*

$$\bigwedge_P \ pc_P{\neq}\mathsf{WC} \rightarrow (\neg legal\_args_P \rightarrow \neg performed_P)$$

A weak backward simulation enables us to construct the behaviour of *Memory*, given the behaviour of *Memory**. We can start in the last state of such a sequence, and work our way back to the beginning.

Theorem 5.9 *The relation* BACK *defined by the following formula is a weak backward simulation from Memory* to Memory, with respect to the reachable states in Memory*.*

$$\begin{array}{lll}
\bigwedge_P & Memory.pc_P & = \ Memory^*.pc_P \\
\bigwedge_P & Memory.loc_P & = \ Memory^*.loc_P \\
\bigwedge_P & Memory.val_P & = \ \mathsf{if} \quad Memory.pc_P{=}\mathsf{R} \wedge \neg Memory.performed_P \\
 & & \ \ \mathsf{then} \ \ Memory^*.hist_P \\
 & & \ \ \mathsf{else} \ \ Memory^*.val_P \\
\bigwedge_P & Memory.legal_args_P & = \ Memory^*.legal_args_P \\
\wedge & Memory.memory & = \ Memory^*.memory \\
\bigwedge_P & \neg Memory^*.performed_P & \rightarrow \neg Memory.performed_P \\
\bigwedge_P & Memory^*.pc_P{\in}\{\mathsf{WC}, \mathsf{W}\} & \rightarrow (Memory^*.performed_P \rightarrow Memory.performed_P)
\end{array}$$

Proof We satisfy the three requirements in [18], which is a bit complicated and takes a lot of paper. ⊠

Corollary 5.10 *Memory* is safe with respect to Memory.*

Proof The elaborate proof for Theorem 5.9 tells us that BACK is image-finite. Combining this observation with Theorem 5.9 and [18]'s Theorem 6.2, we obtain the desired result. ⊠

Theorem 5.11 *For each reachable, quiescent state s of $Memory^*$, each state $r \in \mathrm{BACK}(s)$ is a quiescent state of Memory.*

Proof Considering the preconditions of $Memory^*$, in each quiescent state s, $Memory^*.pc_P$ must be equal to WC for every P. For each $r \in \mathrm{BACK}(s) : r \models \bigwedge_P Memory.pc_P{=}\mathsf{WC}$, hence r is quiescent. ⊠

Corollary 5.12 *$Memory^*$ is deadlock free with respect to Memory.*

Proof By Theorems 5.9 and 5.11 we can construct, for each quiescent execution of $Memory^*$, a corresponding quiescent execution of *Memory* with the same trace. ⊠

Theorem 5.13 *$Memory^*$ implements Memory.*

Proof Assume that $\beta \in fairtraces(Memory^*)$. Let α be a fair execution of $Memory^*$ with the same trace β. If α is finite then α is quiescent and it follows by Corollary 5.12 that *Memory* has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in fairtraces(Memory)$. So we may assume without loss of generality that α is infinite.

Using the fact that BACK is a weak image-finite backward simulation (Theorem 5.9) we can easily construct an execution α' of *Memory* with trace β. It remains to prove that α' is fair.

We need to show that α' must be infinite. The only obstacle in this part is the GET_P action, since this is not always imitated by *Memory*. However, fairness helps us establish the fact that $Memory^*$ cannot do that continuously without issuing a return, and *Memory* imitates each last GET_P before that return. Infinity is then inevitable.

Using the above, the fairness of α' is satisfied quite trivially because of three facts. Firstly, $wfair(Memory) = wfair(Memory^*)$ and $sfair(Memory) = sfair(Memory^*) = \emptyset$. Secondly, if a weak fairness set is not enabled in $Memory^*$, it is certainly not enabled in *Memory*. Thirdly, infinitely many occurrences of action a in α cause infinitely many occurrences of a in α'. ⊠

5.3.2 MemoryImp implements $Memory^*$

Invariants The following list of invariants is rather dull. They are necessary for ensuring that the arguments of an incoming call are transmitted properly among the components of *MemoryImp*, and no component will act before it receives permission to do so.

Component *RPC* will remain quiescent until a request is issued by component *ClerkRPC*:

Lemma 5.14 *The following property* Inv3 *is an invariant of MemoryImp.*

$$\bigwedge_P \; ClerkRPC.pc_P{\neq}\mathsf{WR} \rightarrow RPC.pc_P{=}\mathsf{WC}$$

Component *RRMemory* will remain quiescent until a request is issued by component *RPC*:

Lemma 5.15 *The following property* Inv4 *is an invariant of MemoryImp.*

$$\bigwedge_P \; RPC.pc_P{\neq}\mathsf{WR} \rightarrow RRMemory.pc_P{=}\mathsf{WC}$$

Component *ClerkRPC* only handles read or write calls:

Lemma 5.16 *The following property* Inv5 *is an invariant of MemoryImp.*

$$\bigwedge_P\ ClerkRPC.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \vee \wedge\ ClerkRPC.proc_P = \mathsf{Read} \\ \quad \wedge \exists l : ClerkRPC.loc_P = l \\ \vee \wedge\ ClerkRPC.proc_P = \mathsf{Write} \\ \quad \wedge \exists l : ClerkRPC.loc_P = l \\ \quad \wedge \exists v : ClerkRPC.val_P = v \end{array}$$

Component *RPC* receives the same calls and arguments from *ClerkRPC*, as *ClerkRPC* received from the environment:

Lemma 5.17 *The following property* Inv6 *is an invariant of MemoryImp.*

$$\bigwedge_P\ RPC.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \wedge\ RPC.proc_P = ClerkRPC.proc_P \\ \wedge\ RPC.args_P = \mathsf{if}\quad ClerkRPC.proc_P = \mathsf{Read} \\ \qquad\qquad\qquad\quad \mathsf{then}\ (ClerkRPC.loc_P) \\ \qquad\qquad\qquad\quad \mathsf{else}\ (ClerkRPC.loc_P, ClerkRPC.val_P) \end{array}$$

Component *RPC* only receives read or write calls:

Corollary 5.18 *The following property* Inv7 *is an invariant of MemoryImp.*

$$\bigwedge_P\ RPC.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \vee\ RPC.proc_P = \mathsf{Read} \wedge \exists l : RPC.args_P = (l) \\ \vee\ RPC.proc_P = \mathsf{Write} \wedge \exists l, v : RPC.args_P = (l, v) \end{array}$$

Proof Directly from invariants Inv3, Inv5 and Inv6. ⊠

Since Read and Write are proper procedure names, and *RPC* receives no other procedure calls, the action BAD_CALL_P is never enabled:

Corollary 5.19 *The following property* Inv8 *is an invariant of MemoryImp.*

$$\bigwedge_P\ \neg enabled(BAD\_CALL_P)$$

If *RRMemory* is busy, it is by request of *RPC*, and the arguments have been transmitted properly:

Lemma 5.20 *The following property* Inv9 *is an invariant of MemoryImp.*

$$\begin{array}{ll} \bigwedge_P\ RRMemory.pc_P = \mathsf{R} & \rightarrow \wedge\ RPC.pc_P = \mathsf{WR} \\ & \quad \wedge\ RPC.proc_P = \mathsf{Read} \\ & \quad \wedge\ RPC.args_P = (l) \rightarrow RRMemory.loc_P = l \\ \bigwedge_P\ RRMemory.pc_P = \mathsf{W} & \rightarrow \wedge\ RPC.pc_P = \mathsf{WR} \\ & \quad \wedge\ RPC.proc_P = \mathsf{Write} \\ & \quad \wedge\ RPC.args_P = (l, v) \rightarrow \wedge\ RRMemory.loc_P = l \\ & \qquad\qquad\qquad\qquad\quad \wedge\ RRMemory.val_P = v \end{array}$$

RPC can only issue a return to *ClerkRPC*, following a (possibly exceptional) return by *RRMemory*, and the return value is transmitted properly:

Lemma 5.21 *The following property* Inv10 *is an invariant of MemoryImp.*

$$\begin{array}{l}\bigwedge_P\ RPC.pc_P{=}\mathsf{IR} \rightarrow \vee \wedge\ RRMemory.performed_P \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ RPC.return_P{=}\ \mathsf{if}\quad RPC.proc_P{=}\mathsf{Read} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ \ RRMemory.val_P \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{WriteOk} \\ \qquad\qquad\qquad\qquad\qquad \vee \wedge\ \neg RRMemory.legal\_args_P \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ RPC.return_P{=}\mathsf{BadArg}\end{array}$$

Inv11 states the same result as Inv10, for component *ClerkRPC*:

Lemma 5.22 *The following property* Inv11 *is an invariant of MemoryImp.*

$$\begin{array}{l}\bigwedge_P\ ClerkRPC.pc_P{=}\mathsf{IR} \rightarrow \vee \wedge\ RRMemory.performed_P \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ ClerkRPC.return_P{=}\ \mathsf{if}\quad ClerkRPC.proc_P{=}\mathsf{Read} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ \ RRMemory.val_P \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{WriteOk} \\ \qquad\qquad\qquad\qquad \vee \wedge\ \neg RRMemory.legal\_args_P \\ \qquad\qquad\qquad\qquad\qquad \wedge\ ClerkRPC.return_P{=}\mathsf{BadArg}\end{array}$$

$RRMemory.legal\_args_P$ behaves just like we expect it to, as long as *RRMemory* is busy:

Lemma 5.23 *The following property* Inv12 *is an invariant of MemoryImp.*

$$\begin{array}{l}\bigwedge_P\ RRMemory.pc_P{=}\mathsf{R}\ \rightarrow\ RRMemory.legal\_args_P{=}\ \mathsf{memloc}(RRMemory.loc_P) \\ \bigwedge_P\ RRMemory.pc_P{=}\mathsf{W} \rightarrow\ RRMemory.legal\_args_P{=}\ \wedge\ \mathsf{memloc}(RRMemory.loc_P) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{memval}(RRMemory.val_P)\end{array}$$

$RRMemory.legal\_args_P$ is not changed after *RRMemory* returns to *RPC*:

Lemma 5.24 *The following property* Inv13 *is an invariant of MemoryImp.*

$$\begin{array}{l}\bigwedge_P\ \vee\ RPC.pc_P{\in}\{\mathsf{WR},\mathsf{IR}\} \rightarrow \vee \wedge\ ClerkRPC.proc_P{=}\mathsf{Write} \\ \qquad\ \vee\ ClerkRPC.pc_P{=}\mathsf{IR} \qquad\qquad \wedge\ RRMemory.legal\_args_P{=}\ \wedge\ \mathsf{memloc}(ClerkRPC.loc_P) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{memval}(ClerkRPC.val_P) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \vee \wedge\ ClerkRPC.proc_P{=}\mathsf{Read} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ RRMemory.legal\_args_P{=}\mathsf{memloc}(ClerkRPC.loc_P)\end{array}$$

$Memory^*.legal\_args_P$ behaves just like we expect it to, as long as $Memory^*$ is busy:

Lemma 5.25 *The following property* Inv14 *is an invariant of* $Memory^*$.

$$\begin{array}{l}\bigwedge_P\ pc_P{=}\mathsf{R}\ \rightarrow legal\_args_P{=}\mathsf{memloc}(loc_P) \\ \bigwedge_P\ pc_P{=}\mathsf{W} \rightarrow legal\_args_P{=}\mathsf{memloc}(loc_P) \wedge \mathsf{memval}(val_P)\end{array}$$

Safety We use a weak forward simulation, instead of a weak refinement. In fact, a weak refinement does not exist from *MemoryImp* to $Memory^*$. Suppose *RPC* receives a call from P for the first time, and *MemoryImp* transits to state s. We can only ensure that $Memory^*$ returns the same value as *RRMemory* if they read and write simultaneously. So in the image state of s, $Memory^*.performed_P$ must be false. If *RPC* returns a fail to *ClerkRPC*, *ClerkRPC*

is allowed to retry the call. This may lead to the same state s again. However, *Memory** has imitated the read or write actions performed by *RRMemory*, and $Memory^*.performed_P$ may be true. So a refinement should map s onto a state in which $Memory^*.performed_P$ is both true and false, which is a contradiction.

Theorem 5.26 *The relation* SIM *defined by the following formula is a weak forward simulation from MemoryImp to Memory*, with respect to the reachable states in both MemoryImp and Memory*.*

$$
\begin{array}{lll}
\bigwedge_P\ Memory^*.pc_P & = & \mathsf{if}\quad ClerkRPC.pc_P{=}\mathsf{WC} \\
 & & \mathsf{then\ WC} \\
 & & \mathsf{else\ if}\ ClerkRPC.proc_P{=}\mathsf{Read\ then\ R\ else\ W} \\
\bigwedge_P\ Memory^*.loc_P & = & ClerkRPC.loc_P \\
\wedge\quad Memory^*.memory & = & RRMemory.memory \\
\bigwedge_P\ ClerkRPC.proc_P{=}\mathsf{Write} & \rightarrow & Memory^*.val_P = ClerkRPC.val_P \\
\bigwedge_P\ \wedge\ \vee\ RPC.pc_P{\in}\{\mathsf{WR},\mathsf{IR}\} & \rightarrow & \wedge\ Memory^*.performed_P \\
\quad\ \vee\ ClerkRPC.pc_P{=}\mathsf{IR} & & \wedge\ Memory^*.val_P = RRMemory.val_P \\
\quad\wedge\ RRMemory.performed_P & &
\end{array}
$$

Proof We use the following property.
For each two reachable states s in *MemoryImp*, r in *Memory**:

$$
\begin{array}{l}
r, s \models \bigwedge_P\ Memory^*.pc_P{=}\mathsf{R} \rightarrow Memory^*.legal\_args_P{=}\mathsf{memloc}(ClerkRPC.loc_P) \\
\qquad \bigwedge_P\ Memory^*.pc_P{=}\mathsf{W} \rightarrow Memory^*.legal\_args_P{=} \wedge\ \mathsf{memloc}(ClerkRPC.loc_P) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathsf{memval}(ClerkRPC.val_P)
\end{array}
$$

This follows directly from Inv5, Inv14 and the definition of SIM. Using this property, and the invariants Inv3, Inv5, Inv6, Inv8, Inv9, Inv11 and Inv13, the proof is a straightforward fulfillment of the two requirements in [18]. ⊠

Corollary 5.27 *MemoryImp is safe with respect to Memory*.*

Proof Directly from theorem 5.26 and [18]'s Theorem 6.2. ⊠

Deadlock freeness In order to establish that *MemoryImp* is deadlock free with respect to *Memory**, we need an additional invariant. It expresses that as long as *ClerkRPC* is waiting for a return, *RPC* is busy. Likewise, if *RPC* is waiting for a return, *RRMemory* is busy.

Lemma 5.28 *The following property* Inv15 *is an invariant of MemoryImp.*

$$
\begin{array}{ll}
\bigwedge_P\ ClerkRPC.pc_P{=}\mathsf{WR} & \rightarrow\ RPC.pc_P{\neq}\mathsf{WC} \\
\bigwedge_P\ RPC.pc_P{=}\mathsf{WR} & \rightarrow\ RRMemory.pc_P{\in}\{\mathsf{R},\mathsf{W}\}
\end{array}
$$

Theorem 5.29 *For each reachable and quiescent state s of MemoryImp, each reachable state $r \in Memory^*$ such that $r, s \models$ SIM is a quiescent state of Memory*.*

Proof From the action types of *MemoryImp* and Inv15, we can conclude that *MemoryImp* is quiescent in state s iff $s \models ClerkRPC.pc_P{=}\mathsf{WC}$. Since $r, s \models$ SIM, obviously $r \models Memory^*.pc_P{=}\mathsf{WC}$, hence r is quiescent. ⊠

Corollary 5.30 *MemoryImp is deadlock free with respect to Memory*.*

Proof By Theorems 5.26 and 5.29 we can construct for each quiescent execution of *MemoryImp*, a corresponding quiescent execution of *Memory** with the same trace. ⊠

Theorem 5.31 *MemoryImp implements Memory*.*

Proof We prove $fairtraces(MemoryImp) \subseteq fairtraces(Memory^*)$.
Assume that $\beta \in fairtraces(MemoryImp)$. Let α be a fair execution of *MemoryImp* with trace β. If α is finite then α is quiescent and it follows by Corollary 5.30 that *Memory** has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in fairtraces(Memory^*)$. So we may assume without loss of generality that α is infinite.

Using the fact that SIM is a weak forward simulation (Theorem 5.26) we can easily construct an execution α' of *Memory** with the same trace β. It remains to prove that α' is fair.

First we show that α' is infinite. Then we observe that each non-discarded call to *MemoryImp* will lead to a return within a finite number of steps. Using these two facts, we can easily show for each class in $wfair(Memory^*)$, that α' satisfies the requirements for weak fairness. Since $sfair(Memory^*)$ is empty, this is enough to show that α' is fair. ⊠

5.3.3 The main result

Theorem 5.32 *MemoryImp implements Memory.*

Proof Theorems 5.31 and 5.13 yield $fairtraces(MemoryImp) \subseteq fairtraces(Memory)$. ⊠

6. Specifications for Problem 4

6.1 Problem (4): Specification of a lossy RPC

The lossy RPC is a timed version of the RPC component, as specified in section 4.1. The difference between timed and untimed I/O automata is that time-passage is made explicit by the action *TIME*, and that the fairness constraints are translated into timing restrictions.

6.1.1 Data types We reuse the ingredients of Σ_2 and $\mathcal{A}_2$, given in section 4.1, and add the data type **Time** to obtain a typed signature Σ_4 and a Σ_4-algebra $\mathcal{A}_4$. **Time** is the set $\mathsf{R}^{\geq 0}$ of nonnegative real numbers, with the usual interpretation and functions for addition (+) and multiplication (.).

6.1.2 We will now present the I/O automaton *LossyRPC*, which models a lossy RPC component. It has a new state variable $clock_P$ for each calling process, to keep track of the time elapsed since the last call was received from the sender, or issued to the receiver.

Also a time-passing action *TIME* is added. We let time increase without bounds, except in states where a certain output action should be issued within δ seconds. Here we forbid time passing if it violates this bound.

The code for I/O automaton *LossyRPC* is given in figure 6. Since *LossyRPC* is very similar to *RPC*, we highlight with boxes where the code differs.

Input: REM_CALL_P, I_RETURN_P
Output: $I\_CALL_P, REM\_RETURN_P, BAD\_CALL_P$
State Variables: pc_P: **Cpc** — **Initialization:** $\bigwedge_P\ pc_P$=WC
$proc_P$: **Names**
$args_P$: **Args**
$legal\_call_P$: **Bool**
$return_P$: **ReturnVal**
$clock_P$: **Time**

$REM\_CALL_P(p : \textbf{Names}, a : \textbf{Args})$
Effect:
if pc_P=WC then [$proc_P := p$
$args_P := a$
$legal\_call_P :=$ legal_proc(p) $\wedge$ (num(a)=arg_num(p))
$pc_P :=$ IC
$clock_P := 0$]

$I\_CALL_P(p : \textbf{Procs}, a : \textbf{Args})$
Precondition:
pc_P=IC $\wedge$ $legal\_call_P \wedge p$=$proc_P \wedge a$=$args_P$
Effect:
$pc_P :=$ WR

$I\_RETURN_P(r : \textbf{ReturnVal})$
Effect:
if pc_P=WR then [$pc_P :=$ IR
$return_P := r$
$clock_P := 0$]

$REM\_RETURN_P(r : \textbf{ReturnVal})$
Precondition:
pc_P=IR $\wedge$ r=$return_P$
Effect:
$pc_P :=$ WC

BAD_CALL_P
Precondition:
pc_P=IC $\wedge \neg legal\_call_P$
Effect:
$pc_P :=$ WC

$TIME(t : \textbf{Time})$
Precondition:
$\bigwedge_P\ pc_P \in \{\text{IC}, \text{IR}\} \rightarrow clock_P + t \leq \delta$
Effect:
$\forall P$: $clock_P := clock_P + t$

Figure 6: I/O automaton *LossyRPC*

Input: REM_CALL_P, REM_RETURN_P
Output: $RPC\_FAILURE_P$
State Variables: pc_P: **Rpc** **Initialization:** $\bigwedge_P\ pc_P$=WC
$clock_P$: **Time**

$REM\_CALL_P(p : \mathbf{Names}, a : \mathbf{Args})$
Effect:
if pc_P=WC then $[pc_P := \mathsf{WR}$
$clock_P := 0]$

$REM\_RETURN_P(r : \mathbf{ReturnVal})$
Effect:
if pc_P=WR $\wedge$ $clock_P \leq 2\delta + \epsilon$ then $[pc_P := \mathsf{WC}]$

$RPC\_FAILURE_P$
Precondition:
pc_P=WR $\wedge$ $clock_P > 2\delta + \epsilon$
Effect:
$pc_P := \mathsf{WC}$

$TIME(t : \mathbf{Time})$
Precondition:
true
Effect:
$\forall P$: $clock_P := clock_P + t$

Figure 7: Timed I/O automaton *ClerkLossy*

7. Specifications and verifications for Problem 5

To model an implementation as specified, we need more than the specification of *LossyRPC*. There has to be some sort of monitoring component, that signals the need for a failure output action and issues this failure.

7.1 Problem (5): Specification of a clerk

7.1.1 Data types We reuse the ingredients of Σ_4 and $\mathcal{A}_4$, given in Section 6.1, and add the data types **Cpc** and **Epc** to obtain a typed signature Σ_5, and a Σ_5-algebra $\mathcal{A}_5$. **Cpc** only contains the constants WC and WR. **Epc** only contains the constants WC and IR. Note that the domains of **Cpc** and **Epc** are subsets of the domain of **Rpc**.

7.1.2 Specification We will now present the timed I/O automaton *ClerkLossy*, which models a clerk for the lossy RPC component *LossyRPC*. The domain of $ClerkLossy.pc_P$ contains only two possible values, namely WC and WR. It resets its clock when it signals that *LossyRPC* receives a call from the environment. Then it waits for *LossyRPC* to issue a return within the given bound of $2\delta + \epsilon$ seconds. If *LossyRPC* is not fast enough, *ClerkLossy* assumes that no return will occur, and it issues a $RPC\_FAILURE_P$. For this purpose, *ClerkLossy* has a clock for each process that might issue a call.

Note that REM_CALL_P is an input action for both *LossyRPC* and *ClerkLossy*, and that the output action REM_RETURN_P should be received by both *ClerkLossy* and the environment.

The code for *ClerkLossy* is listed in figure 7.

7.1.3 The composition The implementation *RPCImp* is the composition of the two automata:

$RPCImp \triangleq ClerkLossy || LossyRPC$

7.2 Problem 5(b): RPCImp implements RPC

What we have now is an implementation in with real-time aspects, and an untimed specification. To compare these, we can add time to the specification and prove an 'admissible trace'-inclusion. However, when changing from untimed to timed I/O automata, one expects the fairness restrictions on the automaton's behaviour to be encoded in the real-time aspects. Clearly, these restrictions are lost if we consider the timed specification's admissible traces. A possible solution is to consider the traces that are both admissible, and fair in the sense that we know from the untimed model.

Fair timed traces We need a yet undefined notion of *fair timed I/O automata*, to be able to consider only those executions that show a fair behaviour towards certain discrete actions. Although carrying fairness semantics over from the untimed model to a timed model is very tricky in general, we get away with the same definition as for the untimed case, since in our automata time-passing actions cannot change enabledness of discrete actions. So we assume that the addition of weak and strong fairness sets (over discrete actions) to a timed I/O automaton yields a fair timed I/O automaton, with fair executions as usual.

We will denote the fair timed I/O automaton, constructed from timed I/O automaton A, the collection of weak fairness sets W, and the collection of strong fairness sets S by $fta(A, W, S)$.

Given a fair timed I/O automaton A, the timed traces derived from $fairexecs(A)$ are denoted by $fair\text{-}t\text{-}traces(A)$.

Another problem concerning the implementation relation is that we need to formalize the restriction on the environment, namely that each legal procedure-call that is forwarded by $LossyRPC$, will return within ϵ seconds.

Since there is no straightforward way to express this type of restrictions in I/O automata theory, we model this restriction by a very general timed I/O automaton Env, that takes each call from $LossyRPC$ as input, and returns some answer within ϵ seconds.

The code for Env is listed in figure 8. It receives a call, instantaneously performs a symbolic function compute with the parameters received, and issues a return. The time-passing action $TIME$ ensures that time will not proceed too far before the return has been issued.

Note that we can easily regard the memory components as instances of this general receiver Env.

The compositions and the inclusion The composition for implementation is

$Imp \triangleq (RPCImp || EnvLossy)$

The timed I/O automaton $TimeRPC$ is the untimed RPC plus an extra action $TIME(t :$ **Time**). The precondition of $TIME$ is true, the effect is empty (no state variables change).

The composition for the specification is

$Spec \triangleq (TimeRPC || EnvRPC)$

The implementation relation will be proved by the inclusion

Input: I_CALL_P
Output: I_RETURN_P
State Variables: pc_P: **Epc** — **Initialization:** $\bigwedge_P\ pc_P{=}\mathsf{WC}$
$return_P$: **ReturnVal**
$clock_P$: **Time**

$I\_CALL_P(p : \textbf{Procs}, a : \textbf{Args})$
Effect:
if $pc_P{=}\mathsf{WC}$ then $[return_P := \mathsf{compute}(p, a)$
$clock_P := 0$
$pc_P := \mathsf{IR}]$

$I\_RETURN_P(r : \textbf{ReturnVal})$
Precondition:
$pc_P{=}\mathsf{IR} \wedge r{=}return_P$
Effect:
$pc_P := \mathsf{WC}$

$TIME(t : \textbf{Time})$
Precondition:
$\bigwedge_P\ pc_P{=}\mathsf{IR} \rightarrow clock_P + t \leq \epsilon$
Effect:
$\forall P$: $clock_P := clock_P + t$

Figure 8: Timed I/O automaton Env

$$t\text{-}traces^{\infty}(Imp) \subseteq (t\text{-}traces^{\infty}(Spec) \cap fair\text{-}t\text{-}traces(fta(Spec, wfair(RPC), \emptyset)))$$

so we will first prove $t\text{-}traces^{\infty}(Imp) \subseteq t\text{-}traces^{\infty}(Spec)$, by means of a weak refinement, and then $t\text{-}traces^{\infty}(Imp) \subseteq fair\text{-}t\text{-}traces(fta(Spec, wfair(RPC), \emptyset))$.

In the remainder, we will mostly reason about 'sampling' executions instead of timed executions. Since Lemmas 2.11 - 2.13 in [17] state that both induce the same set of timed traces, and we only consider trace inclusion, this does not make a difference.

7.2.1 Admissible trace inclusion

Lemma 7.1 *The following property* InvT1 *is an invariant of Imp:*

$$\bigwedge_P\ LossyRPC.pc_P{=}\mathsf{WC} \rightarrow ClerkLossy.pc_P{=}\mathsf{WC} \wedge EnvLossy.pc_P{=}\mathsf{WC}$$
$$\bigwedge_P\ EnvLossy.pc_P{\neq}\mathsf{WC} \rightarrow LossyRPC.pc_P{=}\mathsf{WR}$$

Lemma 7.2 *The following property* InvT2 *is an invariant of Imp:*

$$\bigwedge_P\ LossyRPC.pc_P{\in}\{\mathsf{IC}, \mathsf{IR}\} \rightarrow (LossyRPC.clock_P \leq \delta)$$
$$\bigwedge_P\ LossyRPC.pc_P{=}\mathsf{WR} \rightarrow (EnvLossy.clock_P \leq \epsilon)$$

Lemma 7.3 *The following property* InvT3 *is an invariant of Imp:*

$$\bigwedge_P\ LossyRPC.pc_P{=}\mathsf{IC} \rightarrow (ClerkLossy.clock_P{=}LossyRPC.clock_P)$$
$$\bigwedge_P\ LossyRPC.pc_P{=}\mathsf{WR} \rightarrow (ClerkLossy.clock_P \leq EnvLossy.clock_P + \delta)$$
$$\bigwedge_P\ LossyRPC.pc_P{=}\mathsf{IR} \rightarrow (ClerkLossy.clock_P \leq LossyRPC.clock_P + \delta + \epsilon)$$

Corollary 7.4 *The following property* InvT4 *is an invariant of Imp:*

$$\bigwedge_P\ \neg enabled(RPC\_FAILURE_P)$$

7.2.2 Weak refinement

Theorem 7.5 *The function* TREF *which combines the identity functions on variables with the same name from LossyRPC to TimeRPC, and from EnvLossy to EnvRPC, is a weak timed refinement from Imp to Spec, with respect to the reachable states in Imp and Spec.*

Proof A straightforward fulfillment of the requirements in [17]. ⊠

Corollary 7.6 $t\text{-}traces^{\infty}(Imp) \subseteq t\text{-}traces^{\infty}(Spec)$

Proof Directly from Theorem 7.5 and [17]'s Theorem 8.2. ⊠

7.2.3 Fairness is preserved To prove that each timed trace of *Imp* is also the timed trace of a fair execution of *Spec*, we prove first that within *Imp*, each call from the environment leads to a return.

Lemma 7.7 *Let* $s_0 a_1 s_1 a_2 s_2 \ldots$ *be an admissible execution of LossyRPC.*
Then $a_i = REM\_CALL_P$ *and* $s_{i-1} \models pc_P{=}\mathsf{WC}$ *implies that there is a* j *such that* $j > i$, $a_j \in \{REM\_RETURN_P, BAD\_CALL_P\}$, *and the sum of time passing between* s_{i-1} *and* s_{j-1} *is bounded.*

Proof Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ is an admissible execution of *LossyRPC*,
$a_i = REM\_CALL_P$ and $s_{i-1} \models pc_P{=}\mathsf{WC}$.

Clearly, $s_i \models pc_P{=}\mathsf{IC} \wedge clock_P{=}0$. By InvT2 and the definition of *TIME* we know that each following *TIME*-step leads to a state where either *TIME* and I_CALL_P are enabled, or only I_CALL_P is enabled. Since the total time passing with subsequent *TIME*-transitions is bounded, some a_k must be equal to I_CALL_P $(k > i)$. By applying a similar argument twice, we arrive at the obligatory occurrence of either REM_RETURN_P or BAD_CALL_P and the boundedness of the sum of time passing. ⊠

Theorem 7.8 $t\text{-}traces^{\infty}(Imp) \subseteq fair\text{-}t\text{-}traces(fta(Spec, wfair(RPC), \emptyset))$

Proof Suppose β is a timed trace of *Imp*, and $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ is an admissible execution of *Imp* such that $t\text{-}trace(\alpha) = \beta$. By Theorem 7.5 we know that *Spec* has an admissible execution α' such that $\alpha' = \mathrm{TREF}(s_0) a_1 \mathrm{TREF}(s_1) a_2 \mathrm{TREF}(s_2) \ldots$ and $t\text{-}trace(\alpha') = \beta$. It remains to prove that α' is fair.

Lemma 7.7 helps us in proving that for each P, α must contain infinitely many occurrences of states such that $LossyRPC.pc_P = \mathsf{WC}$. All start states satisfy this property, and each action that changes $LossyRPC.pc_P$ from such a state, must be an I_CALL_P and must be followed within a bounded amount of time by a new state in which $LossyRPC.pc_P = \mathsf{WC}$. Using this and InvT1, we observe that for each P, α must contain infinitely many occurrences of states such that both $LossyRPC.pc_P$ and *EnvLossy* are equal to WC.

By definition, for each P, α' must contain infinitely many occurrences of states such that $TimeRPC.pc_P$ and *EnvRPC* are equal to WC. Since in such a state no discrete internal actions are enabled, α' must be weakly fair. Combining this with the fact that there are no strong fairness sets in $fta(Spec, wfair(RPC), \emptyset)$, we obtain that α' is fair. ⊠

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 1992.

2. M. Abadi, L. Lamport, and S. Merz. The Dagstuhl problem – a TLA+ solution, August 1994. Available through URL `http://www.research.digital.com/SRC/personal/Leslie_Lamport/dagstuhl/dagstuhl.html`.

3. E. Astesiano and G. Reggio. A case study in friendly specifications of concurrent systems: The RPC-Memory specification problem, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

4. E. Best. A memory module specification using composable high level nets, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

5. J. Blom and B. Jonsson. Architecture oriented temporal logic specification, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

6. M. Broy. A functional solution to the RPC-Memory specification problem, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

7. M. Broy and L. Lamport. Specification problem, August 1994. Available through the URL `http://www.research.digital.com/SRC/personal/Leslie_Lamport/dagstuhl/all.html`.

8. J.R. Cuéllar, D. Barnard, and M. Huber. A solution relying on the model checking of boolean transition systems, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

9. R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *Proceedings* 21^{th} *ICALP,* Jerusalem, volume 820 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. A full version appears as MIT Technical Report number MIT/LCS/TR-587.

10. R. Gotzhein. Applying a temporal logic to the RPC-Memory specification problem, 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

11. J. Hooman. Using PVS for an assertional verification of the RPC-Memory specification problem, November 1995. Available through URL `http://www.win.tue.nl/win/cs/tt/hooman/RPC.html`.

12. H. Hungar. Using temporal logic — specification for verification (draft), 1995. Available through URL `http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html`.

13. R. Kurki-Suonio. Incremental specification with joint actions: The RPC-Memory specification problem, 1995. Available through URL `http://www4.informatik.`

tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html.

14. K.G. Larsen, B. Steffen, and C. Weise. The methodology of modal constraints, 1995. Available through URL http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html.
15. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
16. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
17. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part II: Timing-based systems. Report CS-R9314, CWI, Amsterdam, March 1993. Also, MIT/LCS/TM-487.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. To appear in *Information and Computation.*
18. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations. part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.
19. M. Rinderspacher. The solution of the RPC-Memory specification problem using reachability analysis, 1995. Available through URL http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html.
20. J.M.T. Romijn and F.W. Vaandrager. A note on fairness in I/O automata. Report CS-R9579, CWI, Amsterdam, December 1995.
21. K. Stølen. RPC-Memory specification problem, 1995. Available through URL http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html.
22. R.T. Udink and J.N. Kok. The RPC-Memory specification problem: UNITY plus refinement calculus, 1995. Available through URL http://www4.informatik.tu-muenchen.de/~spiesk/DAGSTUHL/Solutions.html.

A. Safe and Fair I/O automata

In this appendix we review some basic definitions from [9, 20].

Safe I/O automata A *safe I/O automaton* B consists of the following components:

- A set $states(B)$ of *states* (possibly infinite).
- A nonempty set $start(B) \subseteq states(B)$ of *start states*.
- A set $acts(B)$ of *actions*, partitioned into three sets $in(B)$, $int(B)$ and $out(B)$ of *input*, *internal* and *output* actions, respectively. Actions in $local(B) \stackrel{\Delta}{=} out(B) \cup int(B)$ are called *locally controlled*.
- A set $steps(B) \subseteq states(B) \times acts(B) \times states(B)$ of *transitions*, with the property that for every state s and input action $a \in in(B)$ there is a transition $(s, a, s') \in steps(B)$.

We let s, s',.. range over states, and a,.. over actions. We write $s \stackrel{a}{\longrightarrow}_B s'$, or just $s \stackrel{a}{\longrightarrow} s'$ if B is clear from the context, as a shorthand for $(s, a, s') \in steps(B)$.

Enabling of actions An action a of a safe I/O automaton B is *enabled* in a state s iff $s \stackrel{a}{\longrightarrow} s'$ for some s'. Since every input action is enabled in every state, safe I/O automata are said to be *input enabled*. The intuition behind the input-enabling condition is that input actions are under control of the environment and that the system that is modeled by an safe I/O automaton cannot prevent the environment from doing these actions.

Executions An *execution fragment* of a safe I/O automaton B is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of B, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \stackrel{a_{i+1}}{\longrightarrow} s_{i+1}$. An *execution* is an execution fragment that begins with a start state. We write $execs^*(B)$ for the set of finite executions of B, and $execs(B)$ for the set of all executions of B. A state s of B is *reachable* if it is the last state of some finite execution of B.

Fair I/O automata A *fair I/O automaton* A is a triple consisting of

- a safe I/O automaton $safe(A)$, and
- sets $wfair(A)$ and $sfair(A)$ of subsets of $local(safe(A))$, called the *weak fairness sets* and *strong fairness sets*, respectively.

Enabling of sets Let U be a set of actions of a fair I/O automaton A. Then U is *enabled* in a state s iff an action from U is enabled in s. Set U is *input resistant* if and only if, for each pair of reachable states s, s' and for each input action a, s enables U and $s \stackrel{a}{\longrightarrow} s'$ implies s' enables U. So once U is enabled, it can only be disabled by the occurrence of a locally controlled action.

Fair executions An execution α of a fair I/O automaton A is *weakly fair* if the following conditions hold for each $W \in \mathit{wfair}(A)$:

1. If α is finite then W is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from W, or α contains infinitely many occurrences of states in which W is not enabled.

Execution α is *strongly fair* if the following conditions hold for each $S \in \mathit{sfair}(A)$:

1. If α is finite then S is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from S, or α contains only finitely many occurrences of states in which S is enabled.

Execution α is *fair* if it is both weakly and strongly fair. In a fair execution each weak fairness set gets turns if enabled continuously, and each strong fairness set gets turns if enabled infinitely many times. We write $\mathit{fairexecs}(A)$ for the set of fair executions of A.

We write $\mathit{live}(A)$ for the underlying safe I/O automaton of A paired with $\mathit{fairexecs}(A)$: $\mathit{live}(A) \stackrel{\Delta}{=} (\mathit{safe}(A), \mathit{fairexecs}(A))$.