A TLA Solution to the RPC-Memory Specification Problem

Martín Abadi, Leslie Lamport, and Stephan Merz

Table of Contents

A TLA Solution to the RPC-Memory Specification Problem

Martín Abadi[1], Leslie Lamport[1], and Stephan Merz[2]

[1] Systems Research Center, Digital Equipment Corporation
[2] Institut für Informatik, Technische Universität München

Abstract. We present a complete solution to the Broy-Lamport specification problem. Our specifications are written in TLA$^+$, a formal language based on TLA. We give the high levels of structured proofs and sketch the lower levels, which will appear in full elsewhere.

Introduction

Broy and Lamport have proposed a specification and verification problem [5]. It calls for specifying simple memory and RPC (remote procedure call) components, and proving the correctness of two simple implementations. We present a complete solution to this problem using TLA, the temporal logic of actions [12]. We assume the reader is familiar with Broy and Lamport's problem statement.

Since the problem is so much simpler than the ones encountered in real applications, any approach that claims to be both practical and formal should allow a completely formal solution. Our specifications are written in TLA$^+$, a formal language based on TLA. Our proofs are completely formal, except that some names are abbreviated for readability. We use a hierarchical proof method [10] that is the most reliable way we know of to write hand proofs. Here, we present only the higher levels of the proofs. Proofs carried down to the level where each justification involves instantiation of proof rules and simple predicate logic will be available on a Web page [4]. Although our proofs are careful and detailed, neither they nor the specifications have been checked mechanically; minor errors undoubtedly remain.

Rigor entails a certain degree of tedium. A complete programming language requires boring details like variable declarations that can be omitted in informal pseudo-code. Writing specifications is harder with a formal language than with an informal approach—even one based on a formalism. Formal proofs that are detailed enough to be easy to check are long and boring. However, rigor has its advantages. Informal specifications can be ambiguous. The short, interesting proofs favored by mathematicians are notoriously error prone. Our specifications and proofs are rigorous, hence somewhat laborious.

We assume no prior knowledge of TLA or TLA$^+$. Concepts and notations are explained as they are introduced; the index on page 65 can help the reader find those explanations. TLA is described in detail in [12], and there are several published examples of TLA$^+$ specifications [11, 14]. Further information about TLA and TLA$^+$ can be found on the Web [9].

The problem is not very challenging for TLA, TLA$^+$, or our proof style. With our experience, it was possible to grind out the requisite specifications and proofs without much thought. More difficult was choosing from among the many possible ways of writing the specifications. We tried to make the specifications as clear as possible without unduly complicating the correctness proofs. We benefited from studying the many preliminary solutions presented at a Dagstuhl workshop on the specification problem. In particular, we emulated some of these solutions by writing our specifications as the composition of individual process specifications. We also benefited from comments by Ketil Stølen.

We found no significant ambiguities in the problem statement, perhaps because we had first-hand knowledge of the authors' intent. However, we did discover some anomalies in the lossy-RPC specification, which we discuss in Section 4. Our presentation parallels Broy and Lamport's problem statement. In particular, our section numbering corresponds to theirs, with the addition of lower-level sections.

1 The Procedure Interface

A TLA specification is a temporal-logic formula. It expresses a predicate on behaviors, where a behavior is an infinite sequence of states and a state is an assignment of values to variables. TLA$^+$ is a formal language for writing TLA specifications. It introduces precise conventions for definitions and a module system with name scoping that is modeled after those of programming languages. In this paper, we describe TLA and TLA$^+$ as they are used.

TLA does not have any built-in communication primitives such as message passing or data streams. One can use TLA to define such primitives.[3] We begin by specifying a procedure-calling interface in which a multiprocess caller component interacts with a multiprocess returner component. In this section, we present a module named *ProcedureInterface* that is meant to help specify systems that use the procedure-calling interface. As we explain below, a system may have several such interfaces, described by different "copies" of the module.

We describe a rather arbitrary, abstract procedure-calling interface. One might want a specification that describes an actual procedure-calling software standard, complete with register-usage conventions. One might also want a different high-level abstraction. We can convert our specifications into ones with a different interface abstraction by using an *interface refinement*, as described in [3, page 518] and [7].

Our specification makes precise one important detail that is not quite stated in the informal specification. We interpret the requirement:

> [A]fter one process issues a call, other processes can issue calls to the

[3] Like most logics, TLA uses variables. One could therefore say that TLA formulas use shared variables as a communication primitive. In the same sense, one could say that the equations $x + y = 7$ and $x - y = 1$ communicate via the shared variables x and y.

same component before the component issues a return from the first call.

to imply that the same process cannot issue another call until the first one returns.

1.1 The Module and its Parameters

Module *ProcedureInterface* is given in Figure 1 on this page. The module first declares some parameters, which are the free symbols that may appear in the expressions defined by the module. By replacing defined symbols with their definitions, all expressions defined in the module can be reduced to ones containing only the parameters and the primitives of TLA$^+$. The parameter ch is the variable representing the interface. A VARIABLE parameter can have a different value in different states of a behavior. TLA is an untyped logic, so there are no type constraints on the values a variable can have. A CONSTANT parameter is one that has the same value in every state of a behavior. The constant parameter $PrIds$ is the set of all process identifiers; for each p in $PrIds$, process p of the caller component issues calls to the corresponding process p of the returner component. The parameter $Args$ is the set of all "syntactically correct" procedure arguments.

module *ProcedureInterface*

parameters
$PrIds, Args$: CONSTANT
ch : VARIABLE

$$caller(p) \triangleq \langle ch[p].arg, ch[p].cbit \rangle$$
$$rtrner(p) \triangleq \langle ch[p].res, ch[p].rbit \rangle$$
$$Calling(p) \triangleq ch[p].cbit \neq ch[p].rbit$$
$$\begin{aligned} Call(p, v) \triangleq\ & \land \neg Calling(p) \\ & \land ch[p].cbit' \neq ch[p].rbit \\ & \land ch[p].arg' = v \end{aligned}$$
$$\begin{aligned} Return(p, v) \triangleq\ & \land Calling(p) \\ & \land ch[p].rbit' = ch[p].cbit \\ & \land ch[p].res' = v \end{aligned}$$

$$LegalCaller \triangleq \forall p \in PrIds : \neg Calling(p) \land \Box[\exists a \in Args : Call(p, a)]_{caller(p)}$$
$$LegalReturner \triangleq \forall p \in PrIds : \Box[\exists v : Return(p, v)]_{rtrner(p)}$$

Fig. 1. Module *ProcedureInterface*.

Suppose some module M has a set P of process identifiers and two procedure-calling interfaces, represented by the variables x and y, with syntactically correct argument values in sets S_x and S_y, respectively. Module M will include all the definitions from module *ProcedureInterface* twice, with the following substitutions for its parameters:

$$ch \leftarrow x,\ \ PrIds \leftarrow P,\ \ Args \leftarrow S_x$$
$$ch \leftarrow y,\ \ PrIds \leftarrow P,\ \ Args \leftarrow S_y$$

It is conventional to follow the parameter declarations with a horizontal bar. These bars have no semantic significance.

1.2 State Functions, State Predicates, and Actions

To model the procedure-calling interface, we let $ch[p]$ be a "channel" over which process p of the caller component interacts with process p of the returner component. Our model uses a standard two-phase handshake protocol [16] illustrated in Figure 2 on this page. Channel $ch[p]$ contains two "wires" controlled by the

	initial state	*call* $Read(23)$	*return* $.333$	*call* $Write(14, 3.5)$	
$ch[p].cbit$:	0	1	1	0	...
$ch[p].arg$:	–	⟨"Read", 23⟩	⟨"Read", 23⟩	⟨"Write", 14, 3.5⟩	...
$ch[p].rbit$:	0	0	1	1	...
$ch[p].res$:	–	–	.333	.333	...

Fig. 2. The two-phase handshake protocol for the channel $ch[p]$.

caller—a signaling wire $ch[p].cbit$ and an argument-passing wire $ch[p].arg$—and two wires controlled by the returner—a signaling wire $ch[p].rbit$ and a result-returning wire $ch[p].res$.

In the standard two-phase handshake protocol shown in Figure 2, the signaling values $ch[p].cbit$ and $ch[p].rbit$ are bits. For simplicity, we allow them to assume arbitrary values, since all that matters is whether or not they equal one another.

The *ProcedureInterface* module defines the state function $caller(p)$ to be the pair $\langle ch[p].arg, ch[p].cbit\rangle$ composed of the process p caller's wires. A state function is an expression that may contain variables and constants. It is interpreted semantically as a mapping from states to values. For example, $ch[p].arg$ is the state function that assigns to any state the *arg* record component of the pth array element of the value that the state assigns to the variable ch.[4] The state

[4] This value is unspecified if the value assigned to ch by the state is not an array whose pth element is a record with an *arg* component.

function $rtrner(p)$ is similarly defined to be the pair composed of the returner's wires.

The module defines the state predicate $Calling(p)$ to equal TRUE iff (if and only if) the values on the two signaling wires are unequal. A state predicate is a boolean-valued expression that may contain variables and constants; it is interpreted semantically as a mapping from states to booleans. For the handshake protocol, $Calling(p)$ equals TRUE iff process p is in the middle of a procedure call (the caller has issued a call and the returner has not yet returned).

Next comes the definition of the action $Call(p, v)$. An action is a boolean-valued expression that may contain variables and constants, and the operator $'$ (prime), which may not be nested. Semantically, it is interpreted as a boolean-valued function on steps, where a step is a pair of states. Unprimed expressions refer to the first (old) state, and primed expressions refer to the second (new) state. For example, the action $(x + 1)' = y$ is true of a step iff 1 plus the value assigned to x by the new state equals the value assigned to y by the old state. Action $Call(p, v)$ describes the issuing of a call with argument v by the process p caller. More precisely, a step represents this event iff it is a $Call(p, v)$ step (one for which $Call(p, v)$ equals TRUE). The first conjunct[5] in the definition asserts that a call on channel $ch[p]$ is not in progress. The second conjunct asserts that the new value of $ch[p].cbit$ is different from the old value of $ch[p].rbit$. The final conjunct asserts that the new value of $ch[p].arg$ equals v. Readers familiar with conventional programming languages or state-transition systems can think of $Call(p, v)$ as an atomic statement or transition that is enabled when $\neg Calling(p, v)$ is true, that nondeterministically sets $ch[p].rbit$ to any value different from $ch[p].cbit$, and that sets $ch[p].arg$ to v.

Action $Return(p, v)$ represents the issuing of a return with result v by the process p returner. We do not distinguish in the interface description between normal and exceptional returns—the distinction will be encoded in the result v.

1.3 Temporal Formulas

Module *ProcedureInterface* concludes by defining the two temporal formulas *LegalCaller* and *LegalReturner*. Formula *LegalCaller* is defined in terms of formulas of the form $I \wedge \Box[N]_v$ where I is a state predicate, N is an action (called the *next-state action*), and v is a state function. A temporal formula is true or false on a behavior (an infinite sequence of states). Viewed as a temporal formula, a predicate I is true on a behavior iff I is true in the first state. The formula $\Box[N]_v$ is true of a behavior iff the action $[N]_v$, which is defined to equal $N \vee (v' = v)$, is true for every step (successive pair of states) in the behavior. Thus, $I \wedge \Box[N]_v$ asserts of a behavior that I is true in the first state and every step is an N step or leaves the value of v unchanged. Formula *LegalCaller* therefore asserts that, for every p in *PrIds*:

[5] We let a list of formulas bulleted with $\wedge$ or $\vee$ denote the conjunction or disjunction of the formulas, using indentation to eliminate parentheses. We also let $\Rightarrow$ have lower precedence than the other Boolean operators.

– The predicate $\neg Calling(p)$ is true in the initial state. In other words, initially there is no call in progress on channel $ch[p]$.

– Every step is either a $Call(p, a)$ step, for some a in $Args$, or else leaves $caller(p)$ unchanged. In other words, every step that changes the caller's part of the interface $ch[p]$ is a $Call(p, a)$ step with a legal argument a.

Formula *LegalCaller* specifies what it means for a caller to obey the two-phase handshake protocol. It specifies the values of $caller(p)$, for p in $PrIds$. More precisely, *LegalCaller* is a temporal formula whose semantic meaning is a predicate on behaviors that depends only on the values assigned by the states of a behavior to the state functions $caller(p)$ and $ch[p].rbit$. We interpret it as describing the possible values of $caller(p)$ as a function of the values of $ch[p].rbit$. Since we consider $caller(p)$ to represent the part of an interface controlled by the caller component, we consider *LegalCaller* to be the specification of a caller. However, the reader should not confuse this intuitive interpretation of *LegalCaller* with its formal semantics as a predicate on behaviors.

Formula *LegalReturner* is similar to *LegalCaller*. It asserts that, for every process p, every change to the returner's part of the interface $ch[p]$ is a $Return(p, v)$ step for some value v. It is our specification of what it means for a returner component to obey the handshake protocol. Formula *LegalReturner* has no initial predicate because we have arbitrarily assigned the initial condition on the channel to the caller's specification.[6] Unlike *LegalCaller*, which requires that the arguments be elements of $Args$, formula *LegalReturner* does not place any restriction on the results returned. This asymmetry arises because the specification problem involves syntactic restrictions on arguments, but not on results. A more realistic general-purpose interface specification would include as an additional parameter the set of legal results and would define *LegalReturner* to assert that results are in this set.

Composing a caller component and a returner component produces a system in which the two components interact according to the protocol. In TLA, composition is conjunction [3]. A simple calculation, using predicate logic and the fact that $\Box$ distributes over $\wedge$ and $\forall$, shows that $LegalCaller \wedge LegalReturner$ is equivalent to

$$\forall\, p \in PrIds \;:\; \begin{array}{l} \wedge\ \neg Calling(p) \\ \wedge\ \Box\left[\begin{array}{l} \vee\ \wedge\ \exists\, a \in Args : Call(p, a) \\ \quad\ \wedge\ rtrner(p)' = rtrner(p) \\ \vee\ \wedge\ \exists\, v : Return(p, v) \\ \quad\ \wedge\ caller(p)' = caller(p) \end{array}\right]_{\langle caller(p), rtrner(p) \rangle} \end{array}$$

This formula asserts that, for each process p, initially p is not processing a procedure call, and every step is either[7] (i) a $Call(p, a)$ step, for a legal argument

[6] Each component's specification would have had an initial condition on its signaling wire had we constrained the values of those wires—for example, by requiring signaling values to be 0 or 1.

[7] For any actions A and B, an $A \vee B$ step is an A step or a B step.

a, that leaves $rtrner(p)$ unchanged, (ii) a $Return(p, v)$ step that leaves $caller(p)$ unchanged, or (iii) a step that leaves both $caller(p)$ and $rtrner(p)$ unchanged. The conjunction of the specifications of the two components therefore expresses what we would expect to be the specification of the complete handshake protocol. (The conjunction represents two components communicating over the same channel because the specifications have the same free variable ch.)

We are using a noninterleaving representation [3], in which a single step can represent operations performed by several processes. This approach seems more convenient for this specification problem than the more traditional interleaving representation, in which each step represents an operation of at most one process. TLA is not inherently biased towards either specification style.

2 A Memory Component

In this section we give two specifications of the memory component described in the problem statement. In both specifications, the memory component supports read and write operations. The two specifications differ on whether the memory component is reliable; the unreliable version can return memory-failure exceptions, while the reliable version cannot.

2.1 The Parameters

For expository reasons, we split the specifications into two modules. The first module, *MemoryParameters*, is given in Figure 3 on this page. It declares the parameters of the memory specification. The **export** statement is explained below.

The **parameters** section declares the following parameters.

module *MemoryParameters*

export *MemoryParameters, E*

parameters

$MemLocs, MemVals, InitVal, Vals, PrIds$: CONSTANT
$memCh$: VARIABLE

---

assumption

$$ParamAssump \triangleq \land\ MemLocs \cup MemVals \subseteq Vals$$
$$\land\ \{\text{“BadArg”}, \text{“MemFailure”}\} \cap MemVals = \{\}$$
$$\land\ InitVal \in MemVals$$

$$LegalArgs \triangleq (\{\text{“Read”}\} \times Vals) \cup (\{\text{“Write”}\} \times Vals \times Vals)$$

include *ProcedureInterface* **as** E **with** $ch \leftarrow memCh, Args \leftarrow LegalArgs$

Fig. 3. Module *MemoryParameters*.

memCh This variable represents the procedure-calling interface to the memory.

MemLocs, *MemVals*, *InitVal* As in the problem statement, *MemLocs* is a set of memory locations, *MemVals* is a set of values that can be stored in those locations, and *InitVal* is the initial value of all locations.

Vals This is the set of syntactically legal argument values mentioned in the problem statement. In particular, we assume that the procedure-calling mechanism allows only read and write calls with arguments in *Vals*.

PrIds The same as for the *ProcedureInterface* module.

The module next asserts assumption *ParamAssump* about the constant parameters. The assumption's first conjunct states that *MemLocs* and *MemVals* are subsets of *Vals*, so every semantically legal argument is also syntactically legal. The second conjunct states that the strings "BadArg" and "MemFailure" are not elements of *MemVals*. These strings are used to represent the corresponding exceptions in the problem statement. For convenience, we let a successful read operation return a memory value and represent an exception by returning one of these strings. A successful write operation returns the string "OK". The third conjunct of *ParamAssump* asserts that *InitVal* is an element of *MemVals*, a condition implied by the problem statement.

The module defines *LegalArgs* to be the set of syntactically legal arguments of procedure calls to the memory.

Finally, the **include** statement includes a copy of the *ProcedureInterface* module, with each defined symbol X renamed as $E.X$, with *memCh* substituted for the parameter *ch*, with *LegalArgs* substituted for the parameter *Args*, and with *PrIds* (which is a parameter of the current module) implicitly substituted for the parameter *PrIds*. For example, this statement includes the definitions:

$$
\begin{array}{rcl}
E.caller(p) & \triangleq & \langle memCh[p].arg,\, memCh[p].cbit \rangle \\
E.LegalCaller & \triangleq & \forall\, p \in PrIds : \\
 & & \quad \land\ \neg E.Calling(p) \\
 & & \quad \land\ \Box[\exists\, a \in LegalArgs : E.Call(p, a)]_{E.caller(p)}
\end{array}
$$

The E in the **export** statement asserts that all these included definitions are exported. Exported symbols are the ones obtained by any other module that includes the *MemoryParameters* module. The *MemoryParameters* in the **export** statement asserts that the symbols defined in the module itself—in this case, *ParamAssump* and *LegalArgs*—are exported. Omitting an **export** statement in a module M is equivalent to adding the statement **export** M.

2.2 The Memory Specification

The specifications of the reliable and unreliable memories are contained in module *Memory* of Figure 4 on the next page and Figure 5 on page 30. The module begins by importing the *MemoryParameters* module. The **import** statement is

module *Memory*

import *MemoryParameters*
export *Memory, E*

module *Inner*

parameters
$mem, result$: VARIABLE

$$
\begin{array}{l}
NotAResult \;\triangleq\; \text{CHOOSE } v : \\
\qquad\qquad v \notin \{\text{“OK”}, \text{“BadArg”}, \text{“MemFailure”}\} \cup MemVals \\
MInit(l) \;\triangleq\; mem[l] = InitVal \\
PInit(p) \;\triangleq\; result[p] = NotAResult \\
Read(p) \;\triangleq\; \exists\, l : \land\; E.Calling(p) \\
\qquad\qquad\quad \land\; memCh[p].arg = \langle \text{“Read”}, l \rangle \\
\qquad\qquad\quad \land\; result'[p] = \textbf{if } l \in MemLocs \textbf{ then } mem[l] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \text{“BadArg”} \\
\qquad\qquad\quad \land\; \text{UNCHANGED } E.rtrner(p) \\
Write(p, l) \;\triangleq\; \exists\, v : \land\; E.Calling(p) \\
\qquad\qquad\qquad \land\; memCh[p].arg = \langle \text{“Write”}, l, v \rangle \\
\qquad\qquad\qquad \land\; \lor\; \land\; (l \in MemLocs) \land (v \in MemVals) \\
\qquad\qquad\qquad\qquad\quad \land\; mem'[l] = v \\
\qquad\qquad\qquad\qquad\quad \land\; result'[p] = \text{“OK”} \\
\qquad\qquad\qquad\quad\;\; \lor\; \land\; \neg((l \in MemLocs) \land (v \in MemVals)) \\
\qquad\qquad\qquad\qquad\quad \land\; result'[p] = \text{“BadArg”} \\
\qquad\qquad\qquad\qquad\quad \land\; \text{UNCHANGED } mem[l] \\
\qquad\qquad\qquad \land\; \text{UNCHANGED } E.rtrner(p) \\
Fail(p) \;\triangleq\; \land\; E.Calling(p) \\
\qquad\qquad\;\; \land\; result'[p] = \text{“MemFailure”} \\
\qquad\qquad\;\; \land\; \text{UNCHANGED } E.rtrner(p) \\
Return(p) \;\triangleq\; \land\; result[p] \neq NotAResult \\
\qquad\qquad\quad\;\; \land\; result'[p] = NotAResult \\
\qquad\qquad\quad\;\; \land\; E.Return(p, result[p]) \\
RNext(p) \;\triangleq\; Read(p) \lor (\exists\, l : Write(p, l)) \lor Return(p) \\
UNext(p) \;\triangleq\; RNext(p) \lor Fail(p) \\
pvars(p) \;\triangleq\; \langle E.rtrner(p), result[p] \rangle \\
RPSpec(p) \;\triangleq\; \land\; PInit(p) \\
\qquad\qquad\quad\;\; \land\; \Box[RNext(p)]_{pvars(p)} \\
\qquad\qquad\quad\;\; \land\; \mathrm{WF}_{pvars(p)}(RNext(p)) \;\land\; \mathrm{WF}_{pvars(p)}(Return(p))
\end{array}
$$

Fig. 4. First part of module *Memory*.

$$
\begin{aligned}
\mathit{UPSpec}(p) \;&\triangleq\; \land\ \mathit{PInit}(p)\\
&\quad\ \land\ \Box[\mathit{UNext}(p)]_{\mathit{pvars}(p)}\\
&\quad\ \land\ \mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p)) \land \mathrm{WF}_{\mathit{pvars}(p)}(\mathit{Return}(p))
\end{aligned}
$$

$$\mathit{MSpec}(l) \;\triangleq\; \mathit{MInit}(l) \land \Box[\exists\, p \in \mathit{PrIds} : \mathit{Write}(p, l)]_{\mathit{mem}[l]}$$

$$
\begin{aligned}
\mathit{IRSpec} \;&\triangleq\; (\forall\, p \in \mathit{PrIds} : \mathit{RPSpec}(p)) \land (\forall\, l \in \mathit{MemLocs} : \mathit{MSpec}(l))\\
\mathit{IUSpec} \;&\triangleq\; (\forall\, p \in \mathit{PrIds} : \mathit{UPSpec}(p)) \land (\forall\, l \in \mathit{MemLocs} : \mathit{MSpec}(l))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{RSpec} \;&\triangleq\; \boldsymbol{\exists}\, \mathit{mem}, \mathit{result} : \mathit{Inner.IRSpec}\\
\mathit{USpec} \;&\triangleq\; \boldsymbol{\exists}\, \mathit{mem}, \mathit{result} : \mathit{Inner.IUSpec}
\end{aligned}
$$

Fig. 5. Second part of module *Memory*.

equivalent to simply copying the entire *Memory* module—its parameter declarations, assumption, and definitions—into the current module. (However, imported definitions are not automatically exported.) The **export** statement is needed because we want to use formula *E.LegalCaller* in asserting the correctness of an implementation.

The reader should note the distinction between **import** and **include**. Importing a module imports its definitions and parameters. Including a module includes its definitions (with renaming), but its parameters are instantiated, not included.

We now explain our specification in a top-down fashion, starting with the final definition. Our specifications of the reliable and unreliable memory components are the formulas *RSpec* and *USpec* defined in Figure 5, at the end of the module. The two specifications are almost identical, so we now discuss only *RSpec*, the reliable-memory specification. Afterwards, we explain how *USpec* differs from it.

Formula *RSpec* is defined to equal $\boldsymbol{\exists}\, \mathit{mem}, \mathit{result} : \mathit{Inner.IRSpec}$. Intuitively, it asserts of a behavior that there exist assignments of values for the variables *mem* and *result*—possibly assigning different values in each state of the behavior—for which the behavior satisfies *Inner.IRSpec*. Formula $\boldsymbol{\exists}\, \mathit{mem}, \mathit{result} : \mathit{Inner.IRSpec}$ asserts nothing about the actual values of the variables *mem* and *result*; it is the specification obtained by "hiding" *mem* and *result* in the specification *Inner.IRSpec*.

Since *mem* and *result* are not free variables of the specification, they should not be parameters of module *Memory*. We therefore introduce a submodule named *Inner* having these variables as its parameters.[8] The symbol *IRSpec* defined in submodule *Inner* is named *Inner.IRSpec* when used outside the submodule. The symbol *Inner.IRSpec* can appear only in a context in which *mem* and *result* are declared—for example, in the scope of the quantifier $\boldsymbol{\exists}\, \mathit{mem}, \mathit{result}$.

[8] Instead of introducing a submodule, we could have made *mem* and *result* explicit parameters of all the definitions in which they now occur free.

The bound variable mem represents the current contents of memory; $mem[l]$ equals the contents of memory location l. The bound variable $result$ records the activity of the memory component processes. For each process p, $result[p]$ initially equals $NotAResult$, which is a value different from any that a procedure call can return.[9] When process p is ready to return a result, that result is $result[p]$. (Even though it is ready to return, the process can "change its mind" and choose a different result before actually returning.)

Formula $IRSpec$ is the conjunction of two formulas, which describe two components that constitute the memory component. The first component is responsible for communicating on the channel $memCh$ and managing the variable $result$; the second component manages the variable mem.

The second conjunct is itself the conjunction[10] of formulas $MSpec(l)$, for each memory location l. We view $MSpec(l)$ as the specification of a separate process that manages $mem[l]$. Formula $MSpec(l)$ has the familiar form $I \wedge \Box[N]_v$. It asserts that $MInit(l)$ holds in the initial state, and that every step is either a $Write(p, l)$ step for some process p, or else leaves $mem[l]$ unchanged. The initial predicate $MInit(l)$ asserts that $mem[l]$ equals $InitVal$, the initial memory value. Action $Write(p, l)$, which we discuss below, is enabled only when the memory component is processing a procedure call by process p to write some value v to location l; a $Write(p, l)$ step sets the new value of $mem[l]$ to this v.

The first conjunct of $IRSpec$ is the conjunction of formulas $RPSpec(p)$ for each process p in $PrIds$. We view $RPSpec(p)$ as the specification of a process that manages the returner's part of the channel $memCh[p]$ and the variable $result[p]$. Formula $RPSpec(p)$ has the form $I \wedge \Box[N]_v \wedge F$. A formula $\Box[N]_v$ asserts that every step that changes v is an N step, but it does not require any such steps to occur. It allows a behavior in which v never changes. We require that certain changes do occur by conjoining an additional condition F, which constrains what must eventually happen but does not disallow any individual step. We call $I \wedge \Box[N]_v$ the *safety* condition of the specification and F its *fairness* or *liveness* condition. We now examine the safety condition of $RPSpec(p)$; its fairness condition is considered below.

A formula $I \wedge \Box[N]_v$ describes how the state function v may change. For $RPSpec(p)$, the subscript v is the state function $pvars(p)$, which is defined to be the pair $\langle E.rtrner(p), result[p] \rangle$. A pair changes iff one of its elements changes, so $RPSpec(p)$ describes changes to $E.rtrner(p)$, the returner's part of the communication channel $memCh[p]$, and to $result[p]$.

We explain $RPSpec(p)$ with the help of the predicate-action diagram [13] of Figure 6 on the next page. This diagram has the following meaning.

– The small arrow indicates that initially, $result[p]$ equals $NotAResult$.

[9] The definition of $NotAResult$ in submodule $Inner$ uses the operator CHOOSE, which is the TLA$^+$ name for Hilbert's ε [15]. We can define $NotAResult$ in this way because the axioms of set theory imply that, for every set S, there exists a value not in S.

[10] Informally, we often think of $\forall\, x \in S : F(x)$ as the conjunction of the formulas $F(x)$ for all x in S.

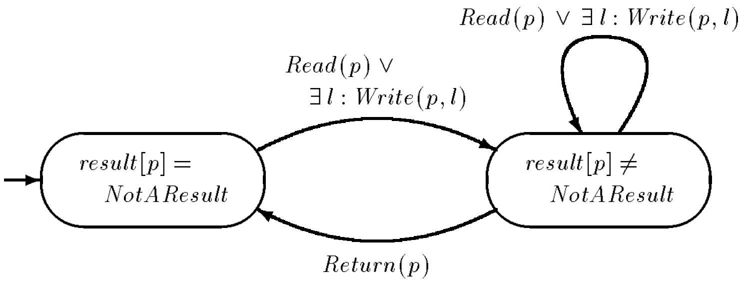


Fig. 6. A predicate-action diagram of $pvars(p)$ for formula $RPSpec(p)$ of the *Memory* module.

- When $result[p]$ equals $NotAResult$, the pair $pvars(p)$ can be changed only by a $Read(p)$ step or a $Write(p, l)$ step, for some l. Such a step sets $result[p]$ unequal to $NotAResult$.
- When $result[p]$ is not equal to $NotAResult$, the pair $pvars(p)$ can be changed only by a $Read(p)$ step, some $Write(p, l)$ step, or a $Return(p)$ step. A $Read(p)$ or $Write(p, l)$ step leaves $result[p]$ unequal to $NotAResult$, while a $Return(p)$ step sets it to $NotAResult$.

Predicate-action diagrams are defined formally in [13] to represent TLA formulas. The assertion that Figure 6 is a diagram for $RPSpec(p)$ means that $RPSpec(p)$ implies the formula represented by the diagram. In general, one can draw many different diagrams for the same formula. Proving that a diagram is a predicate-action diagram for a specification helps confirm our understanding of the specification. The proof for Figure 6 is trivial. This diagram is actually equivalent to the safety part of $RPSpec(p)$.

To complete our understanding of the safety part of $RPSpec(p)$, we must examine what steps are allowed by the actions $Read(p)$, $Write(p, l)$, and $Return(p)$. Action $Read(p)$ is enabled when $E.Calling(p)$ is true and $memCh[p].arg$ equals $\langle$ "Read", $l \rangle$ for some l, so the process p caller has called the read procedure with argument l and the process p returner has not yet returned a result. If l is a legal memory address, then a $Read(p)$ step sets $result[p]$ to $mem[l]$; otherwise it sets $result[p]$ to the string "BadArg". The step leaves $E.rtrner$ unchanged. (The TLA$^+$ action UNCHANGED v is defined to equal $v' = v$, for any state function v.) Action $Write(p, l)$ is similar. It is enabled when there is a pending request to write some value v in memory location l; it sets $result[p]$ to the appropriate result and sets $mem[l]$ to v iff the request is valid.

Action $Return(p)$ issues the return of $result[p]$ and resets $result[p]$ to equal $NotAResult$. The action is enabled iff $result[p]$ is unequal to $NotAResult$ and action $E.Return(p)$ is enabled, which is the case iff $E.Calling(p)$ equals TRUE.

Looking at Figure 6 again, we now see that returner process p goes through the following cycle. It waits (with $result[p]$ equal to $NotAResult$) until a procedure call occurs. It then does one or more internal $Read(p)$ or $Write(p, l)$

steps, which choose $\mathit{result}[p]$. Finally, it returns $\mathit{result}[p]$ and resets $\mathit{result}[p]$ to $\mathit{NotAResult}$. Allowing multiple $\mathit{Write}(p, l)$ steps is important because $\mathit{mem}[l]$ could be changed between those steps by $\mathit{Write}(q, l)$ steps for some q different from p. Such $\mathit{Write}(q, l)$ steps are allowed by Figure 6 (and by $\mathit{RPSpec}(p)$) if they do not change $\mathit{pvars}(p)$. It makes no difference to the final specification RSpec whether or not multiple $\mathit{Read}(p)$ steps are allowed. The changes to memCh are the same as if only the last one were performed, and memCh is the only free variable of RSpec. Allowing multiple $\mathit{Read}(p)$ steps simplifies the specification a bit.

This completes our explanation of the safety condition of $\mathit{RPSpec}(p)$. We now consider the fairness condition. The safety condition of $\mathit{RPSpec}(p)$ implies that the steps described by Figure 6 are the only ones that are allowed to happen. We want the fairness condition to assert that they must happen. In particular, we want to assert the following two requirements: (L1) after a procedure call has been issued, the transition out of the $\mathit{result}[p] = \mathit{NotAResult}$ state eventually occurs, and (L2) the transition back to that state eventually occurs.

These requirements are expressed with *weak fairness* formulas of the form $\mathrm{WF}_v(A)$. Such a formula asserts that if the action $A \wedge (v' \neq v)$ remains continually enabled, then an $A \wedge (v' \neq v)$ step must occur. In other words, if it remains possible to take an A step that changes v, then such a step must eventually be taken.

Condition L1 is implied by $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p))$. To see this, suppose that $\mathit{result}[p]$ equals $\mathit{NotAResult}$ and a read or write call is issued. Then $\mathit{Read}(p)$ or some $\mathit{Write}(p, l)$ action is enabled, so $\mathit{RNext}(p)$ is enabled. Assuming that the caller obeys the handshake protocol, action $\mathit{RNext}(p)$ will remain enabled until a $\mathit{Read}(p)$ or $\mathit{Write}(p, l)$ step occurs. Formula $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p))$ therefore implies that a $\mathit{RNext}(p)$ step does occur, and that step can only be the desired $\mathit{Read}(p)$ or $\mathit{Write}(p, l)$ step.

Formula $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p))$ implies that, while $\mathit{result}[p] \neq \mathit{NotAResult}$ remains true, $\mathit{RNext}(p)$ steps must keep occurring. However, those steps could be $\mathit{Read}(p)$ or $\mathit{Write}(p, l)$ steps. ($\mathit{Read}(p)$ steps can change $\mathit{pvars}(p)$ if intervening steps by other processes keep changing $\mathit{mem}[l]$.) Formula $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{Return}(p))$, the second conjunct of $\mathit{RPSpec}(p)$'s fairness condition, asserts that a $\mathit{Return}(p)$ step must eventually occur when $\mathit{result}[p] \neq \mathit{NotAResult}$ holds.

There are other possible fairness conditions for $\mathit{RPSpec}(p)$. Two other obvious choices are obtained by replacing $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p))$ with one of the following:

$$\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{Read}(p)) \;\wedge\; \mathrm{WF}_{\mathit{pvars}(p)}(\exists\, l \,:\, \mathit{Write}(p, l))$$
$$\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{Read}(p)) \;\wedge\; (\forall\, l \,:\, \mathrm{WF}_{\mathit{pvars}(p)}(\mathit{Write}(p, l)))$$

It is not hard to check that the conjunction of $E.\mathit{LegalCaller}$ (the specification that the caller obeys the handshake protocol) and the safety condition of $\mathit{RPSpec}(p)$ implies that both formulas are equivalent to $\mathrm{WF}_{\mathit{pvars}(p)}(\mathit{RNext}(p))$, for any p in PrIds. We care what the memory component does only when the caller obeys the protocol. Hence, any of these three choices of fairness conditions for $RPSpec(p)$ yield essentially the same specification. (The three fairness conditions need not be equivalent on a behavior in which $memCh[p].arg$ changes while the memory is processing a procedure call by process p.)

Weak fairness is a standard concept of concurrency [6, 17]. The reader who is not already familiar with it may find fairness conditions difficult to understand. Fairness can be subtle, and it is not obvious why we express it in TLA with WF formulas. For example, it might seem easier to express L1 by writing the temporal-logic formula

$$(result[p] = NotAResult) \wedge E.Calling(p) \leadsto (result[p] \neq NotAResult)$$

which asserts that if $result[p]$ ever equals $NotAResult$ when $E.Calling(p)$ is true, then it must eventually become unequal to $NotAResult$. We have found that the use of arbitrary temporal-logic formulas makes it easy to write incorrect specifications, and using WF formulas helps us avoid errors.

Finally, let us consider the specification $USpec$ of the unreliable memory component. It is identical to $RSpec$ except it has action $UNext(p)$ instead of $RNext(p)$ as its next-state action. Action $UNext(p)$ differs from $RNext(p)$ by also allowing internal $Fail(p)$ steps, which set $result[p]$ to "MemFailure". Such steps can occur instead of, before, after, or between $Read(p)$ or $Write(p, l)$ steps. We could have replaced $RNext(p)$ with $UNext(p)$ in the fairness condition; $LegalCaller$ implies that the two definitions of $USpec$ are equivalent. However, it might seem odd to require the eventual occurrence of a step that may be a failure step.

2.3 Solution to Problem 1

(a) Formulas $RSpec$ and $USpec$ are what we call *component specifications.* They describe a system containing a properly operating (reliable or unreliable) memory component. Whether they constitute the specifications of a memory depends on what the specifications are for.

Component specifications can be used to describe a complete system in which all the components function properly, allowing us to prove properties of the system. The simplest such complete-system specification of a system containing a reliable memory component is $RSpec \wedge E.LegalCaller$, which asserts that the memory component behaves properly and the rest of the system follows the handshake protocol.

Another possible use of a memory specification is to serve as a contract between the user of the memory and its implementor. Such a specification should be satisfied by precisely those behaviors that represent physical histories in which the memory fulfills its obligations. Formula $RSpec$ cannot serve as such a specification because it says nothing about the memory's environment. A real memory that uses the two-phase handshake protocol will display completely unpredictable behavior if its environment does not correctly follow the protocol. To be implementable, the specification must assert only that $RSpec$ is satisfied if the memory's environment satisfies the caller's part of the handshake protocol—in other words, if $E.LegalCaller$ is satisfied. We might therefore expect the specification of a reliable memory to be $E.LegalCaller \Rightarrow RSpec$. However, for reasons explained in [3], we instead write this specification as the formula $E.LegalCaller \stackrel{+}{\rhd} RSpec$. This formula means roughly that $RSpec$ remains true as long as $E.LegalCaller$ does. Such a formula is called an *assumption/guarantee specification* [8]; the memory guarantees to satisfy its component-specification $RSpec$ as long as the environment assumption $E.LegalCaller$ is satisfied.

When we present a component specification as a solution to one of the specification problems, we indicate its environment assumption. Writing the corresponding assumption/guarantee specification is then trivial.

When we write a component specification, we think of steps satisfying the specification's next-state action as representing operations performed by that component. We could make this an explicit assumption by formally attributing every step to either the component or its environment, as described in [3]. However, whether the component or its environment actually performs an operation is a question of physical reality, and the connection between a mathematical specification and reality can never be made completely formal.

The assumption $ParamAssump$ about the parameters is not part of our memory component specifications, since the formulas $RSpec$ and $USpec$ are not defined in terms of $ParamAssump$. We could weaken the specifications by adding $ParamAssump$ as an assumption and writing, for example, $ParamAssump \Rightarrow RSpec$. We do not need to do so; as we will see below, putting the assumption $ParamAssump$ into the $Memory$ module allows us to use it when proving the correctness of an implementation of the memory component.

(b) In TLA, implementation is implication. To prove that a reliable memory implements an unreliable one, it suffices to prove the theorem $RSpec \Rightarrow USpec$. The proof is easy; expanding the definitions shows that it suffices to prove $\Box[RNext(p)]_{pvars(p)} \Rightarrow \Box[UNext(p)]_{pvars(p)}$, which is trivial since $RNext(p)$ obviously implies $UNext(p)$.

In general, we would not expect such an implication to be valid. For example, it would not have been valid had we written $\mathrm{WF}_{pvars(p)}(UNext(p))$ instead of $\mathrm{WF}_{pvars(p)}(RNext(p))$ in the fairness condition of $UPSpec(p)$. Component specifications like $RSpec$ and $USpec$ describe how the component should behave when its environment behaves properly. They do not constrain the environment's behavior, and they may allow bizarre behaviors when the environment behaves improperly. A priori, there is no reason why the particular bizarre behaviors allowed by $RSpec$ as the result of an incorrectly functioning environment should also be allowed by $USpec$. Hence, we would expect $RSpec \Rightarrow USpec$ to be true only for those behaviors satisfying the memory's environment specification, $E.LegalCaller$. We would therefore expect to prove only $E.LegalCaller \Rightarrow (RSpec \Rightarrow USpec)$, which is equivalent to

$$E.LegalCaller \land RSpec \Rightarrow USpec \tag{1}$$

We can also phrase implementation in terms of assumption/guarantee specifications. Such specifications are satisfied by precisely those behaviors in which the memory meets its obligation. We would expect the assumption/guarantee specification of the reliable memory to imply that of the unreliable memory:

$$(E.LegalCaller \stackrel{+}{\rhd} RSpec) \Rightarrow (E.LegalCaller \stackrel{+}{\rhd} USpec) \tag{2}$$

The relation between the two forms of implementation conditions exemplified by (1) and (2) is investigated in [3]. Because our two memory-component specifications are so similar, we can prove $RSpec \Rightarrow USpec$, which implies (1) and (2).

(c) If the memory is implemented with unreliable components that can fail forever, then there is no way to guarantee that anything but "MemFailure" exceptions will ever occur. (For example, this will be the case if it is implemented with an RPC component that always returns "RPCFailure" exceptions.)

We can easily define a memory that guarantees eventual success. We do so by requiring that if enough calls of some particular kind are issued, then one of them eventually succeeds. Different conditions are obtained by different choices of the kind of calls—for example, calls to a particular memory location, or reads by a particular process. Such conditions can be expressed using *strong fairness* formulas of the form $\mathrm{SF}_v(A)$. This formula asserts that if $A \wedge (v' \neq v)$ is enabled often enough, then an $A \wedge (v' \neq v)$ step must occur. (Strong fairness is stronger than weak fairness because it requires a step to occur if the action is enabled often enough, even if the action does not remain continuously enabled.) For example, to strengthen the specification to require that, if process p keeps issuing calls, then it will eventually receive a result other than "MemFailure", we simply replace the fairness condition of $UPSpec(p)$ by:

$$\begin{aligned} &\wedge\ \mathrm{SF}_{pvars(p)}(Read(p) \vee (\exists\, l \,:\, Write(p, l))) \\ &\wedge\ \mathrm{SF}_{pvars(p)}(Return(p) \wedge (result[p] \neq \text{"MemFailure"})) \end{aligned}$$

To solve Problem 3 (proving the correctness of a memory implementation), we would then need to add a corresponding liveness condition to the RPC component.

3 Implementing the Memory

The memory implementation is obtained by composing a memory clerk component, an RPC component, and a reliable memory component. The memory clerk translates memory calls into RPC calls, and optionally retries RPC calls when they result in RPC failures. In this section we describe the RPC and the memory clerk components, and then prove the correctness of the implementation.

3.1 The RPC Component

The RPC component connects a sender to a receiver. As with the memory component, we split its specification into two modules.

The Parameters Module The specification of the RPC component begins with module *RPCParameters* in Figure 7 on this page; the module declares parameters and both makes and includes some definitions. The *RPCParameters*

module *RPCParameters*

export *RPCParameters, Snd, Rcv*
import *Naturals, Sequences*

parameters $sndCh, rcvCh$: VARIABLE
$\quad Procs, ArgNum, Vals, PrIds$: CONSTANT

---

assumption $ParamAssump \triangleq ArgNum \in [Procs \rightarrow Nat]$

$LegalSndArgs \triangleq \{\text{“RemoteCall”}\} \times \text{STRING} \times Seq(Vals)$

$LegalRcvArgs \triangleq$

$$\{s \in Seq(Procs \cup Vals) : \begin{array}{l} \wedge\ Len(s) > 0 \\ \wedge\ Head(s) \in Procs \\ \wedge\ Tail(s) \in Seq(Vals) \\ \wedge\ Len(s) = 1 + ArgNum[Head(s)]\ \} \end{array}$$

include *ProcedureInterface* **as** *Snd* **with** $ch \leftarrow sndCh, Args \leftarrow LegalSndArgs$
include *ProcedureInterface* **as** *Rcv* **with** $ch \leftarrow rcvCh, Args \leftarrow LegalRcvArgs$

Fig. 7. Module *RPCParameters*.

module imports the module *Naturals*, a predefined module that defines the natural numbers and the usual operators on them. It then imports the *Sequences* module, which defines operators on finite sequences. In TLA$^+$, an n-tuple $\langle v_1, \ldots, v_n \rangle$ is a function whose domain is the set $\{1, \ldots, n\}$ of natural numbers, where $\langle v_1, \ldots, v_n \rangle[i]$ equals v_i, for $1 \leq i \leq n$.[11] The *Sequences* module represents sequences as tuples. The module appeared in [14] (without the definition of *Seq*, which was not needed there) and is given without further explanation in Figure 8 on the next page. It defines the usual operators *Head*, *Tail*, $\circ$ (concatenation), and *Len* (length) on sequences, as well as the operator *Seq*, where $Seq(S)$ is the set of sequences of elements in S. (The values of $Head(s)$ and $Tail(s)$ are not constrained when s is the empty sequence.)

The parameters declared in module *RPCParameters* have the following interpretations:

sndCh The procedure-calling interface between the sender and the RPC component.

[11] TLA$^+$ uses square brackets to denote function application. An “array variable” is just a variable whose value is a function.

rcvCh The procedure-calling interface between the RPC component and the receiver.

Procs, *ArgNum* As in the problem statement, *Procs* is a set of legal procedure names and *ArgNum* is a function that assigns to each legal procedure name its number of arguments.

Vals The set of all possible syntactically valid arguments.

PrIds The same as for the *ProcedureInterface* module.

Assumption *ParamAssump* asserts that *ArgNum* is a function with domain *Procs* and range a subset of the set *Nat* of natural numbers. (The definition of *Nat* comes from the *Naturals* module.)

The module next defines *LegalSndArgs* to be the set of syntactically valid arguments with which the RPC component can be called. Calls to the RPC component take two arguments, the first of which is an element of STRING, the set of all strings, and the second of which is a sequence of elements in *Vals*. We use the same convention as in the memory specification, that the argument of a procedure call is a tuple consisting of the procedure name followed by its arguments. The RPC component has a single procedure, whose name is "RemoteCall".

The module defines *LegalRcvArgs* to be the set of syntactically valid arguments with which the RPC component can call the receiver. These consist of all tuples of the form $\langle p, v_1, \ldots, v_n \rangle$ with p in *Procs*, the v_i in *Vals*, and n equal to $ArgNum[p]$.

Finally, the module includes two copies of the *ProcedureInterface* module, one for each of the interfaces, with the appropriate instantiations. The **export** statement (at the beginning of the module) exports these included definitions.

module *Sequences*

import *Naturals*

$$
\begin{array}{lcl}
OneTo(n) & \triangleq & \{i \in Nat : (1 \leq i) \land (i \leq n)\} \\
Seq(S) & \triangleq & \text{UNION } \{[OneTo(n) \rightarrow S] : n \in Nat\} \\
Len(s) & \triangleq & \text{CHOOSE } n : (n \in Nat) \land ((\text{DOMAIN } s) = OneTo(n)) \\
Head(s) & \triangleq & s[1] \\
Tail(s) & \triangleq & [i \in OneTo(Len(s) - 1) \mapsto s[i + 1]] \\
(s) \circ (t) & \triangleq & [i \in OneTo(Len(s) + Len(t)) \mapsto \textbf{if } i \leq Len(s) \\
 & & \qquad \textbf{then } s[i] \\
 & & \qquad \textbf{else } t[i - Len(s)]\ ]
\end{array}
$$

Fig. 8. Module *Sequences*.

Problem 2: The RPC Component's Specification The specification of the RPC component appears in module *RPC* of Figure 10 on page 40. It is the formula $\exists\, rstate : Inner.ISpec$, where *ISpec* is defined in a submodule named *Inner*.

We explain the specification *ISpec* with the aid of the diagram of Figure 9 on this page. This is a predicate-action diagram for *ISpec* of all changes to

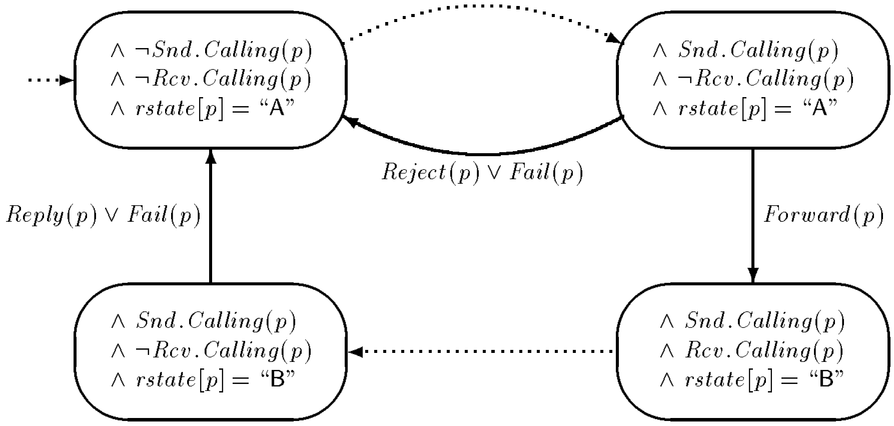


Fig. 9. A predicate-action diagram of $vars(p)$ for formula *ISpec* of module *RPC*, where p is an element of *PrIds*. (The dotted arrows are not formally part of the diagram.)

$vars(p)$, where p is any element of *PrIds*. The state function $vars(p)$ is the triple $\langle rstate[p], Snd.rtrner(p), Rcv.caller(p)\rangle$ that forms the state of the RPC component's process p. The dotted arrows are not formally part of the diagram. The initial-condition arrow indicates obligations of both the RPC component and its environment; the other dotted arrows represent state changes caused by the environment that do not change the RPC component's state. (Recall that $\ldots Calling(p)$ can be changed by either the caller changing $\ldots caller(p)$ or the returner changing $\ldots rtrner(p)$.) The top dotted arrow represents the sender's action of calling the RPC component, which makes $Snd.Calling(p)$ true. The bottom dotted arrow represents the receiver's return action, which makes $Rcv.Calling$ false.

The solid arrows (the real arrows of the predicate-action diagram) represent steps of process p of the RPC component. The $Forward(p)$ action relays the call to the receiver, making $Rcv.Calling(p)$ true. The $Fail(p)$ action returns "RPCFailure". The $Reject(p)$ action returns "BadCall" without relaying the request. The $Reply(p)$ action returns to the sender the result returned by the receiver. The variable *rstate* is needed to distinguish the upper right and lower-left states. The values "A" and "B" are arbitrary; any two values can be used.

The specification *Spec* of the RPC component appears in module *RPC* of

Figure 10 on this page. It is similar enough to the memory specification that it

module *RPC*

import *RPCParameters, Naturals, Sequences*
export *RPC, Snd, Rcv*

module *Inner*

parameters
$rstate$: VARIABLE

$$
\begin{aligned}
Init(p) &\triangleq (rstate[p] = \text{“A”}) \land \neg Rcv.Calling(p)\\
RelayArg(p) &\triangleq \langle sndCh[p].arg[2]\rangle \circ sndCh[p].arg[3]\\
Forward(p) &\triangleq \land\ Snd.Calling(p) \land (rstate[p] = \text{“A”})\\
&\qquad \land\ RelayArg(p) \in LegalRcvArgs\\
&\qquad \land\ Rcv.Call(p, RelayArg(p))\\
&\qquad \land\ rstate'[p] = \text{“B”}\\
&\qquad \land\ \text{UNCHANGED } Snd.rtrner(p)\\
Reject(p) &\triangleq \land\ rstate[p] = \text{“A”}\\
&\qquad \land\ RelayArg(p) \notin LegalRcvArgs\\
&\qquad \land\ Snd.Return(p, \text{“BadCall”})\\
&\qquad \land\ \text{UNCHANGED } \langle rstate[p], Rcv.caller(p)\rangle\\
Fail(p) &\triangleq \land\ \neg Rcv.Calling(p)\\
&\qquad \land\ Snd.Return(p, \text{“RPCFailure”})\\
&\qquad \land\ rstate'[p] = \text{“A”}\\
&\qquad \land\ \text{UNCHANGED } Rcv.caller(p)\\
Reply(p) &\triangleq \land\ \neg Rcv.Calling(p) \land (rstate[p] = \text{“B”})\\
&\qquad \land\ Snd.Return(p, rcvCh[p].res)\\
&\qquad \land\ rstate'[p] = \text{“A”}\\
&\qquad \land\ \text{UNCHANGED } Rcv.caller(p)\\
Next(p) &\triangleq Forward(p) \lor Reject(p) \lor Fail(p) \lor Reply(p)\\
vars(p) &\triangleq \langle rstate[p], Snd.rtrner(p), Rcv.caller(p)\rangle\\
ISpec &\triangleq \forall\, p \in PrIds : Init(p) \land \Box[Next(p)]_{vars(p)} \land \mathrm{WF}_{vars(p)}(Next(p))
\end{aligned}
$$

$$Spec \triangleq \exists\, rstate : Inner.ISpec$$

Fig. 10. The specification of the RPC component.

should require little additional explanation. The definition of *RelayArg* makes use of the way sequences are represented as tuples, and it may seem a little obscure. When the sender's process p has called the RPC component, $RelayArg(p)$ is the argument with which the RPC component should call the receiver. For example, if the RPC component is called with argument $\langle$ "RemoteCall", "Write", $\langle 17, \sqrt{2}\rangle\rangle$, then $RelayArg(p)$ equals $\langle$ "Write", $17, \sqrt{2}\rangle$. In the definitions of the actions, we have eliminated some redundant instances of the conjuncts $Snd.Calling(p)$ and $\neg Rcv.Calling(p)$ that appear in the predicate-action diagram; $Snd.Calling(p)$ is implied by $Snd.Return(p, \ldots)$, and the diagram shows that $\neg Rcv.Calling(p)$ is implied by $rstate[p] =$ "A" in every reachable state.

Formula *Spec* of the *RPC* module is the component specification of the RPC component. The component's environment specification is $Snd.LegalCaller \wedge Rcv.LegalReturner$. As described above, the conjunction of these two formulas is the specification of a complete system consisting of an RPC component and a sender and receiver that obey the handshake protocol; combining the formulas with the $\stackrel{+}{\triangleright}$ operator yields an assumption/guarantee specification of the *RPC* component.

3.2 The Implementation

The Memory Clerk We now present the specification of the memory clerk, which is quite similar to that of the RPC component. It begins with module *MemClerkParameters* of Figure 11 on this page. The module declares the following parameters:

sndCh, *rcvCh* The procedure-calling interfaces between the clerk and the memory's caller, and between the clerk and the RPC component.

Vals, *PrIds* The same as for the *MemoryParameters* and *ProcedureInterface* modules, respectively.

The definitions of *LegalSndArgs* and *LegalRcvArgs* and the inclusion of two copies of the *ProcedureInterface* module serve the same purpose as they do in the *RPCParameters* module.

module *MemClerkParameters*

export *MemClerkParameters, Snd, Rcv*

parameters $sndCh, rcvCh$: VARIABLE
$PrIds, Vals$: CONSTANT

$$LegalSndArgs \triangleq (\{\text{"Read"}\} \times Vals) \cup (\{\text{"Write"}\} \times Vals \times Vals)$$

$$LegalRcvArgs \triangleq \{\text{"RemoteCall"}\} \times \{\text{"Read"}, \text{"Write"}\} \times Seq(Vals)$$

include *ProcedureInterface* **as** *Snd* **with** $ch \leftarrow sndCh, Args \leftarrow LegalSndArgs$
include *ProcedureInterface* **as** *Rcv* **with** $ch \leftarrow rcvCh, Args \leftarrow LegalRcvArgs$

Fig. 11. Module *MemClerkParameters*.

The specification of the memory clerk is a formula $\exists\, cstate : Inner.ISpec$. The formula *ISpec* is described by the predicate-action diagram of Figure 12 on this page, which is similar to that of Figure 9 (page 39). The $Reply(p)$ and

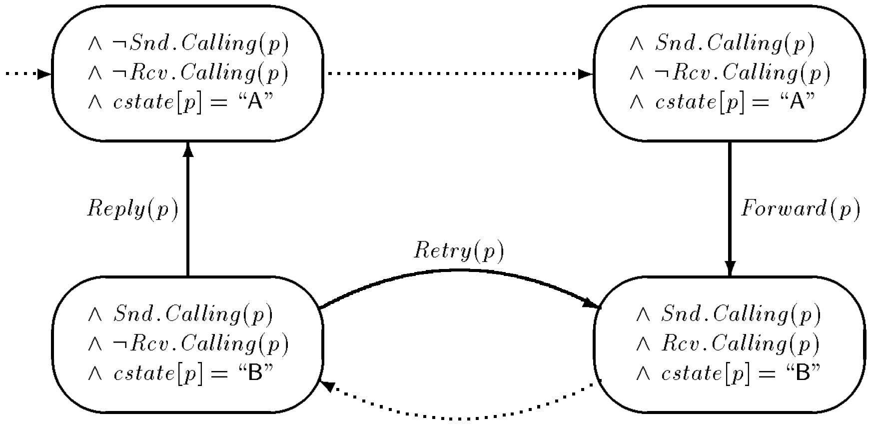


Fig. 12. A predicate-action diagram of $vars(p)$ for formula *ISpec* of module *MemClerk*, where p is an element of *PrIds*. (The dotted arrows are not formally part of the diagram.)

$Forward(p)$ actions play the same role as in the RPC component's specification. Action $Retry(p)$ retries an RPC call that has yielded an RPC failure.

The clerk's specification *Spec* appears in Module *MemClerk* of Figure 13 on the next page. The safety part can be deduced from the predicate-action diagram as we did for the RPC component. The liveness part is a bit trickier. We want to require that the clerk eventually returns from a call, assuming the RPC component eventually returns from each call. Weak fairness on the $Forward(p)$ action ensures progress from the upper-right to the lower-right state of the predicate-action diagram. Strong fairness of $Reply(p)$ is required to ensure eventual progress from the lower-left to the upper-left state; weak fairness would allow behaviors in which the clerk keeps performing $Retry(p)$ steps without ever performing a $Reply(p)$ step.

The Implementation Proof We now formally assert that the composition of a memory clerk, an RPC component, and a reliable memory implements an unreliable memory; and we describe the proof of that assertion.

Since implementation is implication, the assertion that every behavior allowed by an implementation *Imp* satisfies a specification *Spec* is expressed by the formula $Imp \Rightarrow Spec$. However, as discussed in Section 2.3, we expect to prove the correctness of an implementation only under the assumption that the environment behaves correctly. If *Env* is the environment's specification, then we expect $Imp \Rightarrow Spec$ to be satisfied only by behaviors that satisfy *Env*. Thus, correctness of the implementation means that $Env \land Imp \Rightarrow Spec$ is valid. Composition is conjunction, so validity of this formula asserts that every behavior allowed by the composition of the environment and the implementation satisfies the specification.

The assertion that the composition of the clerk, RPC component, reliable memory, and environment specifications implies the unreliable memory's specification is theorem *Impl* of module *MemoryImplementation* in Figure 14 on the next page. The specification of the unreliable memory's environment is for-

module *MemClerk*

import *MemClerkParameters, Sequences*

module *Inner*

parameters

$cstate$: VARIABLE

$$Init(p) \triangleq (cstate[p] = \text{“A”}) \land \neg Rcv.Calling(p)$$

$$RelayArg(p) \triangleq \langle \text{“RemoteCall”}, Head(sndCh[p].arg), Tail(sndCh[p].arg) \rangle$$

$$ReplyVal(p) \triangleq \textbf{if } rcvCh[p].res = \text{“RPCFailure”} \textbf{ then } \text{“MemFailure”} \textbf{ else } rcvCh[p].res$$

$$\begin{aligned} Forward(p) \triangleq\ & \land Snd.Calling(p) \land (cstate[p] = \text{“A”}) \\ & \land Rcv.Call(p, RelayArg(p)) \\ & \land cstate'[p] = \text{“B”} \\ & \land \text{UNCHANGED } Snd.rtrner(p) \end{aligned}$$

$$\begin{aligned} Retry(p) \triangleq\ & \land (cstate[p] = \text{“B”}) \land (rcvCh[p].res = \text{“RPCFailure”}) \\ & \land Rcv.Call(p, RelayArg(p)) \\ & \land \text{UNCHANGED } \langle cstate[p], Snd.rtrner(p) \rangle \end{aligned}$$

$$\begin{aligned} Reply(p) \triangleq\ & \land \neg Rcv.Calling(p) \land (cstate[p] = \text{“B”}) \\ & \land Snd.Return(p, ReplyVal(p)) \\ & \land cstate'[p] = \text{“A”} \\ & \land \text{UNCHANGED } Rcv.caller(p) \end{aligned}$$

$$Next(p) \triangleq Forward(p) \lor Retry(p) \lor Reply(p)$$

$$vars(p) \triangleq \langle cstate[p], Snd.rtrner(p), Rcv.caller(p) \rangle$$

$$\begin{aligned} ISpec \triangleq\ & \forall\, p \in PrIds : \\ & \quad \land Init(p) \land \Box[Next(p)]_{vars(p)} \\ & \quad \land \mathrm{WF}_{vars(p)}(Forward(p)) \land \mathrm{SF}_{vars(p)}(Reply(p)) \end{aligned}$$

$$Spec \triangleq \exists\, cstate : Inner.ISpec$$

Fig. 13. The component specification of the memory clerk.

module *MemoryImplementation*

import *MemoryParameters, Memory*

parameters
$crCh, rmCh$: VARIABLE

---

assumption
$FailureNotAValue \triangleq \text{“RPCFailure”} \notin MemVals$

$Procs \triangleq \{\text{“Read”}, \text{“Write”}\}$

$ArgNum \triangleq [i \in Procs \mapsto \textbf{case } (i = \text{“Read”}) \rightarrow 1,\ (i = \text{“Write”}) \rightarrow 2]$

include *RPC* **as** *R* **with** $sndCh \leftarrow crCh,\ rcvCh \leftarrow rmCh$

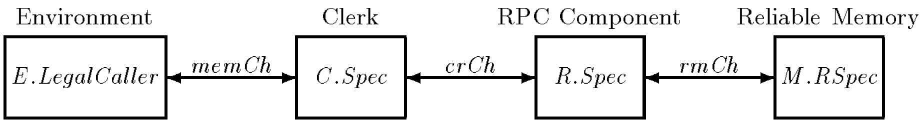
include *MemClerk* **as** *C* **with** $sndCh \leftarrow memCh,\ rcvCh \leftarrow crCh$

include *Memory* **as** *M* **with** $memCh \leftarrow rmCh$

theorem
$Impl \triangleq E.LegalCaller \wedge C.Spec \wedge R.Spec \wedge M.RSpec \Rightarrow USpec$

Fig. 14. Module *MemoryImplementation*.

mula *E.LegalCaller*, included from module *ProcedureInterface* by the imported module *Memory*. The composition is described schematically by the following picture.



When composing two components by conjoining their specifications, the components are “connected” by instantiating their corresponding interface variable parameters with the same variable. The implementing module's specifications have been included with renaming; the specification *USpec* of the memory is imported from the *Memory* module.

The **theorem** statement asserts that the formula named *Impl* is a consequence of the assumptions *FailureNotAValue*[12] and *ParamAssump* (imported from module *MemoryParameters*), and the laws of TLA.

For convenience, we have gathered many of the definitions imported and included by module *MemoryImplementation* in Figure 15 on page 46. In this figure and in our proof, we use the following naming conventions: (i) we eliminate the “*Inner.*” from symbol names—for example, writing $C.Retry(p)$ instead of $C.Inner.Retry(p)$, and (ii) if X is the name of a formula of the form $\forall\, p \in PrIds$:

[12] We believe the theorem to be correct without assumption *FailureNotAValue*, but our proof uses the assumption.

Y, then we let $X(p)$ denote the formula Y—as in $R.ISpec(p)$. The figure also defines the following additional symbols: pv, m, e, c, r, and $E.Next$.[13]

Theorem *Impl* has the form $H \Rightarrow \exists\, mem, result : G$. In predicate logic, one proves a formula $\exists\, y : P(y) \Rightarrow \exists\, x : Q(x)$ by proving $P(y) \Rightarrow Q(\overline{x})$ for a suitable instantiation $\overline{x}$ of x. In temporal logic, the instantiation is called a *refinement mapping* [1]. To prove *Impl*, we define a pair of state functions $\overline{mem}$ and $\overline{result}$ and prove $F \Rightarrow \overline{G}$, where F is the formula obtained by removing the existential quantifiers from H, and $\overline{G}$ is the formula obtained by substituting $\overline{mem}$ and $\overline{result}$ for mem and $result$ in G.

For our proof, we let $\overline{mem}$ equal mem (which comes from $M.RSpec$). To define $\overline{result}$, we must introduce a *history variable* [1]. Intuitively, a history variable a is one that is added to remember what happened in the past. Formally, proving $F \Rightarrow \overline{G}$ by "adding a history variable a" means choosing a variable a that does not appear in F and $\overline{G}$, finding a formula *Hist* of a particular form that guarantees that $\exists\, a : Hist$ is valid, and proving $F \wedge Hist \Rightarrow \overline{G}$. Our history variable *rmhist* is defined so that, for each p in *PrIds*, the value of $rmhist[p]$ is initially equal to "A". It is set to "B" when process p of the reliable memory component returns to the RPC component or when process p of the RPC component issues a failure return to the clerk. It is reset to "A" when process p of the clerk returns to the caller. Formally, we define:

$$
\begin{array}{lcl}
h & \triangleq & rmhist[p] \\
HNext(p) & \triangleq & h' = \textbf{if } M.Return(p) \vee R.Fail(p) \\
 & & \qquad\quad \textbf{then } \text{"B"} \\
 & & \qquad\quad \textbf{else } \ \textbf{if } C.Reply(p) \ \textbf{then } \text{"A"} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \ h \\
Hist & \triangleq & \forall\, p \in PrIds : (h = \text{"A"}) \wedge \Box[HNext(p)]_{\langle c, r, m, h \rangle}
\end{array}
$$

It should be intuitively obvious that, for every p in *PrIds*, formula *Hist* implies that the value of $rmhist[p]$ at any time is determined by the values of c, r, and m up to that time. A general theorem of TLA proves the validity of $\exists\, rmhist : Hist$.

The High-Level Proof We describe a structured proof of theorem *Impl*, in the style of [10]. We first present the high-level proof. It uses the state function $\overline{result}$, which we define later (the high-level proof is independent of its definition), and the temporal formula:

$$
\begin{array}{lcl}
IPImp(p) & \triangleq & E.LegalCaller(p) \wedge C.ISpec(p) \wedge R.ISpec(p) \\
 & & \quad \wedge\, M.RPSpec(p) \wedge (\forall\, l \in MemLocs : M.MSpec(l)) \wedge Hist(p)
\end{array}
$$

For any formula F, we let $\overline{F}$ be the formula obtained by substituting $\overline{result}$ for $result$ in F. Note that all formulas in the proof are interpreted in the con-

[13] We define a number of operators with implicit parameters that are not parameters of module *MemoryImplementation*—for example, the parameters p and $result[p]$ that appear in the definition of m. If we were being truly formal, such definitions would occur in modules that made the parameters explicit, and these modules would then be included in the proof in contexts where the parameters are declared.

The Specification

Unreliable Memory Component (imported from *Memory*)

$$\begin{aligned}
&pv \triangleq pvars(p)\\
&UNext(p) \triangleq Read(p) \lor (\exists\, l : Write(p,l)) \lor Return(p) \lor Fail(p)\\
&UPSpec(p) \triangleq \land\ PInit(p) \land \Box[UNext(p)]_{pv}\\
&\qquad\qquad\qquad \land\ \mathrm{WF}_{pv}(RNext(p)) \land \mathrm{WF}_{pv}(Return(p))\\
&MSpec(l) \triangleq MInit(l) \land \Box[\exists\, p \in PrIds : Write(p,l)]_{mem[l]}\\
&IUSpec \triangleq \land\ \forall\, p \in PrIds : UPSpec(p)\\
&\qquad\qquad\quad \land\ \forall\, l \in MemLocs : MSpec(l)\\
&USpec \triangleq \boldsymbol{\exists}\, mem, result : IUSpec
\end{aligned}$$

The Implementation

The Environment (included from *ProcedureInterface* via import of *Memory*)

$$\begin{aligned}
&e \triangleq E.caller(p)\\
&E.Next(p) \triangleq \exists\, a \in LegalArgs : E.Call(p,a)\\
&E.LegalCaller \triangleq \forall\, p \in PrIds : \neg E.Calling(p) \land \Box[E.Next(p)]_e
\end{aligned}$$

Clerk (included from *MemClerk*)

$$\begin{aligned}
&c \triangleq C.vars(p)\\
&C.Next(p) \triangleq C.Forward(p) \lor C.Retry(p) \lor C.Reply(p)\\
&C.ISpec(p) \triangleq \land\ C.Init(p) \land \Box[C.Next(p)]_c\\
&\qquad\qquad\qquad \land\ \mathrm{WF}_c(C.Forward(p)) \land \mathrm{SF}_c(C.Reply(p))\\
&C.Spec \triangleq \boldsymbol{\exists}\, cstate : \forall\, p \in PrIds : C.ISpec(p)
\end{aligned}$$

RPC Component (included from *RPC*)

$$\begin{aligned}
&r \triangleq R.vars(p)\\
&R.Next(p) \triangleq R.Forward(p) \lor R.Reject(p) \lor R.Fail(p) \lor R.Reply(p)\\
&R.ISpec(p) \triangleq R.Init(p) \land \Box[R.Next(p)]_r \land \mathrm{WF}_r(R.Next(p))\\
&R.Spec \triangleq \boldsymbol{\exists}\, rstate : \forall\, p \in PrIds : R.ISpec(p)
\end{aligned}$$

Reliable Memory Component (included from *Memory*)

$$\begin{aligned}
&m \triangleq M.pvars(p)\\
&M.RNext(p) \triangleq M.Read(p) \lor (\exists\, l : M.Write(p,l)) \lor M.Return(p)\\
&M.RPSpec(p) \triangleq \land\ M.PInit(p) \land \Box[M.RNext(p)]_m\\
&\qquad\qquad\qquad\quad \land\ \mathrm{WF}_m(M.RNext(p)) \land \mathrm{WF}_m(M.Return(p))\\
&M.MSpec(l) \triangleq M.MInit(l) \land \Box[\exists\, p \in PrIds : M.Write(p,l)]_{mem[l]}\\
&M.IRSpec \triangleq \land\ \forall\, p \in PrIds : M.RPSpec(p)\\
&\qquad\qquad\quad \land\ \forall\, l \in MemLocs : M.MSpec(l)\\
&M.RSpec \triangleq \boldsymbol{\exists}\, mem, result : M.IRSpec
\end{aligned}$$

Fig. 15. Formulas defined in module *MemoryImplementation*, plus a few extra definitions.

text of the *MemoryImplementation* module. The variable declarations in the ASSUME (including the implicit declaration of p in the assumption $p \in \mathit{PrIds}$) are necessary, otherwise the formulas in the PROVE part would contain undeclared variables. The following high-level proof is a simple exercise in predicate-logic reasoning with the operators $\forall$ and $\exists$, since these operators (applied to temporal-logic formulas) obey the usual rules of first-order logic.

1. ASSUME: 1. $\mathit{cstate}, \mathit{rstate}, \mathit{mem}, \mathit{result}, \mathit{rmhist}$: VARIABLE
   2. $p \in \mathit{PrIds}$
   PROVE: $\mathit{IPImp}(p) \Rightarrow \overline{\mathit{UPSpec}(p)}$
   PROOF: Proved below.
2. ASSUME: 1. $\mathit{cstate}, \mathit{rstate}, \mathit{mem}, \mathit{result}, \mathit{rmhist}$: VARIABLE
   2. $l \in \mathit{MemLocs}$
   PROVE: $(\forall\, q \in \mathit{PrIds} : \mathit{IPImp}(q)) \Rightarrow \overline{\mathit{MSpec}(l)}$
   PROOF: Proved below.
3. ASSUME: $\mathit{cstate}, \mathit{rstate}, \mathit{mem}, \mathit{result}, \mathit{rmhist}$: VARIABLE
   PROVE: $E.\mathit{LegalCaller} \wedge C.\mathit{ISpec} \wedge R.\mathit{ISpec} \wedge M.\mathit{IRSpec} \wedge \mathit{Hist} \Rightarrow \overline{\mathit{IUSpec}}$
   PROOF: By steps 1 and 2, since $\forall$ distributes over $\wedge$, barring (which is just substitution) distributes over $\forall$ and $\wedge$, and we can deduce $(\forall\, u \in U : P(u)) \Rightarrow (\forall\, u \in U : Q(u))$ by proving $P(u) \Rightarrow Q(u)$ for any u in U.
4. ASSUME: $\mathit{cstate}, \mathit{rstate}, \mathit{mem}, \mathit{result}, \mathit{rmhist}$: VARIABLE
   PROVE: $E.\mathit{LegalCaller} \wedge C.\mathit{ISpec} \wedge R.\mathit{ISpec} \wedge M.\mathit{IRSpec} \wedge \mathit{Hist} \Rightarrow \mathit{USpec}$
   PROOF: By step 3, since we can deduce $F \Rightarrow \exists\, x : G(x)$ by proving $F \Rightarrow G(\overline{x})$, for some state function $\overline{x}$.
5. Q.E.D.
   PROOF: By step 4 and the validity of $\exists\, \mathit{rmhist} : \mathit{Hist}$, since we can deduce $(\exists\, x : F(x)) \Rightarrow G$ by proving $F(x) \Rightarrow G$, assuming x does not occur in G, and we can deduce the equivalence of $\exists\, x, y : F(x) \wedge G(y)$ and $(\exists\, x : F(x)) \wedge (\exists\, y : G(y))$, assuming x does not occur in $G(y)$ and y does not occur in $F(x)$.

The Lower-Level Proof At the heart of our argument lie the proofs of steps 1 and 2. They are based on the predicate-action diagram of Figure 16 on the next page. We introduce the abbreviations T and F for TRUE and FALSE, and UC for UNCHANGED. The operator S is defined to assert that

- For each of the three channels memCh, crCh, and rmCh, there is a call in progress on that channel iff the corresponding one of the first three arguments equals T.
- The values of $\mathit{cstate}[p]$, $\mathit{rstate}[p]$, and $\mathit{rmhist}[p]$ equal the last three arguments, where "AB" indicates a value of either "A" or "B".
- Certain relations hold among the other variables—for example, if the first argument is T, then $\mathit{memCh}[p].\mathit{arg}$ is an element of $\mathit{LegalArgs}$.
- $\mathit{mem}[l]$ is an element of $\mathit{MemVals}$, for all l in $\mathit{MemLocs}$.

The formal definition of S appears in Figure 17 on page 49. It may help in

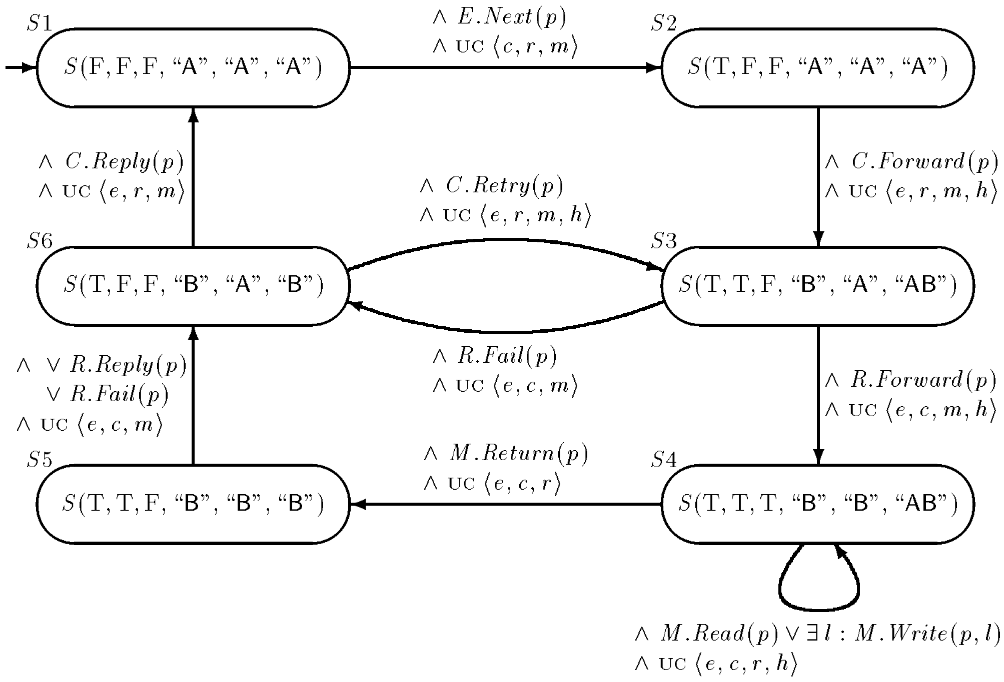


Fig. 16. A predicate-action diagram of $\langle e, c, r, m, h\rangle$ for $IPImp(p)$, where p is an element of $PrIds$.

understanding this definition to observe that:

$$\begin{aligned} E.Calling(p) &\equiv C.Snd.Calling(p) \\ C.Rcv.Calling(p) &\equiv R.Snd.Calling(p) \\ R.Rcv.Calling(p) &\equiv M.E.Calling(p) \end{aligned}$$

We have labeled the state predicates in the predicate-action diagram $S1, \ldots, S6$. We define those labels to be synonymous with their respective predicates, so $S2$ equals $S(\mathrm{T}, \mathrm{F}, \mathrm{F}, \text{"A"}, \text{"A"}, \text{"A"})$. We define the state function $\overline{result}$ so that $\overline{result}[p]$ has the value given in Figure 17, for each p in $PrIds$.

The Proof of Step 1 Intuitively, the proof of step 1 is as follows.

1.1. The implementation's initial condition implies the initial condition $S1$ of the predicate-action diagram.
1.2. The implementation's next-state action implies that the diagram describes all possible state transitions. There are six conditions, one for each state predicate in the diagram.
1.3. The initial condition $S1$ of the predicate-action diagram implies the initial condition $\overline{PInit(p)}$ of $\overline{UPSpec(p)}$.
1.4. Each of the actions allowed by the predicate-action diagram implements (implies) some disjunct of the next-state action $\overline{UNext(p)}$ of $\overline{UPSpec(p)}$, or else leaves $\overline{pv}$ unchanged.

$$
\begin{array}{l}
S(ECalling, CCalling, RCalling, cs, rs, rh) \ \triangleq \\
\quad \land \land \ ECalling \equiv E.Calling(p) \\
\quad \quad \land \ ECalling \Rightarrow (memCh[p].arg \in LegalArgs) \\
\quad \land \land \ CCalling \equiv R.Snd.Calling(p) \\
\quad \quad \land \ CCalling \Rightarrow (crCh[p].arg = C.RelayArg(p)) \\
\quad \quad \land \ \neg CCalling \land (cstate[p] = \text{“B”}) \ \Rightarrow \\
\quad \quad \quad crCh[p].res \ \in \ MemVals \cup \{\text{“OK”}, \text{“BadArg”}, \text{“RPCFailure”}\} \\
\quad \land \land \ RCalling \equiv M.E.Calling(p) \\
\quad \quad \land \ RCalling \Rightarrow (rmCh[p].arg = R.RelayArg(p)) \\
\quad \quad \land \ \neg RCalling \Rightarrow (result[p] = NotAResult) \\
\quad \quad \land \ \neg RCalling \land (rstate[p] = \text{“B”}) \ \Rightarrow \\
\quad \quad \quad rmCh[p].res \ \in \ MemVals \cup \{\text{“OK”}, \text{“BadArg”}\} \\
\quad \land \ cs = cstate[p] \\
\quad \land \ rs = rstate[p] \\
\quad \land \ rmhist[p] \in \textbf{if } rh = \text{“AB”} \ \textbf{then } \{\text{“A”}, \text{“B”}\} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{else } \ \{rh\} \\
\quad \land \ result[p] \in MemVals \cup \{NotAResult, \text{“OK”}, \text{“BadArg”}\} \\
\quad \land \ \forall \, l \in MemLocs \ : \ mem[l] \in MemVals
\end{array}
$$

$$
\begin{array}{l}
\overline{result}[p] \ = \\
\quad \textbf{case } S1 \lor S2 \ \rightarrow \ result[p] \ , \\
\quad \quad S3 \ \rightarrow \ \textbf{if } h = \text{“A”} \ \textbf{then } result[p] \\
\quad \quad \quad \quad \quad \quad \quad \quad \textbf{else } \ \text{“MemFailure”} \ , \\
\quad \quad S4 \ \rightarrow \ \textbf{if } \ (h = \text{“B”}) \land (result[p] = NotAResult) \\
\quad \quad \quad \quad \textbf{then } \text{“MemFailure”} \\
\quad \quad \quad \quad \textbf{else } \ result[p] \ , \\
\quad \quad S5 \ \rightarrow \ rmCh[p].res \ , \\
\quad \quad S6 \ \rightarrow \ \textbf{if } crCh[p].res = \text{“RPCFailure”} \ \textbf{then } \text{“MemFailure”} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{else } \ crCh[p].res
\end{array}
$$

Fig. 17. The formal definitions of S and $\overline{result}[p]$, for p in $PrIds$.

1.5. All the temporal reasoning, including the proof of the fairness properties, is left for the final Q.E.D. step.

The formal proof is as follows.

1.1. $\neg E.Calling(p) \land C.Init(p) \land R.Init(p) \land M.PInit(p)$
$\quad \land (\forall \, l \in MemLocs \ : \ M.MInit(l)) \land (h = \text{“A”}) \ \Rightarrow \ S1$

1.2. ASSUME: 1. $\land \ [E.Next(p)]_e$
$\quad \land \ [C.Next(p)]_c \land [R.Next(p)]_r \land [M.RNext(p)]_m$
$\quad \land \ \forall \, l \in MemLocs \ : \ [\exists \, q \in PrIds : M.Write(q, l)]_{mem[l]}$
$\quad \land \ [HNext(p)]_{\langle c, r, m, h \rangle}$
2. $\neg$UNCHANGED $\langle e, c, r, m, h \rangle$

PROVE: 1. $S1 \Rightarrow S2' \land E.Next(p) \land \text{UC} \ \langle c, r, m \rangle$
2. $S2 \Rightarrow S3' \land C.Forward(p) \land \text{UC} \ \langle e, r, m, h \rangle$

3. $S3 \Rightarrow \vee\ S4' \wedge R.Forward(p) \wedge \text{UC}\ \langle e, c, m, h \rangle$
$\qquad \vee\ S6' \wedge R.Fail(p) \wedge \text{UC}\ \langle e, c, m \rangle$
4. $S4 \Rightarrow \vee\ S4' \wedge (M.Read(p) \vee \exists\, l : M.Write(p, l))$
$\qquad \wedge\ \text{UC}\ \langle e, c, r, h \rangle$
$\qquad \vee\ S5' \wedge M.Return(p) \wedge \text{UC}\ \langle e, c, r \rangle$
5. $S5 \Rightarrow S6' \wedge (R.Reply(p) \vee R.Fail(p)) \wedge \text{UC}\ \langle e, c, m \rangle$
6. $S6 \Rightarrow \vee\ S1' \wedge C.Reply(p) \wedge \text{UC}\ \langle e, r, m \rangle$
$\qquad \vee\ S3' \wedge C.Retry(p) \wedge \text{UC}\ \langle e, r, m, h \rangle$

1.3. $S1 \Rightarrow \overline{PInit(p)}$
1.4. 1. $S1 \wedge S2' \wedge E.Next(p) \wedge \text{UC}\ \langle c, r, m \rangle \Rightarrow \text{UC}\ \overline{pv}$
2. $S2 \wedge S3' \wedge C.Forward(p) \wedge \text{UC}\ \langle e, r, m, h \rangle \Rightarrow \text{UC}\ \overline{pv}$
3. a. $S3 \wedge S4' \wedge R.Forward(p) \wedge \text{UC}\ \langle e, c, m, h \rangle \Rightarrow \text{UC}\ \overline{pv}$
b. $S3 \wedge S6' \wedge R.Fail(p) \wedge \text{UC}\ \langle e, c, m \rangle \Rightarrow \overline{Fail(p)}$
4. a. $S4 \wedge S4' \wedge M.Read(p) \wedge \text{UC}\ \langle e, c, r, h \rangle \Rightarrow \overline{Read(p)}$
b. Assume: l : constant
Prove: $S4 \wedge S4' \wedge M.Write(p, l) \wedge \text{UC}\ \langle e, c, r, h \rangle \Rightarrow \overline{Write(p, l)}$
c. $S4 \wedge S5' \wedge M.Return(p) \wedge \text{UC}\ \langle e, c, r \rangle \Rightarrow \text{UC}\ \overline{pv}$
5. a. $S5 \wedge S6' \wedge R.Reply(p) \wedge \text{UC}\ \langle e, c, m \rangle \Rightarrow \text{UC}\ \overline{pv}$
b. $S5 \wedge S6' \wedge R.Fail(p) \wedge \text{UC}\ \langle e, c, m \rangle \Rightarrow \overline{Fail(p)}$
6. a. $S6 \wedge S1' \wedge C.Reply(p) \wedge \text{UC}\ \langle e, r, m \rangle \Rightarrow \overline{Return(p)}$
b. $S6 \wedge S3' \wedge C.Retry(p) \wedge \text{UC}\ \langle e, r, m, h \rangle \Rightarrow \overline{Fail(p)}$
7. $\text{UC}\ \langle e, c, r, m, h \rangle \Rightarrow \text{UC}\ \overline{pv}$

1.5. Q.E.D.

The proofs of 1.1–1.4 are straightforward, tedious exercises. The part of the proof that shows that the Clerk and RPC components relay their arguments properly requires a bit of simple reasoning about sequences—for example, to prove

$$(memCh[p].arg \in LegalArgs) \Rightarrow (C.RelayArg(p) \in C.LegalRcvArgs)$$

The rest of the proof involves a fairly mindless expanding of definitions and application of first-order logic.

The Proof of Step 1.5 We now give the high-level proof of step 1.5, which is the only part of the proof of step 1 that involves temporal logic.

Let: $Inv(p) \ \triangleq\ S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S6$

1.5.1. $IPImp(p) \Rightarrow \Box Inv(p)$
Proof: By 1.1, 1.2, and the laws of TLA, which allow us in general to deduce $P \wedge (\forall\, u \in U : \Box[N(u)]_{v(u)}) \Rightarrow \Box I$ from $P \Rightarrow I$ together with $I \wedge (\forall\, u \in U : [N(u)]_{v(u)}) \Rightarrow I'$. (Take U to be $\{u_1, u_2, u_3\} \cup MemLocs$, let $N(u_1)$ be $E.Next(p)$, etc.)

1.5.2. $IPImp(p) \wedge \Box Inv(p) \Rightarrow \overline{PInit(p)} \wedge \Box[\overline{UNext(p)}]_{\overline{pv}}$
Proof: 1.1–1.4 show that $IPImp(p)$ implies $\overline{PInit(p)}$ and that
$Inv(p) \wedge [E.Next(p)]_e \wedge [C.Next(p)]_c \wedge [R.Next(p)]_r \wedge [M.RNext(p)]_m$
$\wedge\ (\forall\, l \in MemLocs : [\exists\, q \in PrIds : M.Write(q, l)]_{mem[l]})$
$\wedge\ [HNext(p)]_{\langle c, r, m, h \rangle}$

implies $[\overline{UNext(p)}]_{\overline{pv}}$. The result is now obtained from the laws of TLA, which allow us in general to infer $\Box I \wedge (\forall u \in U : \Box[N(u)]_{v(u)}) \Rightarrow \Box[M]_w$ from $I \wedge I' \wedge (\forall u \in U : [N(u)]_{v(u)}) \Rightarrow [M]_w$.

1.5.3. $IPImp(p) \wedge \Box Inv(p) \Rightarrow \overline{\mathrm{WF}_{pv}(RNext(p))} \wedge \overline{\mathrm{WF}_{pv}(Return(p))}$
PROOF: Described below.

1.5.4. Q.E.D.
PROOF: Step 1.5 (which asserts step 1) follows from 1.5.1–1.5.3 by propositional logic.

The Proof of Step 1.5.3 To complete the proof of step 1, we must prove 1.5.3, which shows that the specification's fairness properties are satisfied. We give an intuitive sketch of the proof. To prove $\overline{\mathrm{WF}_{pv}(RNext(p))}$, we must show that if $\overline{RNext(p)}$ is continuously enabled, then a $\overline{RNext(p)}$ step must eventually occur. To prove $\overline{\mathrm{WF}_{pv}(Return(p))}$, we must show that if $\overline{Return(p)}$ is continuously enabled, then a $\overline{Return(p)}$ step must eventually occur. The two actions are disabled in state $S1$. Therefore, to prove the two fairness properties, it suffices to show that, if any of $S2$–$S6$ ever holds, then $S1$ must eventually hold. It is clear from the diagram that this follows from the two conditions: (i) none of the predicates $S2$–$S5$ can hold forever and (ii) if $S6$ holds repeatedly, then $S1$ must eventually hold. The following implementation fairness properties imply condition (i):

- $\mathrm{WF}_c(C.Forward(p))$ implies that $S2$ cannot hold forever.
- $\mathrm{WF}_r(R.Next(p))$ implies that $S3$ cannot hold forever.
- $\mathrm{WF}_m(M.RNext(p))$ implies that if $S4 \wedge (result[p] = NotAResult)$ holds, then $S4 \wedge (result[p] \neq NotAResult)$ eventually holds, and $\mathrm{WF}_m(M.Return(p))$ then implies that $S5$ eventually holds.
- $\mathrm{WF}_r(R.Next(p))$ implies that $S5$ cannot hold forever.

Condition (ii) follows from $\mathrm{SF}_c(C.Reply(p))$, which implies that if $S6$ holds repeatedly, then $S1$ eventually holds. The proof rules of TLA have been designed expressly to formalize this style of informal reasoning. We omit the formal proof.

The Proof of Step 2 Finally, we must prove step 2. We now confess that, to simplify the exposition, we have structured the proof incorrectly. The proof of 2 requires steps 1.1–1.4 and step 1.5.1, for an arbitrary p in $PrIds$. Those steps should therefore be brought out either as a separate lemma, or as level-1 steps. Here, we violate the rules of structured proofs and use those steps directly in the proof of 2.

2.1. $(\forall\, q \in PrIds : IPImp(q) \wedge \Box Inv(q)) \Rightarrow \overline{MInit}(l)$
PROOF: By the assumption that $l \in MemLocs$, since $M.MInit(l)$ trivially implies $\overline{MInit(l)}$ (the two formulas are the same).

2.2. $\wedge\ [E.Next(p)]_e$
$\wedge\ [C.Next(p)]_c \wedge [R.Next(p)]_r \wedge [M.RNext(p)]_m$
$\wedge\ \forall\, l1 \in MemLocs\ :\ [\exists\, q1 \in PrIds : M.Write(q1, l1)]_{mem[l1]}$
$\wedge\ [HNext(p)]_{\langle c, r, m, h \rangle}$
$\wedge\ Inv(p) \wedge Inv(p)'$
$\wedge\ [M.Write(q, l)]_{mem[l]}$
$\Rightarrow [\overline{Write(q, l)}]_{\overline{mem[l]}}$
PROOF: $Inv(p) \wedge M.Write(p, l)$ implies $S4$, for any p. We consider two cases. (i) If $\neg$UNCHANGED $\langle e, c, r, m, h \rangle$ holds, then the result follows from part 4 of 1.2 and part 4b of 1.4. (ii) If UNCHANGED $\langle e, c, r, m, h \rangle$ holds, then $S4$ and $Inv(p)'$ imply $S4'$, and the result follows from part 4b of 1.4.

2.3. Q.E.D.
PROOF: 2.1, 2.2, and the laws of TLA show that
$$(\forall\, q \in PrIds\ :\ IPImp(q) \wedge \Box Inv(q)) \Rightarrow \overline{MSpec(l)}$$
The result then follows from 1.5.1.

4 Implementing the RPC Component

The problem statement introduces a lossy RPC component, which resembles the RPC component but does not raise "RPCFailure" exceptions and may fail to return. Much as with the memory implementation of Section 3, we specify the lossy RPC and an RPC clerk, and prove that their composition implements the RPC specification.

The problem statement's informal description of the lossy RPC component is problematic for reasons we now explain. The RPC component of Problem 2, specified in module RPC, is just as lossy as the "lossy" one—neither will return to the sender if the receiver fails to return. The additional timing constraints on the lossy RPC component, together with the description of the RPC implementation, suggest that a sender process should be able to issue a new call if a previous one has not returned. However, issuing a second call without waiting for a return violates the handshake protocol of the procedure-calling interface.

A physical component cannot act correctly without some form of synchronization with its environment. If we eliminate the handshake protocol's requirement that the environment must wait for a return before issuing the next call, we must introduce some other form of synchronization. The problem suggests a new protocol in which a sender process can issue a call when either (a) there is no outstanding call, or (b) some time ρ has elapsed since the previous call. For such an interface to be useful, the sender needs to know for which call a result is being returned. This requires either tagging the calls and returns or, more conventionally, specifying that the lossy RPC component never reply to a call more than time ρ after it was issued.

Although replacing the handshake protocol with a timed protocol would produce a more sophisticated example, it is a departure from the problem statement. A literal reading of that statement requires the lossy RPC component to obey the procedure-calling protocol, which forbids more than one outstanding call per process. We therefore adopt this requirement in the specification of the lossy RPC component in Section 4.1 below. This requirement affects our solution to Problem 5, the implementation of an RPC component by composing an RPC clerk with a lossy RPC component. If the lossy RPC component never returns a call by process p and the clerk has returned an RPC failure for that call, then the clerk must always return RPC failures to later calls by p.

4.1 A Lossy RPC

The only novelty in the specification of the lossy RPC component is its use of real-time constraints. We express these constraints as in [2], by introducing a variable parameter *now*, whose value represents the current time, and defining five temporal operators RT, *VTimer*, *MaxTimer*, *MinTimer*, and *NonZeno* (called NZ in [2]). We briefly review these operators.

- The temporal formula $RT(v)$ asserts that (a) *now* is a monotonically nondecreasing real number and (b) steps that change *now* leave v unchanged. Typically, v is a tuple of relevant variables other than *now*, so (b) essentially means that changes to these variables are considered to be instantaneous.
- If A is an action and v a state function such that any A step changes v, and if t is a variable that does not occur in A or v, then the temporal formula $VTimer(t, A, \delta, v) \wedge MaxTimer(t)$ asserts that A cannot be enabled for more than δ time units before the next A step occurs.
- If A is an action and v a state function such that any A step changes v, and if t is a variable that does not occur in A or v, then the temporal formula $VTimer(t, A, \delta, v) \wedge MinTimer(t, A, v)$ asserts that A must be continuously enabled for at least δ time units before the next A step occurs.
- The temporal formula *NonZeno* asserts that *now* keeps increasing without bound, so time marches on.

We define these operators in module *RealTime* of Figure 18 on the next page. This module has appeared before [11, 14], except that earlier versions did not include *NonZeno*. It imports module *Reals*, which defines the set *Real* of real numbers and some of the usual operators on them such as $>$.

The specification of the lossy RPC component is given in module *LossyRPC* of Figure 19 on page 55. The structure of this specification is familiar.[14] This specification is based on that of the RPC component. The initial condition and next-state action are the same as for the ordinary RPC component, except for the use of timing constraints and the absence of $Fail(p)$ steps. The timing constraint $MaxProcess(s, p)$ asserts that a $Forward(p)$ or a $Reject(p)$ step must occur within

[14] We have not bothered to introduce a separate module containing the parameter declarations. Names prefixed by "*Inner*." are defined in submodule *Inner* of the imported *RPC* module. Module *LossyRPC*'s submodule is called *LInner* to avoid name conflicts with the imported submodule.

module *RealTime*

import *Reals*

parameters now : VARIABLE
∞ : CONSTANT

assumption $InfinityUnReal \triangleq \infty \notin Real$

$$
\begin{array}{l}
RT(v) \;\triangleq\; \land\ now \in Real \\
\qquad\qquad\; \land\ \Box[(now' \in \{r \in Real : now < r\}) \land (v' = v)]_{now} \\
VTimer(x, A, \delta, v) \;\triangleq \\
\quad \land\ x = \textbf{if } \text{ENABLED } \langle A \rangle_v \ \textbf{ then } now + \delta \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \infty \\
\quad \land\ \Box[x' = \textbf{if } (\text{ENABLED } \langle A \rangle_v)' \\
\qquad\qquad \textbf{then if } \langle A \rangle_v \lor \neg\text{ENABLED } \langle A \rangle_v \ \textbf{ then } now' + \delta \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } x \\
\qquad\qquad \textbf{else } \infty \ \ ]_{\langle x, v \rangle} \\
MaxTimer(x) \;\triangleq\; \Box[(x \neq \infty) \Rightarrow (now' \leq x)]_{now} \\
MinTimer(x, A, v) \;\triangleq\; \Box[A \Rightarrow (now \geq x)]_v \\
NonZeno \;\triangleq\; \forall\, t \in Real : \Diamond(now > t)
\end{array}
$$

Fig. 18. Module *RealTime*.

δ seconds[15] of when it becomes enabled; the timing constraint $MaxReturn(s, p)$ asserts that a $Return(p)$ step must occur within δ seconds of when it becomes enabled.

4.2 The RPC Implementation

The RPC Clerk The RPC clerk passes requests to the lossy RPC component. According to the problem statement:

> The RPC component is implemented with a Lossy RPC component by passing the *RemoteCall* call through to the Lossy RPC, passing the return back to the caller, and raising an exception if the corresponding return has not been issued after $2\delta + \epsilon$ seconds.

Read literally, this requirement implies that, if the lossy RPC component returns more than $2\delta + \epsilon$ seconds after it is called, then the clerk must raise an exception. For example, if the RPC component returns a result $3\delta + \epsilon$ seconds after it is

[15] Strictly speaking, it asserts that the step must occur before *now* increases by more than δ; we interpret such an increase to represent the passing of δ seconds—rather than the passing of δ years or δ kilometers.

module *LossyRPC*

import *RPC, RealTime, Reals*

parameters δ : CONSTANT

assumption

$DeltaAssump \triangleq (\delta \in Real) \wedge (\delta > 0)$

module *LInner*

parameters

$rstate$: VARIABLE

$$LNext(p) \triangleq Inner.Forward(p) \vee Inner.Reject(p) \vee Inner.Reply(p)$$

$$\begin{aligned} &MaxProcess(s, p) \triangleq \\ &\quad \wedge VTimer(s, Inner.Forward(p) \vee Inner.Reject(p), \delta, \\ &\qquad\qquad \langle Inner.vars(p), Snd.caller(p) \rangle) \\ &\quad \wedge MaxTimer(s) \end{aligned}$$

$$\begin{aligned} &MaxReturn(s, p) \triangleq \\ &\quad \wedge VTimer(s, Inner.Reply(p), \delta, \langle Inner.vars(p), Rcv.rtrner(p) \rangle) \\ &\quad \wedge MaxTimer(s) \end{aligned}$$

$$\begin{aligned} &LISpec \triangleq \\ &\quad \forall p \in PrIds : \wedge Inner.Init(p) \wedge \Box[LNext(p)]_{Inner.vars(p)} \\ &\qquad\qquad\qquad \wedge RT(Inner.vars(p)) \\ &\qquad\qquad\qquad \wedge \exists s : MaxProcess(s, p) \\ &\qquad\qquad\qquad \wedge \exists s : MaxReturn(s, p) \end{aligned}$$

$$Spec \triangleq \exists\, rstate : LInner.LISpec$$

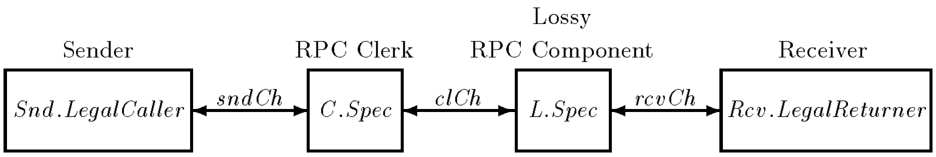
Fig. 19. Module *LossyRPC*.

called and the clerk has not yet raised an exception, then the clerk cannot return the result; it must raise an exception. We find it convenient to adopt the more sensible requirement that the clerk returns an exception only if it has not yet received a result. Thus, if the RPC component returns a result $3\delta + \epsilon$ seconds after it is called, and the clerk has not yet raised an exception, then the clerk will return the result.

There is another aspect of the problem statement that is bizarre. In light of the timing assumptions on the environment, one would expect the clerk to have to return either a result or an exception within some fixed length of time. However, the problem statement makes no such requirement, implying only that the clerk must eventually return. We follow the problem statement in this respect; the resulting mix of eventuality and real-time requirements yields a more interesting example.

Our RPC clerk is specified in module *RPCClerk* of Figure 20 on the next page. The specification is similar to that of the memory clerk. The major differences are that there are no $Retry(p)$ steps, and that there is a $Fail(p)$ timeout action, which cannot be executed until it has been enabled for at least τ seconds. Correctness of the *RPC* component's implementation is proved under the assumption that τ is greater than $2\delta + \epsilon$.

The Implementation Proof The correctness of the RPC implementation is asserted in Module *RPCImplementation* in Figure 21 on page 58. The four components of the implementation are pictured below, where the sender and receiver form the environment.



Formula *RcvTiming* asserts the requirement that the receiver always return within ϵ seconds of when it is called. Theorem *Impl* asserts that the composition of the components' specifications, together with condition *RcvTiming* and the assumption *NonZeno* that time keeps advancing, implies the specification of the *RPC* component.

The proof of theorem *Impl* has a structure similar to that of the proof of the memory implementation in Section 3.2. As in that proof, we eliminate the prefixes "*Inner.*" and "*LInner.*" from symbol names. Definitions from module *RPCImplementation* along with some additional definitions appear in Figure 22 on page 59. We have overloaded symbols such as *C.ISpec*, using the convention that $X(a_1, \ldots, a_n)$ is defined to be X with quantification over $a_1, \ldots, a_n$ removed. The "timing" definitions give names to actions and predicates that occur in the *RealTime* module.

To define the refinement mapping $\overline{rstate}$, we must again introduce a history variable *lrhist*, where $lrhist[p]$ equals "A" iff the lossy RPC component has performed a $Reject(p)$ action, but the RPC clerk component has not yet returned the result to the sender. The formal definition is as follows:

$$\begin{aligned}
h \;&\triangleq\; lrhist[p] \\
HNext(p) \;&\triangleq\; h' = \textbf{if } L.Reject(p) \textbf{ then } \text{“A”} \\
&\qquad\qquad\qquad\quad \textbf{else if } C.Reply(p) \textbf{ then } \text{“B”} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } h \\
Hist \;&\triangleq\; \forall p \in PrIds : (h = \text{“B”}) \wedge \Box[HNext(p)]_{\langle c, l, h \rangle}
\end{aligned}$$

The validity of $\boldsymbol{\exists}\, lrhist : Hist$ is again asserted by a general TLA theorem.

module *RPCClerk*

import *Sequences, Reals*

parameters $sndCh, rcvCh$: VARIABLE
$PrIds, Vals, \tau$: CONSTANT

assumption

$TauAssump \triangleq (\tau \in Real) \wedge (\tau > 0)$

$LegalArgs \triangleq \{\text{“RemoteCall”}\} \times \text{STRING} \times Seq(Vals)$

include *ProcedureInterface* **as** *Snd* **with** $ch \leftarrow sndCh, Args \leftarrow LegalArgs$
include *ProcedureInterface* **as** *Rcv* **with** $ch \leftarrow rcvCh, Args \leftarrow LegalArgs$

module *Inner*

parameters
$cstate$: VARIABLE

$$Init(p) \triangleq (cstate[p] = \text{“A”}) \wedge \neg Rcv.Calling(p)$$

$$\begin{aligned} Forward(p) \triangleq\ & \wedge\ Snd.Calling(p) \wedge (cstate[p] = \text{“A”}) \\ & \wedge\ Rcv.Call(p, sndCh[p].arg) \\ & \wedge\ cstate'[p] = \text{“B”} \\ & \wedge\ \text{UNCHANGED } Snd.rtrner(p) \end{aligned}$$

$$\begin{aligned} Fail(p) \triangleq\ & \wedge\ Rcv.Calling(p) \wedge (cstate[p] = \text{“B”}) \\ & \wedge\ Snd.Return(p, \text{“RPCFailure”}) \\ & \wedge\ cstate'[p] = \text{“A”} \\ & \wedge\ \text{UNCHANGED } Rcv.caller(p) \end{aligned}$$

$$\begin{aligned} Reply(p) \triangleq\ & \wedge\ \neg Rcv.Calling(p) \wedge (cstate[p] = \text{“B”}) \\ & \wedge\ Snd.Return(p, rcvCh[p].res) \\ & \wedge\ cstate'[p] = \text{“A”} \\ & \wedge\ \text{UNCHANGED } Rcv.caller(p) \end{aligned}$$

$$Next(p) \triangleq Forward(p) \vee Fail(p) \vee Reply(p)$$

$$vars(p) \triangleq \langle cstate[p], Snd.rtrner(p), Rcv.caller(p) \rangle$$

$$\begin{aligned} MinFail(s, p) \triangleq\ & \wedge\ VTimer(s, Fail(p), \tau, vars(p)) \\ & \wedge\ MinTimer(s, Fail(p), vars(p)) \end{aligned}$$

$$\begin{aligned} ISpec \triangleq \forall p \in PrIds :\ & \wedge\ Init(p)\ \wedge \Box[Next(p)]_{vars(p)} \\ & \wedge\ RT(vars(p)) \wedge \exists\!\!\exists\, s : MinFail(s, p) \\ & \wedge\ WF_{vars(p)}(Next(p)) \end{aligned}$$

$$Spec \triangleq \exists\!\!\exists\, cstate : Inner.ISpec$$

Fig. 20. The component specification of the RPC clerk.

The High-Level Proof The high-level proof uses the state function $\overline{rstate}$ and the temporal formula

$$\begin{array}{l} IImp(p, et, ct, pt, rt) \;\triangleq \\ \quad Snd.LegalCaller(p) \land Rcv.LegalReturner(p) \land RcvT(p, et) \\ \qquad \land\; C.ISpec(p, ct) \land L.LISpec(p, pt, rt) \land Hist(p) \land NonZeno \end{array}$$

We define $\overline{rstate}$ later; the only property we use in the high-level proof is that the timers et, ct, pt, and rt do not occur in its definition.

1. Assume: 1. $cstate, rstate, et, ct, pt, rt, lrhist$: VARIABLE
   2. $p \in PrIds$

   Prove: $IImp(p, et, ct, pt, rt) \Rightarrow \overline{ISpec(p)}$

   Proof: Proved below.
2. Assume: 1. $cstate, rstate, lrhist$: VARIABLE
   2. $p \in PrIds$

   Prove: $(\boldsymbol{\exists}\, et, ct, pt, rt : Snd.LegalCaller(p) \land Rcv.LegalReturner(p) \land RcvT(p, et) \land C.ISpec(p, ct) \land L.LISpec(p, pt, rt) \land Hist(p)) \;\Rightarrow\; \overline{ISpec(p)}$

   Proof: By step 1 and TLA quantifier rules, because et, ct, pt, and rt do not occur in the definition of $IPSpec(p)$ or $\overline{rstate}$, so they do not occur in $\overline{ISpec(p)}$.

module *RPCImplementation*

import *RPC, RPCParameters, Reals, RealTime*

parameters

τ, δ, ϵ : CONSTANT
$clCh$: VARIABLE

assumption

$$\begin{array}{lll} TDEAssump & \triangleq & \land\; \{\tau, \delta, \epsilon\} \subseteq \{r \in Real : r > 0\} \\ & & \land\; \tau > 2 * \delta + \epsilon \end{array}$$

$$\begin{array}{l} RcvTiming \;\triangleq\; \forall\, p \in PrIds : \\ \qquad \boldsymbol{\exists}\, s : \land\; RT(Rcv.rtrner(p)) \\ \qquad\qquad \land\; VTimer(s, \exists\, v : Rcv.Return(p, v), \epsilon, rcvCh[p]) \\ \qquad\qquad \land\; MaxTimer(s) \end{array}$$

include *LossyRPC* **as** *L* **with** $sndCh \leftarrow clCh,\ rcvCh \leftarrow rcvCh$

include *RPCClerk* **as** *C* **with** $sndCh \leftarrow sndCh,\ rcvCh \leftarrow clCh$

theorem

$$\begin{array}{l} Impl \;\triangleq\; Snd.LegalCaller \land Rcv.LegalReturner \land RcvTiming \\ \qquad\qquad \land\; C.Spec \land L.Spec \land NonZeno \;\Rightarrow\; Spec \end{array}$$

Fig. 21. Module *RPCImplementation*.

The Specification

RPC Component (imported from RPC)

$v \;\triangleq\; vars(p)$

$Init(p) \;\triangleq\; (rstate[p] = \text{“A”}) \wedge \neg Rcv.Calling(p)$

$Next(p) \;\triangleq\; Forward(p) \vee Reject(p) \vee Fail(p) \vee Reply(p)$

$ISpec(p) \;\triangleq\; Init(p) \wedge \Box[Next(p)]_v \wedge \mathrm{WF}_v(Next(p))$

$Spec \;\triangleq\; \boldsymbol{\exists}\, rstate : \forall\, p \in PrIds : ISpec(p)$

The Implementation

The Sender (imported from RPC)

$s \;\triangleq\; Snd.caller(p)$

$Snd.Next(p) \;\triangleq\; \exists\, a \in LegalSndArgs : Snd.Call(p, a)$

$Snd.LegalCaller \;\triangleq\; \forall\, p \in PrIds : \neg Snd.Calling(p) \wedge \Box[Snd.Next(p)]_s$

The Receiver (imported from RPC)

$r \;\triangleq\; Rcv.rtrner(p)$

$Rcv.Next(p) \;\triangleq\; \exists\, v : Rcv.Return(p, v)$

$Rcv.LegalReturner \;\triangleq\; \forall\, p \in PrIds : \Box[Rcv.Next(p)]_r$

$$RcvT(p, et) \;\triangleq\; \begin{array}{l} \wedge\; RT(r) \wedge VTimer(et, Rcv.Next(p), \epsilon, rcvCh[p]) \\ \wedge\; MaxTimer(et) \end{array}$$

RPC Clerk (included from $RPCClerk$)

$c \;\triangleq\; C.vars(p)$

$$C.ISpec(p, ct) \;\triangleq\; \begin{array}{l} \wedge\; C.Init(p) \;\wedge\; \Box[C.Next(p)]_c \;\wedge\; WF_c(C.Next(p)) \\ \wedge\; RT(c) \wedge C.MinFail(ct, p) \end{array}$$

$C.Spec \;\triangleq\; \boldsymbol{\exists}\, cstate : \forall\, p \in PrIds : C.ISpec(p)$

Lossy RPC (included from $LossyRPC$)

$l \;\triangleq\; L.vars(p)$

$$L.LISpec(p, pt, rt) \;\triangleq\; \begin{array}{l} \wedge\; L.Init(p) \wedge \Box[L.LNext(p)]_l \\ \wedge\; RT(l) \wedge L.MaxProcess(pt, p) \wedge L.MaxReturn(rt, p) \end{array}$$

$L.Spec \;\triangleq\; \boldsymbol{\exists}\, rstate : \forall\, p \in PrIds : L.LISpec(p)$

Timing

$TNext(x) \;\triangleq\; (now' \in \{r \in Real : now < r\}) \wedge (x' = x)$

$VInit(t, A, \delta, x) \;\triangleq\; t = \textbf{if } \textsc{Enabled}\, \langle A \rangle_x \textbf{ then } now + \delta \textbf{ else } \infty$

$$VNext(t, A, \delta, x) \;\triangleq\; t' = \begin{array}[t]{l} \textbf{if } (\textsc{Enabled}\, \langle A \rangle_x)' \\ \quad \textbf{then if } \langle A \rangle_x \vee \neg \textsc{Enabled}\, \langle A \rangle_x \;\; \begin{array}[t]{l} \textbf{then } now' + \delta \\ \textbf{else } \; t \end{array} \\ \quad \textbf{else } \; \infty \end{array}$$

$MaxNext(t) \;\triangleq\; (t \neq \infty) \Rightarrow (now' \leq t)$

$MinNext(t, A) \;\triangleq\; A \Rightarrow (now \geq t)$

Fig. 22. Definitions from module *RPCImplementation*, plus a few more.

3. Assume: *cstate*, *rstate*, *lrhist* : variable
   Prove: $Snd.LegalCaller \land Rcv.LegalReturner \land RcvTiming$
   $\land\ C.ISpec \land L.LISpec \land Hist \Rightarrow Spec$
   Proof: By step 2 and TLA quantifier rules.
4. Q.E.D.
   Proof: By step 3, the validity of $\exists\, lrhist : Hist$, and TLA quantifier rules.

The Proof of Step 1 The proof of step 1 is based on the predicate-action diagram of Figure 23 on this page. In this diagram and in the rest of the proof, we assume that $\geq$ is defined so that $r \geq s$ is false unless both r and s are elements of the set *Real* of real numbers. We use notation similar to that in the proof of the

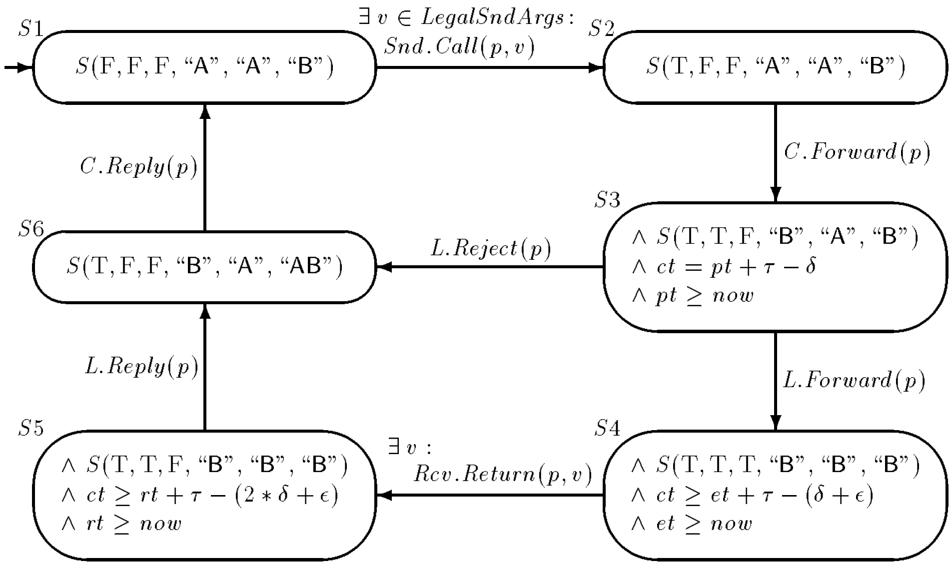


Fig. 23. A predicate-action diagram of $\langle s, r, c, l, h \rangle$ for $IImp(p, et, ct, pt, rt)$.

memory implementation. The formal definition of S appears in Figure 24 on the next page. Again, we define the labels $S1, \ldots, S6$ that appear in the predicate-action diagram of Figure 23 to be synonymous with their respective predicates. We define the state function $\overline{rstate}$ so that:

$$\overline{rstate}[p] \;=\; \textbf{case}\; \begin{array}{ll} S1 \lor S2 \lor S3 & \rightarrow \text{“A”}, \\ S4 \lor S5 & \rightarrow \text{“B”}, \\ S6 & \rightarrow lrhist[p] \end{array}$$

Because we require that the timer variables not occur in $\overline{rstate}$, we must replace $S3$, $S4$, and $S5$ by just their S conjuncts in the actual definition of $\overline{rstate}[p]$.

The proof of the safety part of the RPC component's specification involves proving that Figure 23 is a correct predicate-action diagram for the formula

$$
\begin{array}{l}
S(ECalling, CCalling, LCalling, cs, rs, lh) \ \triangleq \\
\quad \wedge \wedge\ Snd.Calling(p) \equiv ECalling \\
\quad \quad \wedge\ ECalling \Rightarrow (sndCh[p].arg \in LegalSndArgs) \\
\quad \wedge \wedge\ C.Rcv.Calling(p) \equiv CCalling \\
\quad \quad \wedge\ CCalling \Rightarrow (clCh[p].arg = sndCh[p].arg) \\
\quad \wedge\ L.Rcv.Calling(p) \equiv LCalling \\
\quad \wedge\ cstate[p] = cs \\
\quad \wedge\ rstate[p] = rs \\
\quad \wedge \wedge\ lrhist[p] \in \textbf{if } lh = \text{“AB”} \ \textbf{then } \{\text{“A”}, \text{“B”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \{lh\} \\
\quad \quad \wedge\ (lrhist[p] = \text{“A”}) \Rightarrow \wedge\ L.RelayArg(p) \notin L.LegalRcvArgs \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \wedge\ clCh[p].res = \text{“BadCall”} \\
\quad \quad \wedge\ (lrhist[p] = \text{“B”}) \wedge (cs = \text{“B”}) \wedge \neg CCalling \\
\qquad\qquad \Rightarrow (clCh[p].res = rcvCh[p].res) \\
\quad \wedge\ now \in Real
\end{array}
$$

Fig. 24. The formal definition of S, for p in $PrIds$.

$IImp(p, et, ct, pt, rt)$. The key step in this proof is showing that the clerk can never take a $C.Fail(p)$ step. The proof is essentially as follows. Because $C.Fail(p)$ is enabled only when $C.Rcv.Calling(p)$ is true and $cstate[p]$ equals “B”, such a step is possible only in states satisfying $S3$, $S4$, or $S5$. The equality and inequalities in these state predicates, together with the assumptions on τ, δ, and ϵ, imply that ct is greater than now when $S3$, $S4$, or $S5$ holds. However, the conjunct $C.MinFail(ct, p)$ in the clerk's specification asserts that a $C.Fail(p)$ step can occur only when ct is less than or equal to now, so such a step is impossible. This invariance reasoning about timer values is a direct formalization of the intuitive argument that the lossy RPC component must return from a call before the clerk can take a $Fail(p)$ step. It is typical of assertional proofs of real-time properties.

The formal proof of step 1 is analogous to the proof of step 1 of the memory implementation. Steps 1.1 and 1.2 assert that the predicate-action diagram describes the initial state and transitions of formula $IImp(p, et, ct, pt, rt)$; steps 1.3 and 1.4 assert that the system described by the predicate-action implements the initial condition and next-state relation of $\overline{ISpec(p)}$; and step 1.5 completes the proof.

1.1. $\neg Snd.Calling(p) \wedge C.Init(p) \wedge L.Init(p) \wedge (h = \text{“B”}) \wedge (now \in Real)$
$\quad \Rightarrow S1$

1.2. Assume: $\wedge\ [Snd.Next(p)]_s \wedge [Rcv.Next(p)]_r$
$\wedge\ [C.Next(p)]_c \wedge [L.LNext(p)]_l \wedge [HNext(p)]_{\langle c,l,h\rangle}$
$\wedge\ [TNext(r)]_{now} \wedge [TNext(c)]_{now} \wedge [TNext(l)]_{now}$
$\wedge\ [VNext(et, Rcv.Next(p), \epsilon, rcvCh[p])]_{\langle et, rcvCh[p]\rangle}$
$\wedge\ [MaxNext(et)]_{now}$
$\wedge\ [VNext(ct, C.Fail(p), \tau, c)]_{\langle ct, c\rangle}$
$\wedge\ [MinNext(ct, C.Fail(p))]_c$
$\wedge\ [VNext(pt, L.Forward(p) \vee L.Reject(p), \delta,$
$\quad \langle l, C.Rcv.caller(p)\rangle)]_{\langle pt, \langle l, C.Rcv.caller(p)\rangle\rangle}$
$\wedge\ [MaxNext(pt)]_{now}$
$\wedge\ [VNext(rt, L.Reply(p), \delta,$
$\quad \langle l, Rcv.rtrner(p)\rangle)]_{\langle rt, \langle l, Rcv.rtrner(p)\rangle\rangle}$
$\wedge\ [MaxNext(rt)]_{now}$

Prove: 1. $S1 \Rightarrow \vee\ S1' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S2' \wedge Snd.Next(p) \wedge \text{UC}\ \langle r, c, l\rangle$
2. $S2 \Rightarrow \vee\ S2' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S3' \wedge C.Forward(p) \wedge \text{UC}\ \langle s, r, l\rangle$
3. $S3 \Rightarrow \vee\ S3' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S4' \wedge L.Forward(p) \wedge \text{UC}\ \langle s, r, c\rangle$
$\qquad \vee\ S6' \wedge L.Reject(p) \wedge (h' = \text{“A”}) \wedge \text{UC}\ \langle s, r, c\rangle$
4. $S4 \Rightarrow \vee\ S4' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S5' \wedge Rcv.Next(p) \wedge \text{UC}\ \langle s, c, l\rangle$
5. $S5 \Rightarrow \vee\ S5' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S6' \wedge L.Reply(p) \wedge \text{UC}\ \langle s, r, c, h\rangle$
6. $S6 \Rightarrow \vee\ S6' \wedge \text{UC}\ \langle s, r, c, l, h\rangle$
$\qquad \vee\ S1' \wedge C.Reply(p) \wedge \text{UC}\ \langle s, r, l\rangle$

1.3. $S1 \Rightarrow \overline{Init(p)}$

1.4. 1. $S1 \wedge S2' \wedge Snd.Next(p) \wedge \text{UC}\ \langle r, c, l\rangle \Rightarrow \text{UC}\ \overline{v}$
2. $S2 \wedge S3' \wedge C.Forward(p) \wedge \text{UC}\ \langle s, r, l\rangle \Rightarrow \text{UC}\ \overline{v}$
3. $S3 \wedge S4' \wedge L.Forward(p) \wedge \text{UC}\ \langle s, r, c\rangle \Rightarrow \overline{Forward(p)}$
4. $S3 \wedge S6' \wedge L.Reject(p) \wedge (h' = \text{“A”}) \wedge \text{UC}\ \langle s, r, c\rangle \Rightarrow \text{UC}\ \overline{v}$
5. $S4 \wedge S5' \wedge Rcv.Next(p) \wedge \text{UC}\ \langle s, c, l\rangle \Rightarrow \text{UC}\ \overline{v}$
6. $S5 \wedge S6' \wedge L.Reply(p) \wedge \text{UC}\ \langle s, r, c, h\rangle \Rightarrow \text{UC}\ \overline{v}$
7. $S6 \wedge S1' \wedge C.Reply(p) \wedge \text{UC}\ \langle s, r, l\rangle \Rightarrow \overline{Reply(p)} \vee \overline{Reject(p)}$
8. $\text{UC}\ \langle s, r, c, l, h\rangle \Rightarrow \text{UC}\ \overline{v}$

1.5. Q.E.D.

The proofs of steps 1.1–1.4 use simple properties of real numbers and predicate logic. They are omitted.

The Proof of Step 1.5 The high-level proof of step 1.5 is analogous to the proof of step 1.5 of the memory implementation in Section 3.2.

Let: $Inv(p) \ \triangleq\ S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S6$

1.5.1. $IImp(p, et, ct, pt, rt) \Rightarrow \Box Inv(p)$

PROOF: This follows from 1.1 and 1.2, using exactly the same reasoning as in the corresponding step of the memory implementation proof.

1.5.2. $IImp(p, et, ct, pt, rt) \wedge \Box Inv(p) \Rightarrow \overline{Init(p)} \wedge \Box[\overline{Next(p)}]_{\overline{v}}$

PROOF: From 1.1, 1.3, and 1.4, using the TLA proof rules explained in the memory implementation proof.

1.5.3. $IImp(p, et, ct, pt, rt) \wedge \Box Inv(p) \Rightarrow \overline{\mathrm{WF}_v(Next(p))}$

PROOF: Described below.

1.5.4. Q.E.D.

PROOF: Step 1 follows from 1.5.1–1.5.3 by propositional logic.

The Proof of Step 1.5.3 It remains to prove step 1.5.3, which asserts that the fairness property of the *RPC* specification is satisfied. We sketch the argument intuitively. To prove $\overline{\mathrm{WF}_v(Next(p))}$, it is enough to show that $\overline{Next(p)}$ cannot be continuously enabled. The action is disabled in state $S1$. Therefore, it suffices to show that, if any of $S2$–$S6$ ever holds, then $S1$ must eventually hold. It is clear from the predicate-action diagram of Figure 23 that this follows if we can prove that none of the predicates $S2$–$S6$ can hold forever, which is established as follows:

- The implementation fairness property $\mathrm{WF}_c(C.Next(p))$ implies that neither $S2$ nor $S6$ can hold forever.
- To show that $S3$ cannot hold forever, observe that *pt* remains unchanged while $S3$ holds. Since $S3$ asserts that *pt* is greater than or equal to *now*, and *NonZeno* implies that *now* increases without bound, $S3$ must eventually become false. Similar reasoning shows that neither $S4$ nor $S5$ can hold forever.

Observe how *NonZeno* allows us to deduce eventual progress from invariance properties. The *RT*, *VTimer*, *MinTimer*, and *MaxTimer* formulas used to specify real-time system requirements are all safety properties. We infer liveness properties from them by using the *NonZeno* assumption.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
3. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
4. Martín Abadi, Leslie Lamport, and Stephan Merz. The Dagstuhl example—a TLA solution. World Wide Web page at `http://www.research.digital.com/SRC/dagstuhl/dagstuhl.html`. It can also be found by searching the Web for the 26-letter string formed by concatenating `uid` and `lamportdagstuhlspecprob`.
5. Manfred Broy and Leslie Lamport. The RPC-memory specification problem. In this volume. Also available on [4].

6. Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.
7. Rob Gerth, Ruurd Kuiper, and John Segers. Interface refinement in reactive systems. In W. R. Cleaveland, editor, *3rd International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 77–93, Berlin, Heidelberg, 1992. Springer-Verlag.
8. Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332, Amsterdam, September 1983. IFIP, North-Holland.
9. Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web. It can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.
10. Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1993.
11. Leslie Lamport. Hybrid systems in TLA^+. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102, Berlin, Heidelberg, 1993. Springer-Verlag.
12. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
13. Leslie Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, September 1995.
14. Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, September 1994.
15. A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.
16. Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Massachusetts, 1980.
17. Amir Pnueli. The temporal semantics of concurrent programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, July 1979.

Index