Broy-Lamport Specification Problem: A Gurevich Abstract State Machine Solution*

James K. Huggins†
EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA.

December 19, 1996

Abstract

We apply the Gurevich Abstract State Machine methodology to a benchmark specification problem of Broy and Lamport.

As part of the Dagstuhl Workshop on Reactive Systems, Manfred Broy and Leslie Lamport proposed a "Specification Problem" [1]. The problem calls for the specification and validation of a small distributed system dealing with a remote procedure call interface. Broy and Lamport invited proponents of different formal methods to specify and validate the system, in order to compare the results of different methods on a common problem.

We take up the challenge and specify the problem using the Gurevich abstract state machine (ASM) methodology. This paper is self-contained. In Section 1, we present an introduction to Gurevich abstract state machines, including real-time machines. The remaining sections contain the original problem description of Broy and Lamport, interspersed with our ASM specifications and validations.

Acknowledgements. The suggestion to give a ASM solution to this problem was made by Egon Börger and Yuri Gurevich; in particular, Yuri Gurevich actively contributed to an early version of the work [6]. Leslie Lamport made comments on an early draft of this work; Chuck Wallace made comments on a later draft.

1 Gurevich Abstract State Machines

Gurevich abstract state machines, formerly known as *evolving algebras* or *ealgebras*, were introduced in [2]; a more complete definition (including distributed aspects) appeared in [3]. A discussion of real-time ASMs appeared most recently in [4].

We present here a self-contained introduction to ASMs. Sections 1.1 through 1.4 describe distributed ASMs (adapted from [5]); section 1.5 describes real-time ASMs. Those already familiar with ASMs may skip ahead to section 2.

1.1 States

The states of a ASM are simply the structures of first-order logic, except that relations are treated as Boolean-valued functions.

A *vocabulary* is a finite collection of function names, each with a fixed arity. Every ASM vocabulary contains the following *logic symbols*: nullary function names *true*, *false*, *undef*, the equality sign, (the names

*University of Michigan EECS Department Technical Report CSE-TR-320-96.

†huggins@eecs.umich.edu. Partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-95-04375.

of) the usual Boolean operations, and (for convenience) a unary function name Bool. Some function symbols (such as Bool) are tagged as *relations*; others may be tagged as *external*.

A *state* S of vocabulary Υ is a non-empty set X (the *superuniverse* of S), together with interpretations of all function symbols in Υ over X (the *basic functions* of S). A function symbol f of arity r is interpreted as an r-ary operation over X; if $r = 0$, f is interpreted as an element of X. The interpretations of the function symbols *true*, *false*, and *undef* are distinct, and are operated upon by the Boolean operations in the usual way.

Let f be a relation symbol of arity r. We require that (the interpretation of) f is *true* or *false* for every r-tuple of elements of S. If f is unary, it can be viewed as a *universe*: the set of elements a for which $f(a)$ evaluates to *true*. For example, *Bool* is a universe consisting of the two elements (named) *true* and *false*.

Let f be an r-ary basic function and $U_0, \ldots, U_r$ be universes. We say that f has *type* $U_1 \times \ldots \times U_r \rightarrow U_0$ in a given state if $f(\bar{x})$ is in the universe U_0 for every $\bar{x} \in U_1 \times \ldots \times U_r$, and $f(\bar{x})$ has the value *undef* otherwise.

1.2 Updates

The simplest change that can occur to a state is the change of an interpretation of one function at one particular tuple of arguments. We formalize this notion.

A *location* of a state S is a pair $\ell = (f, \bar{x})$, where f is an r-ary function name in the vocabulary of S and $\bar{x}$ is an r-tuple of elements of (the superuniverse of) S. (If f is nullary, ℓ is simply f.) An *update* of a state S is a pair (ℓ, y), where ℓ is a location of S and y is an element of S. To *fire* α at S, put y into location ℓ; that is, if $\ell = (f, \bar{x})$, redefine S to interpret $f(\bar{x})$ as y and leave everything else unchanged.

1.3 Transition Rules

We introduce rules for describing changes to states. At a given state S whose vocabulary includes that of a rule R, R gives rise to a set of updates; to execute R at S, fire all the updates in the corresponding update set. We suppose throughout that a state of discourse S as a sufficiently rich vocabulary.

An *update instruction* R has the form

$$f(t_1, t_2, \ldots, t_n) := t_0$$

where f is an r-ary function name and each t_i is a term. (If $r = 0$, we write $f := t_0$ rather than $f() := t_0$.) The update set for R contains a single update (ℓ, y), where y is the value $(t_0)_S$ of t_0 at S, and $\ell = (f, (x_1, \ldots, x_r))$, where $x_i = (t_i)_S$. In other words, to execute R at S, set $f(x_1, \ldots, x_n)$ to y, where x_i is the value of t_i at S and y is the value of t_0 at S.

A *block rule* R is a sequence $R_1, \ldots, R_n$ of transition rules. To execute R at S, execute all the R_i at S simultaneously. That is, the update set of R at S is the union of the update sets of the R_i at S.

A *conditional rule* R has the form

if g **then** R_0 **else** R_1 **endif**

where g (the *guard*) is a term and R_0, R_1 are rules. The meaning of R is the obvious one: if g evaluates to *true* in S, then the update set for R at S is the same as that for R_0 at S; otherwise, the update set for R at S is the same as that for R_1 at S.

A *choice rule* R has the form

choose v **satisfying** $c(v)$
$R_0(v)$
endchoose

where v is a variable, $c(v)$ is a term involving variable v, and $R_0(v)$ is a rule with free variable v. This rule is nondeterministic. To execute R in state S, choose some element of a of S such that $c(a)$ evaluates to *true* in S, and execute rule R_0, interpreting v as a. If no such element exists, do nothing.

1.4 Distributed Machines

In this section we describe how distributed ASMs evolve over time. The intuition is that each agent of a distributed ASM operates in a sequential manner, and moves of different agents are ordered only when necessary (for example, when two agents attempt contradictory updates).

How do ASMs interact with the external world? There are several ways to model such interactions. One common way is through the use of *external functions*. The values of external functions are provided not by the ASM itself but by some external oracle. The value of an external function may change from state to state without any explicit action by the ASM. If S is a state of a ASM, let S^- be the reduct of S to (the vocabulary) of non-external functions.

Let Υ be a vocabulary containing the universe *agents*, a unary function *Mod*, and a nullary function *Me*. A *distributed ASM program* Π of vocabulary Υ consists of a finite set of *modules*, each of which is a transition rule over the vocabulary Υ. Each module has a unique name different from Υ or *Me*. The intuition is that a module is a program to be executed by one or more agents.

A (global) *state* of Π is a structure S of vocabulary $\Upsilon - \{Me\}$, where different module names are interpreted as different elements of S and *Mod* maps module names to elements of *agents* and all other elements to *undef*. If $Mod(\alpha) = M$, we say that α is an *agent* with program M.

For every agent α, $\text{View}_\alpha(S)$ is the reduct of S to the functions mentioned in α's program $Mod(\alpha)$, extended by interpreting the special function *Me* as α. $\text{View}_\alpha(S)$ can be seen as the local state of agent α corresponding to the global state S. To *fire* an agent α at a state S, execute $Mod(\alpha)$ at state $\text{View}_\alpha(S)$.

A *run* of a distributed ASM program Π is a triple (M, A, σ), satisfying the following conditions:

1. M, the set of *moves* of ρ, is a partially ordered set where every set $\{\nu : \nu \leq \mu\}$ is finite.

   Intuitively, $\nu < \mu$ means that move ν occurs before move μ. If M is totally ordered, we call ρ a *sequential* run.

2. A assigns agents (of S_0) to moves such that every non-empty set $\{\mu : A(\mu) = \alpha\}$ is linearly ordered.

   Intuitively, $A(\mu)$ is the agent which performs move μ; the condition asserts that every agent acts sequentially.

3. σ maps finite initial segments of M (including $\emptyset$) to states of Π.

   Intuitively, $\sigma(X)$ is the result of performing all moves of X; $\sigma(\emptyset)$ is the initial state S_0.

4. (Coherence) If μ is a maximal element of a finite initial segment Y of M, and $X = Y - \{\mu\}$, then $\sigma(Y)^-$ is obtained from $\sigma(X)$ by firing $A(\mu)$ at $\sigma(X)$.

1.5 Real-Time Machines

Real-time ASMs are an extension of distributed ASMs which incorporate the notion of real-time. The notion of state is basically unchanged; what changes is the notion of run. The definitions presented here are taken from [4]; they may not be sufficient to model every real-time system but certainly suffice for the models to be presented in this paper.

Let Υ be a vocabulary with a universe symbol *Reals* which does not contain the nullary function CT. Let Υ^+ be the extension of Υ to include CT. We will restrict attention to Υ^+-states where the universe *Reals* is the set of real numbers and CT evaluates to a real number. Intuitively, CT gives the current time of a given state.

A *pre-run* R of vocabulary Υ^+ is a mapping from the interval $[0, \infty)$ to states of vocabulary Υ^+ satisfying the following requirements, where $\rho(t)$ is the reduct of $R(t)$ to Υ:

1. The superuniverse of every $R(t)$ is that of $R(0)$; that is, the superuniverse does not change during the pre-run.

2. At every $R(t)$, CT evaluates to t. CT represents the current (global) time of a given state.

3. For every $\tau > 0$, there is a finite sequence $0 = t_0 < t_1 < \ldots < t_n = \tau$ such that if $t_i < \alpha < \beta < t_{i+1}$, then $\rho(\alpha) = \rho(\beta)$. That is, for every time t, there is a finite, discrete sequence of moments prior to t when a change occurs in the states of the run (other than a change to CT).

In the remainder of this section, let R be a pre-run of vocabulary Υ^* and $\rho(t)$ be the reduct of $R(t)$ to Υ. $\rho(t+)$ is any state $\rho(t+\epsilon)$ such that $\epsilon > 0$ and $\rho(t+\delta) = \rho(t+\epsilon)$ for all positive $\delta < \epsilon$. Similarly, if $t > 0$, then $\rho(t-)$ is any state $\rho(t-\epsilon)$ such that $0 < \epsilon \leq t$ and $\rho(t-\delta) = \rho(t-\epsilon)$ for all positive $\delta < \epsilon$.

A *pre-run* R of vocabulary Υ^+ is a *run* of Π if it satisfies the following conditions:

1. If $\rho(t+)$ differs from $\rho(t)$ then $\rho(t+)$ is the Υ-reduct of the state resulting from executing some modules $M_1, \ldots, M_k$ at $R(t)$. All external functions have the same values in $\rho(t)$ and $\rho(t+)$.

2. If $t > 0$ and $\rho(t)$ differs from $\rho(t-)$ then they differ only in the values of external functions. All internal functions have the same values in $\rho(t-)$ and $\rho(t)$.

2 The Procedure Interface

The Broy-Lamport specification problem begins as follows:

> The problem calls for the specification and verification of a series of *components*. Components interact with one another using a procedure-calling interface. One component issues a *call* to another, and the second component responds by issuing a *return*. A call is an indivisible (atomic) action that communicates a procedure name and a list of *arguments* to the called component. A return is an atomic action issued in response to a call. There are two kinds of returns, *normal* and *exceptional*. A normal call returns a *value* (which could be a list). An exceptional return also returns a value, usually indicating some error condition. An exceptional return of a value e is called *raising exception* e. A return is issued only in response to a call. There may be "syntactic" restrictions on the types of arguments and return values.
>
> A component may contain multiple *processes* that can concurrently issue procedure calls. More precisely, after one process issues a call, other processes can issue calls to the same component before the component issues a return from the first call. A return action communicates to the calling component the identity of the process that issued the corresponding call.

The modules in our ASM represent components; each component is described by one module. The universe *Components* contains (elements representing) the modules of the system.

The agents in our ASM represent processes; just as each component can have several processes, so a given module can belong to several agents. The universe *agents* contains elements representing the agents of the system. A function *Component: agents* $\rightarrow$ *modules* indicates the component to which a given process belongs[1].

Calls and returns are represented by the execution of transition rules which convey the appropriate information between two processes. The following unary functions are used to transmit this information, where the domain of each function is the universe of *agents*:

- *CallMade*: whether or not this process made a call while handling the current call
- *CallSender*: which process made a call most recently to this process
- *CallName*: what procedure name was sent to this process
- *CallArgs*: what arguments were sent to this process
- *CallReply*: what type of return (normal or exceptional) was sent to this process

[1] For those familiar with the Lipari Guide, this is a renaming of the function *Mod*.

- *CallReplyValue*: what return value was sent to this process

We use two macros (or abbreviations), CALL and RETURN, which are used to make the indivisible (atomic) actions of issuing calls and returns. Each macro performs the task of transferring the relevant information between the caller and callee. The definitions of CALL and RETURN are given in Figure 1.

Abbreviation: CALL(procname, arglist, destination)
choose *p* **satisfying** *(Component(p)=destination* **and** *CallSender(p)=undef)*
CallSender(p) := Me
CallName(p) := procname
CallArgs(p) := arglist
CallMade(Me) := true
CallReply(Me) := undef
CallReplyValue(Me) := undef
endchoose

Abbreviation: RETURN(type, value)
CallReply(CallSender(Me)) := type
CallReplyValue(CallSender(Me)) := value
CallSender(Me) := undef
CallName(Me) := undef
CallArgs(Me) := undef
CallMade(Me) := false

Figure 1: Definitions of the CALL and RETURN abbreviations.

3 A Memory Component

The Broy-Lamport problem calls for the specification of a memory component. The requirements are as follows:

> The component to be specified is a memory that maintains the contents of a set MemLocs of locations. The contents of a location is an element of a set MemVals. This component has two procedures, described informally below. Note that being an element of MemLocs or MemVals is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.

Name	Read
Arguments	loc : an element of MemLocs
Return Value	an element of MemVals
Exceptions	BadArg : argument loc is not an element of MemLocs.
	MemFailure : the memory cannot be read.
Description	Returns the value stored in address loc.

Name	Write
Arguments	loc : an element of MemLocs
	val : an element of MemVals
Return Value	some fixed value
Exceptions	BadArg : argument loc is not an element of MemLocs, or
	argument val is not an element of MemVals.

MemFailure : the write *might* not have succeeded.

Description Stores the value val in address loc.

The memory must eventually issue a return for every Read and Write call.

Define an *operation* to consist of a procedure call and the corresponding return. The operation is said to be *successful* iff it has a normal (nonexceptional) return. The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value InitVal, such that:

- An operation that raises a BadArg exception has no effect on the memory.
- Each successful Read(l) operation performs a single atomic read to location l at some time between the call and return.
- Each successful Write(l, v) operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.
- Each unsuccessful Write(l, v) operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.

A variant of the Memory Component is the Reliable Memory Component, in which no MemFailure exceptions can be raised.

Problem 1 (a) Write a formal specification of the Memory component and of the Reliable Memory component.

(b) Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be.

(c) If your specification of the Memory component allows an implementation that does nothing but raise MemFailure exceptions, explain why this is reasonable.

3.1 ASM Description

As suggested by the description, our ASM has universes *MemLocs* and *MemVals*, and a function *Memory:* $MemLocs \rightarrow MemVals$ which indicates the contents of memory at a given location. We also have the following universes and associated functions:

- *procnames*: names of procedures to be called. Includes the distinguished elements *read* and *write*.
- *lists*: lists of elements in the superuniverse. Unary functions *First, Second* extract the corresponding element from the given list.
- *returntypes*: types of returns to be issued. Includes the distinguished elements *normal* and *exception*.
- *exceptions*: types of exceptions to be issued. Includes the distinguished elements *BadArg* and *MemFailure*.
- *values*: types of return values. Includes the universe *MemVals* as well as a distinguished element *Ok* (used for returns from successful *write* operations, where the nature of the return value is unimportant).

In addition, we use two Boolean-valued external functions, *Succeed* and *Fail*. The intuition is that *Fail* indicates when a component should unconditionally fail, while *Succeed* indicates when a component may succeed during an attempt to write to memory. We require that *Succeed* cannot be false forever; that is, for any state σ in which *Succeed* is false, *Succeed* cannot be false in every successor state $\rho > \sigma$. This ensures that every operation eventually terminates.

```
if CallName(Me)=read then
    if MemLocs(First(CallArgs(Me)))=false then RETURN(exception, BadArg)
    elseif Fail then RETURN (exception, MemFailure)
    else RETURN(normal, Memory(First(CallArgs(Me))))
    endif
elseif CallName(Me)=write then
    if MemLocs(First(CallArgs(Me)))=false or MemVals(Second(CallArgs(Me)))=false then
        RETURN(exception, BadArg)
    elseif Fail then RETURN(exception, MemFailure)
    else
        Memory(First(CallArgs(Me))) := Second(CallArgs(Me))
        if Succeed then RETURN(normal, Ok) endif
    endif
endif
```

Figure 2: Memory component program.

3.2 Component Program

Our ASM contains an unspecified number of agents comprising the memory component. The program for these agents is shown in Figure 2.

In order to make this evolving algebra complete, we need a component which makes calls to the memory component. Of course, we don't wish to constrain how the memory component is called, other than that the CALL interface is used.

Our ASM contains an unspecified number of processes implementing a calling component, whose simple program is shown in Figure 3.

```
if MakeCall then CALL(GetName, GetArgs, MemComponent) endif
```

Figure 3: Calling component program.

This component uses several external functions. *MakeCall* returns a Boolean value indicating whether a call should be made at a given moment. *GetName* and *GetArgs* supply the procedure name and argument list to be passed to the memory component. *MemComponent* is a static (non-external) function indicating the memory component.

We define an *operation* to be the linearly-ordered sequence of moves beginning with the execution of CALL by one component and ending with the execution of RETURN by the component which received the call. In the simplest case, this sequence has exactly two moves (an execution of CALL followed immediately by an execution of RETURN).

A reliable memory component is identical to a memory component except that the external function *Fail* is required to have the value *false* at all times. Thus, the MemFailure exception cannot be raised.

3.3 Correctness

We now show that the Memory and Reliable Memory components specified above satisfy the given requirements. The Broy-Lamport problem does not require us to demonstrate that the specification in fact satisfies the given requirements; nonetheless, it seems quite reasonable and important to do so.

Lemma 1 *Every operation resulting in a* BadArg *exception has no effect on the memory.*

Proof. Observe from the component specification above that any such operation consists of exactly two moves: the original call to the Memory component, and the move which raises the BadArg exception. Observe further that the rule executed by the Memory component to issue the BadArg exception neither reads nor alters the function *Memory*. QED.

Lemma 2 *Every successful* Read(l) *operation performs a single atomic read to location l at some time between the call and return.*

Proof. Observe from the component specification above that any such operation consists of exactly two moves: the original call to the Memory component, and the move which issues the successful return. Observe further that the rule executed by the Memory component to issue the successful return accesses the *Memory* function exactly once, when evaluating *Memory(First(CallArgs(Me)))*. QED.

Lemma 3 *Every successful* Write(l, v) *operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.*

Proof. Observe from the component specification above that any such operation consists of several moves: the original call to the Memory component, and one or more moves which write value v to location l. The last of these writing moves also issues the successful return. QED.

Lemma 4 *Every unsuccessful* Write(l, v) *operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.*

Proof. Observe from the component specification above that any such operation consists of several moves: the original call to the Memory component, zero or more moves which write value v to location l, and the move which returns an exception (either *MemFailure* or *BadArg*). QED.

We thus have immediately:

Theorem 1 *The ASM specification of the memory component correctly implements the requirements given for memory components.*

As to the other issues we are asked to consider:

- It is trivial to see that a reliable memory component is a correct implementation of a memory component; all of the proofs above apply to reliable memory components, other than the fact that Write operations cannot raise *MemFailure* exceptions.
- Our specification does allow for a memory component to return only MemFailure exceptions. It seems reasonable to allow this behavior; it corresponds to the real-world scenario where a memory component is irreparable or cannot be reached through the network.

4 The RPC Component

The Broy-Lamport problem calls for the specification of an RPC (for "remote procedure call") component. Its description is as follows:

> The RPC component interfaces with two environment components, a *sender* and a *receiver*. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender. Parameters of the component are a set Procs of procedure names and a mapping ArgNum, where ArgNum(p) is the number of arguments of each procedure p. The RPC component contains a single procedure:

Name	RemoteCall
Arguments	proc : name of a procedure
	args : list of arguments
Return Value	any value that can be returned by a call to proc
Exceptions	RPCFailure : the call failed
	BadCall : proc is not a valid name or args is not a syntactically correct list of arguments for proc.
	Raises any exception raised by a call to proc
Description	Calls procedure proc with arguments args

A call of RemoteCall(proc, args) causes the RPC component to do one of the following:

- Raise a BadCall exception if args is not a list of ArgNum(proc) arguments.
- Issue one call to procedure proc with arguments args, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the RPCFailure exception.
- Issue no procedure call, and raise the RPCFailure exception.

The component accepts concurrent calls of RemoteCall from the sender, and can have multiple outstanding calls to the receiver.

Problem 2 Write a formal specification of the RPC component.

4.1 ASM Description

We add a new distinguished element *remotecall* to the universe of *procnames*, and distinguished elements *BadCall* and *RPCFailure* to the universe of *exceptions*. We use the standard universe of *integers* in conjunction with the following functions:

- *ArgNum: procnames* $\rightarrow$ *integers* indicates the number of arguments to be supplied with each procedure name
- *Length: lists* $\rightarrow$ *integers* returns the length of the given argument list

A distinguished element *Destination* indicates the component to which this RPC component is supposed to forward its procedure call.

The ASM program for the RPC component is shown in Figure 4.

To complete the specification, we need to supply a component to call the RPC component (such as our caller component from the previous section) and a component for the RPC component to call (such as the memory component from the previous section).

Again, we are not asked to prove that the specification satisfies the requirements given above; the proof is similar to that given in the last section and is omitted.

5 Implementing The Memory Component

The Broy-Lamport problem calls us to create a memory component using a reliable memory component and an RPC component. The requirements are as follows:

> A Memory component is implemented by combining an RPC component with a Reliable Memory component as follows. A Read or Write call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component. If this call returns without raising an RPCFailure exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.) If the call raises an RPCFailure exception, then the implementation

```
if CallName(Me) = remotecall then
    if Length(Second(CallArgs(Me))) ≠ ArgNum(First(CallArgs(Me))) then
        RETURN(exception, BadCall)
    elseif CallMade(Me)=false then
        if Fail then RETURN(exception, RPCFailure)
        else CALL(First(CallArgs(Me)),Second(CallArgs(Me)),Destination)
        endif
    elseif CallReply(Me) ≠ undef then
        if Fail then RETURN(exception, RPCFailure)
        else RETURN(CallReply(Me), CallReplyValue(Me))
        endif
    endif
endif
```

Figure 4: RPC component program.

may either reissue the call to the RPC component or raise a MemFailure exception. The RPC call can be retried arbitrarily many times because of RPCFailure exceptions, but a return from the Read or Write call must eventually be issued.

Problem 3 Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component of Problem 1.

Our implementation includes three modules. Two of the modules are, naturally, instances of the reliable memory component and the RPC component. The program for the third module is shown in Figure 5.

```
if CallName(Me) ≠ undef then
    if CallMade(Me) = false then
        CALL(CallName(Me), CallArgs(Me), RPCComponent)
    elseif CallReply(Me) ≠ undef then
        if (CallReply(Me) ≠ exception) or (CallReplyValue(Me) ≠ RPCFailure) then
            RETURN(CallReply(Me),CallReplyValue(Me))
        elseif Retry then CALL(CallName(Me), CallArgs(Me), RPCComponent)
        else RETURN(exception, MemFail)
        endif
    endif
endif
```

Figure 5: Implementing component program.

The component uses an external Boolean-valued function *Retry*, which indicates whether or not an *RPCFailure* exception should result in another attempt to send the call to the RPC component. The distinguished element *RPCComponent* indicates the RPC component module; the distinguished element *MemFail* is a member of the universe of *exceptions*. We require that *Retry* cannot force the component to resend the call forever; more precisely, for any given agent, and for any state σ such that *CallReply(Me)=exception*, *CallReplyValue(Me)=RPCFailure*, and *Retry=true*, not every successor state $\rho > \sigma$ which satisfies *CallReply(Me)=exception* and *CallReplyValue(Me)=RPCFailure* also satisfies *Reply=true*.

It remains to prove that this implementation is correct. We consider the four original requirements for memory components.

Lemma 5 *Every operation resulting in a* BadArg *exception has no effect on the memory.*

Proof. From the module specifications given above, we observe that an operation resulting in a BadArg exception consists of the following sequence of moves:

- a call from the caller component to the implementing component given above
- a call from the implementing component to the RPC component
- a return of the BadArg exception from the RPC component to the implementing component
- a return of the BadArg exception from the implementing component to the caller component

An examination of the rules involved shows that the *Memory* function is neither read nor updated in any of these moves. QED.

Lemma 6 *Every unsuccessful* Read(l) *operation performs zero or more atomic reads to location l at some time between the call and return.*

This is not one of the original requirements, but the result is used later.

Proof. Fix a sequence of moves which comprise an unsuccessful Read operation. The first element of this sequence is the call from the caller component to the implementation component; the last element of this sequence is the corresponding exceptional return.

The moves between these two elements can be divided into one or more disjoint subsequences of moves, each of which is an operation of the RPC component resulting in an exceptional return. There are several cases.

- The RPC component may raise an *RPCException* without calling the reliable memory component. In this case, *Memory* is never accessed.
- The RPC component may make a call to the reliable memory component, ignore the return value, and raise an *RPCException*. In this case, as was shown earlier, *Memory* is accessed exactly once.
- The RPC component may make a call to the reliable memory component and pass the (exceptional) return value to the caller. In this case, *Memory* is never accessed.

Thus, every operation of the RPC component may result in zero or one atomic reads of location l in *Memory*. Consequently, the entire sequence of RPC component calls may result in zero or more atomic reads of location l in *Memory*. QED.

Lemma 7 *Every successful* Read(l) *operation performs one or more atomic reads to location l at some time between the call and return.*

Note that this is different from the original requirement: that a successful Read(l) operation performs exactly one atomic read of location l. The described composition given above cannot possibly satisfy this requirement. To see this, observe that the *RPCFailure* exception can be raised by the RPC component before any call to the Reliable Memory component (in which case no read of l occurs) or after it calls the Reliable Memory component (in which case a single read of l has occurred). The implementation component above cannot tell the difference between these two conditions; since it is to retry the call some number of times before failing, we cannot ensure that a read to l only occurs once if retries are to be permitted. We choose to allow retries and proceed to prove the modified requirement.

Proof. As in the previous lemma, the first and last move in a successful Read(l) operation are the corresponding calls and return between the environment component and the implementation component. The moves between these two elements can be divided into one or more disjoint subsequences of moves; the last

of these subsequences is a successful RPC operation, while the remainder (if any) are unsuccessful RPC operations.

The previous lemma shows that a sequence of unsuccessful RPC operations for a Read(l) call results in zero or more atomic reads to l. A similar argument shows that a successful RPC call results in a single atomic read to l; the result follows. QED.

Lemma 8 *Every unsuccessful* Write(l, v) *operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.*

Lemma 9 *Every successful* Write(l, v) *operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.*

The proof of these lemmas are similar to those for Read operations and are thus omitted. Combining these lemmas yields the desired conclusion:

Theorem 2 *The ASM specification given correctly implements the requirements given for a memory component, except that* Read *operations may perform more than one atomic read between the call and return.*

6 A Lossy RPC Component

The Broy-Lamport problem calls for the specification of a Lossy RPC Component, whose requirements are as follows:

> The Lossy RPC component is the same as the RPC component except for the following differences, where δ is a parameter.
>
> - The RPCFailure exception is never raised. Instead of raising this exception, the RemoteCall procedure never returns.
> - If a call to RemoteCall raises a BadCall exception, then that exception will be raised within δ seconds of the call.
> - If a RemoteCall(p, a) call results in a call of procedure p, then that call of p will occur within δ seconds of the call of RemoteCall.
> - If a RemoteCall(p, a) call returns other than by raising a BadCall exception, then that return will occur within δ seconds of the return from the call to procedure p.
>
> **Problem 4** Write a formal specification of the Lossy RPC component.

Clearly the requirements suggest the need for a modeling environment which includes time. We use the real-time ASM model presented in [4] and reviewed in Section 1.

Implicit in the description above is the fact that every call and return occurs at a specific moment in time. Consequently, our ASM descriptions of calls and returns will need to record the time at which each call and return occurs. Our ASM will make use of several new unary functions:

- *CallInTime*: the time that a call was received by the given process
- *CallOutTime*: the time that a call was placed by the given process
- *ReturnTime*: the time that a return was received by the given process

The new definitions for the CALL and RETURN abbreviations are shown in Figure 6.

The ASM program for the lossy RPC component is given in Figure 7. It uses a distinguished element δ as specified in the problem description.

Lemma 10 *Every operation of the Lossy RPC component has one of the following forms:*

Abbreviation: CALL(procname, arglist, destination)

choose p **satisfying** *(Component(p)=destination* **and** *CallSender(p)=undef)*

CallSender(p) := Me
CallName(p) := procname
CallArgs(p) := arglist
CallInTime(p) := CT
ReturnTime(p) := undef
CallOutTime(Me) := CT
CallMade(Me) := true
CallReply(Me) := undef
CallReplyValue(Me) := undef

endchoose

Abbreviation: RETURN(type, value)

CallReply(CallSender(Me)) := type
CallReplyValue(CallSender(Me)) := value
ReturnTime(CallSender(Me)) := CT
CallOutTime(CallSender(Me)) := undef
CallInTime(CallSender(Me)) := undef
CallSender(Me) := undef
CallName(Me) := undef
CallArgs(Me) := undef
CallMade(Me) := false

Figure 6: The new CALL and RETURN abbreviations.

```
if CallName(Me) = remotecall then
    if CallInTime(Me) ≠ undef and CallOutTime(Me)=undef then
        if CT ≥ CallInTime(Me) +δ then FAIL
        elseif Length(Second(CallArgs(Me))) ≠ ArgNum(First(CallArgs(Me))) then
            RETURN(exception, BadCall)
        else
            CALL(First(CallArgs(Me)),Second(CallArgs(Me)),Destination)
        endif
    elseif ReturnTime(Me) ≠ undef then
        if CT ≥ ReturnTime(Me) +δ then FAIL
        else RETURN(CallReply(Me), CallReplyValue(Me))
        endif
    endif
endif

where FAIL abbreviates
    CallName(Me) := false
    CallArgs(Me) := false
    CallMade(Me) := false
    CallInTime(Me) := undef
    CallOutTime(Me) := undef
    ReturnTime(Me) := undef
```

Figure 7: Lossy RPC component program.

- *A call which results in no call of the destination component and no return to the caller*
- *A call which results in a* BadCall *exception being raised within δ seconds of the call*
- *A call which results in a call of the destination component within δ seconds of the call, but results in no return to the caller*
- *A call which results in a call of the destination component within δ seconds of the call, whose return is relayed to the caller within δ seconds of the return*

This lemma can be easily verified by analysis of the program above. The key point to notice is that any call or return action by the Lossy RPC component is guaranteed to occur within δ seconds of the event which prompted that action; if too much time passes, the rules ensure that the FAIL abbreviation will be executed instead of the call or return action.

7 The RPC Implementation

The Broy-Lamport calls for one final implementation, whose requirements are as follows:

> 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.

Problem 5 (a) Write a formal specification of this implementation.

(b) Prove that, if every call to a procedure in Procs returns within ϵ seconds, then the implementation satisfies the specification of the RPC component in Problem 2.

The implementation module is shown in Figure 8. It uses a few new functions whose meaning should be clear by now.

```
if CallName(Me) ≠ undef then
    if CallMade(Me) = false then
        CALL(CallName(Me), CallArgs(Me), LossyRPC)
    elseif CallReply(Me) ≠ undef and ReturnTime(Me) ≤ CallOutTime(Me) +2δ + ε then
        RETURN(CallReply(Me), CallReplyValue(Me))
    elseif (CT ≥ CallOutTime(Me) +2δ + ε) then
        RETURN(exception, RPCFailure)
    endif
endif
```

Figure 8: RPC implementation component module.

We combine this implementation module with an instance of the caller module, an instance of the lossy RPC module, and an instance of the memory (or reliable memory) Component (so that the lossy RPC module has someone to whom it passes calls). We assert that the memory component is bounded with bound ϵ.

Theorem 3 *Every operation of the implementation component above has one of the following forms:*

- *a call to the LossyRPC component which returns a* BadCall *exception*
- *a call to the LossyRPC component which makes no call to the Memory component; the implementation component then returns an* RPCFailure *exception*
- *a call to the LossyRPC component which makes a call to the Memory component, waits for the return value from the Memory component, and ignores the return value; the implementation component then returns an* RPCFailure *exception*
- *a call to the LossyRPC component which makes a call to the Memory component from the Memory component, waits for the return value, and returns the value*

Proof. The previous lemma establishes the behavior of the lossy RPC component, to which the implementation component forwards its calls. Notice that any operation which does not result in an BadCall or an RPCFailure exception requires an operation between the lossy RPC component and the memory component, which by supposition is guaranteed to complete within time ϵ. Notice that further that the lossy RPC component must send its call to the memory component within δ seconds of the call in order to receive any return; otherwise, the FAIL abbreviation will be executed, discarding the call. Similarly, the lossy RPC component must relay the return value from the Memory component to the implementing component within δ seconds of the return in order to return anything at all. Thus, any successful return will occur within $2\delta + \epsilon$ seconds of the original call. The implementing component can thus safely assume that any call to the lossy RPC component which has not resulted in a return within that time interval will never have a return, and the RPCFailure exception may be safely generated. QED.

References

[1] Manfred Broy and Leslie Lamport, "Specification Problem", unpublished manuscript.

[2] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. (First published in Bull. EATCS 57 (1991), 264–284.)

[3] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9–36.

[4] Yuri Gurevich and James K. Huggins, "The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions," in *Computer Science Logic, Selected papers from CSL'95*, ed. H. K. Büning, Springer Lecture Notes in Computer Science 1092, 1996, 266–290.

[5] Yuri Gurevich and James K. Huggins, "Equivalence is in the Eye of the Beholder", *Theoretical Computer Science*, vol. 179, June 1997, to appear.

[6] James K. Huggins, "Broy-Lamport Specification Problem: An Evolving Algebras Solution", University of Michigan EECS Department Technical Report CSE-TR-223-94.