Using PVS for an Assertional Verification of the RPC-Memory Specification Problem

Jozef Hooman

Dept. of Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract. The RPC-Memory Specification Problem has been specified and verified in an assertional method, supported by the verification system PVS. Properties of the components are expressed in the higher-order logic of PVS and all implementations have been verified by means of the interactive proof checker of PVS. A simplification of the memory specification - allowing multiple atomic reads - has been proved correct. Additionally, to increase the confidence in the specification, an implementation-oriented specification of the inner memory is shown to be equivalent to our original property-oriented formulation.

0 Introduction

A solution of the RPC-Memory Specification Problem is presented using an assertional method supported by the verification system PVS[1] (Prototype Verification System) [ORS92]. The PVS specification language is a higher-order typed logic, with many built-in types including booleans, integers, rationals, sequences, lists, and sets. Specifications can be structured into a hierarchy of parameterized theories. The tool contains an interactive proof checker with powerful commands for, e.g., inductive proofs, automatic rewriting, and the use of decision procedures for propositional and arithmetical simplification.

All components mentioned in the RPC-Memory Specification Problem are specified in the PVS specification language. Events are used to model the occurrence of actions such as a procedure call, a return of a procedure, and an atomic read or write. Causality relations between events are represented by a partial order. Further we use the notation of event attributes from [JPZ94] to express, for instance, the arguments and the timing of an action.

Moreover, all implementations given in the formulation of the specification problem are verified by means of the PVS proof checker, using a compositional rule for parallel composition. This rule is taken from previous work, where a framework based on assumption/commitment pairs has been devised for the top-down design of distributed real-time systems. In [Hoo94a], a mixed formalism in which programs and assumption/commitment based specifications are combined, has been defined in the PVS specification language.

---

[1] PVS is free available, see WWW page `http://www.csl.sri.com/sri-csl-pvs.html`

To our experience, the use of the tool PVS significantly increases both the speed with which a complete solution can be obtained and the size of the problems that can solved by an assertional approach. Having a mechanical check of the correctness proofs, tedious proof reading is avoided and more attention can be paid to the essential issues, such as a discussion about the axioms and the correctness of the definitions. Since proofs are stored and can be rerun, it is relatively easy to make modifications and check which assumptions are essential. However, when constructing a proof with an interactive tool such as PVS, it is important to have a clear outline of the proof. Starting without a good intuition about the proof could undo a possible gain of time.

In our assertional approach we specify a component by listing its properties. An advantage is that this does not give any implementation bias and one obtains insight in the essential properties that are needed for correctness. There is, however, a danger that the specification does not correspond to the informal specification and it is relatively easy to introduce inconsistencies (see also the remarks about axiomatic specifications in [Lam94]). In the specification given here we have tried to stay as close as possible to the informal formulation of the specification problem. When needed, the resulting formal specification can be transformed into an equivalent, more convenient, specification. E.g., in section 2.4 the original specification of the memory component, in which exactly one atomic read is performed for each successful read operation, is shown to be equivalent to a specification where multiple atomic reads are allowed.

Another technique in PVS is the use of "putative" theorems, that is, one tries to prove properties that should follow from the specification (see [ORSvH95]). Especially the failure of such a proof is helpful because it might reveal errors in the specification or the understanding of the system. As an illustration of this method, in section 2.2 an implementation-oriented specification of the inner memory is proved to be equivalent to our original, property-oriented, formulation.

A brief background of our formal method is presented in section 0.1. In section 0.2, the basic formalism is formulated in the PVS specification language and the use of the PVS proof checker is illustrated by a small example. In section 1 we start with the specification and verification of the RPC-Memory Specification Problem, following the arrangement of the original description. Concluding remarks can be found in section 5.

0.1 Formal Background

Starting point of our approach is a formal method for the specification and verification of distributed real-time systems [Hoo91]. The method has been inspired by Hoare triples (precondition, program, postcondition), which have been extended to deal with the timing of events [Hoo94b]. To distinguish the resulting framework from classical Hoare logic, we use "assumption" and "commitment" instead of "precondition" and "postcondition".

The assumption-commitment structure of specifications is especially convenient for the formulation of compositional proof rules for sequential constructs.

Compositionality means that the specification of a compound programming construct can be derived from the specifications of its components, without knowing the implementation of these components. This makes it possible to verify design steps during the process of program design.

For parallel composition we have formulated a proof rule which essentially expresses that the specification of a parallel system can be derived from the conjunction of the specifications of its components. To obtain such a rule, the assumption-commitment style specification of a component is extended with a static part describing the interface of the component (its alphabet), i.e. the events that could occur during execution. Note that the assertions (assumption and commitment) describe the dynamic part of a system, that is, the events that actually occur during execution, their order, and maybe other attributes of events such as their timing. Then a compositional rule for parallel composition has been formulated by requiring that the assertions of a component only refer to events of its own static interface.

In previous work, our framework for the top-down design of distributed real-time programs has been formulated in PVS [Hoo94a]. There the semantics of programs is essentially a relation on states with some additional components to record the timing behavior of systems. Based on this semantics, the soundness of a number of proof rules for programming constructs (e.g., sequential composition, iteration, parallel composition) have been proved. Since program design is not part of the RPC-Memory Specification, the semantic framework with states is not needed. To avoid unnecessary notation, the states are left out in this paper and the only programming construct considered here is parallel composition.

For simplicity, also the syntactic constraints of the parallel composition rule, which require that the assertions of a component only refer to its static interface, are omitted. We refer to [Hoo91] for more details and a soundness proof of the parallel composition rule. Here the proof rules are simply postulated as axioms. Further, it should be mentioned that the assumption-commitment framework has been used here mainly because it was available in PVS from earlier examples [Hoo94a]. In the RPC-Memory Specification the assumption is not relevant and will be true in all specifications.

0.2 Basic PVS Framework

In general, a PVS specification consists of a number of theories. First the basic specification primitives are defined in a theory `specs` which can be imported by other theories. In this theory we first declare uninterpreted types `Value` and `Program`. The symbol "`+`" indicates that these types are nonempty.

```
specs : THEORY
BEGIN
Value, Program : TYPE+
```

Next we define variables of type `Program` and of the predefined type `bool`. Specifications consist of a pair of boolean formulas (predicates on states in the full framework of [Hoo94a]), called assumption and commitment. To achieve a flexible framework in which programs and specifications can be mixed freely, a specification is also considered as a program. Hence we define `spec(A, C) : Program` in PVS, saying that `spec(A, C)` has type `Program`, given boolean variables `A` and `C`. Note that this equivalent to `spec : [bool, bool -> Program]`.

As usual one should be able to express that a program satisfies a specification and, in general, that one program refines another. For refinement we overload the infix operator `=>` and make it a pre-order on programs by defining `=>` as a constant of the type `(preorder?[Program])` which is constructed using the predefined predicate `preorder?`.

```
prog, prog1, prog2, prog3, prog4 : VAR Program
A, A1, A2, C, C1, C2             : VAR bool

spec(A, C) : Program

=> : (preorder?[Program])
```

Reflexivity and transitivity of `=>` can be made explicit by proving two lemmas, with labels `refrefl` and `reftrans`.

```
refrefl : LEMMA prog => prog

reftrans : LEMMA (prog1 => prog2) IFF
                 (EXISTS prog : (prog1 => prog) AND
                                (prog => prog2))
```

The next two axioms correspond to the usual consequence and conjunction rules of Hoare logic.

```
rulecons : AXIOM (A IMPLIES  A1) AND (C1 IMPLIES C)
                  IMPLIES
                 (spec(A1, C1) => spec(A, C))

ruleconj : AXIOM (prog => spec(A1, C1)) AND (prog => spec(A2, C2))
                  IMPLIES
                 (prog => spec(A1 AND A2, C1 AND C2) ;
```

Finally we define the parallel composition operation on programs, using the infix operator `//`, axiomatize two properties, and derive a convenient proof rule which combines some of the axioms.

```
// : [Program, Program -> Program]

rulepar : AXIOM  spec(A1, C1) // spec(A2, C2)
                 =>
                 spec(A1 AND A2, C1 AND C2)
```

```
% monotonicity property
monopar : AXIOM  (prog3 => prog1) AND (prog4 => prog2)
                  IMPLIES
                 (prog3 // prog4 => prog1 // prog2)

% combined proof rule for parallel composition
% including monotonicity and consequence
specpar : THEOREM (EXISTS A1, A2, C1, C2 :
                      (prog1 => spec(A1, C1)) AND
                      (prog2 => spec(A2, C2)) AND
                      (A IMPLIES A1 AND A2) AND
                      (C1 AND C2 IMPLIES C) )
                   IMPLIES
                   prog1 // prog2 => spec(A, C)
END specs
```

The PVS parser and type-checker can be applied to such a theory to check consistency. Lemmas and theorems can be proved by means of the interactive proof checker of PVS. As an illustration of its user interface, we briefly show a simple proof of lemma `reftrans`. By invoking a certain command on this lemma, the PVS system creates a new EMACS buffer (PVS uses EMACS as its interface), displays the formula, and asks the user for a command by the `Rule?` prompt. (In the text below, only the part after this prompt is typed in by the user, the rest is generated by the system.)

```
reftrans :
  |-------
{1}    (FORALL (prog1: Program, prog2: Program):
          (prog1 => prog2)
              IFF (EXISTS (prog: Program):
                     (prog1 => prog) AND (prog => prog2)))
Rule?
```

Now a proof command of PVS can be invoked. By typing the `(skolem!)` command, Skolem constants `prog1!1` and `prog2!1` are introduced for `prog1` and `prog2`. Observing the `IFF` in the goal, we decide to split the proof of formula `1` into two subgoals, and the system presents the first one.

```
Rule? (skolem!)
Skolemizing, this simplifies to:
reftrans :
  |-------
{1}    (prog1!1 => prog2!1)
           IFF (EXISTS (prog: Program):
                  (prog1!1 => prog) AND (prog => prog2!1))
Rule? (split 1)
Splitting conjunctions, this yields  2 subgoals:
```

```
reftrans.1 :
  |-------
{1}    (prog1!1 => prog2!1)
          IMPLIES (EXISTS (prog: Program):
                    (prog1!1 => prog) AND (prog => prog2!1))
```

This goal is flattened and we ask the system to find a suitable instantiation of the quantified variable by means of the command `(inst?)`.

```
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
reftrans.1 :
{-1}    (prog1!1 => prog2!1)
  |-------
{1}    (EXISTS (prog: Program):
          (prog1!1 => prog) AND (prog => prog2!1))
Rule? (inst?)
Found substitution:
prog gets prog2!1,
Instantiating quantified variables, this simplifies to:
reftrans.1 :
[-1]    (prog1!1 => prog2!1)
  |-------
{1}    (prog1!1 => prog2!1) AND (prog2!1 => prog2!1)
```

Simplifying the formula and using the reflexivity lemma completes this case.

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
reftrans.1 :
[-1]    (prog1!1 => prog2!1)
  |-------
{1}    (prog2!1 => prog2!1)
Rule? (use "refrefl")
Using lemma refrefl,
This completes the proof of reftrans.1.

reftrans.2 :
  |-------
{1}    (EXISTS (prog: Program):
          (prog1!1 => prog) AND (prog => prog2!1))
        IMPLIES (prog1!1 => prog2!1)
```

For this second subgoal, we first use command `(skosimp*)` to simplify it and then introduce the type of the `=>` operator.

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening, this simplifies to:
reftrans.2 :
{-1}    (prog1!1 => prog!1)
{-2}    (prog!1 => prog2!1)
  |-------
{1}    (prog1!1 => prog2!1)

Rule? (typepred "specs.=>")
Adding type constraints for  specs.=>, this simplifies to:
reftrans.2 :
{-1}    preorder?[Program](specs.=>)
[-2]    (prog1!1 => prog!1)
[-3]    (prog!1 => prog2!1)
  |-------
[1]    (prog1!1 => prog2!1)
```

By expanding the definition of `preorder?` into
`reflexive?(specs.=>) & transitive?(specs.=>)`,
hiding reflexivity, and next expanding transitivity, we obtain:

```
reftrans.2 :
{-1}    FORALL (x: Program), (y: Program), (z: Program):
          specs.=>(x, y) & specs.=>(y, z) => specs.=>(x, z)
[-2]    (prog1!1 => prog!1)
[-3]    (prog!1 => prog2!1)
  |-------
[1]    (prog1!1 => prog2!1)
```

By an instantiation of `-1` and the command `(assert)`, the proof is completed.

```
Rule? (inst -1 "prog1!1" "prog!1" "prog2!1")
Instantiating the top quantifier in -1 with the terms:
prog1!1, prog!1, prog2!1, this simplifies to:
reftrans.2 :
{-1}    (specs.=>(prog1!1, prog!1)) & (specs.=>(prog!1, prog2!1))
          => (specs.=>(prog1!1, prog2!1))
[-2]    (prog1!1 => prog!1)
[-3]    (prog!1 => prog2!1)
  |-------
[1]    (prog1!1 => prog2!1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of reftrans.2.
Q.E.D.
Run time  = 5.83 secs.
Real time = 216.45 secs.
```

Note that reasoning in PVS is based on the sequent calculus; the last sequent above consists of antecedents numbered `-1`, `-2`, and `-3`, and a succedent numbered `1`. Please note that the proof above is just to show the user interface and not the power of the theorem prover. There are several more powerful commands, combining skolemization, simplification, instantiation, decision procedures, etc., which allow a reduction of the number of proof steps. For instance, the proof above, with 14 commands, can be easily reduced to 7 commands.

Since the amount of run time, indicated above, depends heavily on the equipment used, we will henceforth mention the number of commands as a measure for the complexity of a proof. Note that we have not tried to minimize proofs, so it gives an indication about the number of interactions that are required by an average user to perform the proofs.

Henceforth, declarations of variables are not repeated. So in subsequent theories `A` is a variable of type `bool`, `prog` a variable of type `Program`, etc.

1 The Procedure Interface

The procedure interface is defined in a theory `procinterface` which imports the theory `specs`. The occurrence of actions is represented by an uninterpreted nonempty type `Event`. We define a strict order `<` on events to represent a "happens before" relation, indicating potential causality. Transitivity of this relation is made explicit in lemma `caustrans`.

```
procinterface : THEORY
BEGIN
IMPORTING specs
Event     : TYPE+
E, E1, E2, E3, E4, E5 : VAR Event

< : (strict_order?[Event]) ;

caustrans : LEMMA (EXISTS E2 : E1 < E2 AND E2 < E3)
                  IMPLIES E1 < E3
```

Similar to [JPZ94] we introduce event attributes. For the current specification problem we start with five attributes, event type `etype`, component number `comp`, process identity `ident`, argument `argu`, and name `name`. The event type `Etype` is defined as an enumeration type consisting of three elements, which implies in PVS that these three elements are different. This fact is used by the PVS decision procedures. The component number is used later to distinguish between different components with the same interface. The argument of an event is a list of values (`list` is a built-in type in PVS). For the name of a procedure we use an uninterpreted type `Name` and declare the names `read` and `write` as constants. We have not used the enumeration type here, since also other names should be allowed (e.g. for the RPC component). Hence we have an explicit axiom to express that `read` and `write` are different. Further we define a constant

ProcName? which has type [Name -> bool], with the additional constraint that read and write are procedure names. Similarly, the constant function ArgNum : [(ProcName?) -> nat] specifies the number of arguments of these procedures.

```
Etype       : TYPE = {call, return, atomic}
Identity    : TYPE+
Arg         : TYPE = list[Value]
Name        : TYPE+
etype : [Event -> Etype]
comp  : [Event -> nat]
ident : [Event -> Identity]
argu  : [Event -> Arg]
name  : [Event -> Name]

read, write : Name
RWDiff  : AXIOM read /= write

ProcName? : {PN : [Name -> bool] | PN(read) AND PN(write)}
ArgNum    : {AN : [(ProcName?) -> nat] | AN(read) = 1 AND
                                         AN(write) = 2}
```

Next we define, by proced, when a pair of events forms a procedure.

```
NoEventBetween(E1,E2) : bool =
 NOT (EXISTS E : name(E) = name(E1) AND ident(E) = ident(E1) AND
                 comp(E) = comp(E1) AND
                 (etype(E) = call OR etype(E) = return) AND
                 E1 < E AND E < E2)

proced (E1,E2) : bool =
    E1 < E2 AND etype(E1) = call AND etype(E2) = return AND
    name(E1) = name(E2) AND ident(E1) = ident(E2) AND
    comp(E1) = comp(E2) AND NoEventBetween(E1,E2)
```

Assuming that events with the same identity are totally ordered (as expressed by axiom IdentTotal), two overlapping procedures (i.e. there is an event between both call/return pairs) with the same name, identity, and component number are proved to be identical.

```
IdentTotal : AXIOM NOT (E1 < E2) AND NOT (E2 < E1) AND
                   ident(E1) = ident(E2)
                   IMPLIES E1 = E2

UniqueProced : LEMMA
   proced (E1,E2) AND proced (E3,E4) AND name(E1) = name(E3) AND
   ident(E1) = ident(E3) AND comp(E1) = comp(E3) AND
   E1 < E5 AND E5 < E2 AND E3 < E5 AND E5 < E4
   IMPLIES
   E1 = E3 AND E2 = E4
```

Finally, we axiomatize that each return has a corresponding call, i.e. is part of a procedure.

```
ReturnCallAx: AXIOM etype(E2) = return
                    IMPLIES (EXISTS E1 : proced (E1,E2))
END procinterface
```

2 A Memory Component

First, in section 2.1, a few basic primitives of memory are defined. Then, in section 2.2, we axiomatize an inner memory component which is used, in section 2.3, to specify the memory component. In section 2.4 we show that this specification can be simplified by not restricting the number of atomic read actions. A reliable memory component is specified in section 2.5. Section 2.6 contains a description of a failing memory component.

2.1 Basic Properties of Memory

Theory `membasic` contains predicates `MemLoc?` and `MemVal?` to describe which values are considered as locations and memory values, respectively. They are extended to arguments, i.e. lists of values, using the short-hand notation `(: v1, v2, v3 , .. :)` of PVS for a list of values. Further we declare four special values and axiomatize a few properties of these values.

```
membasic : THEORY
BEGIN
IMPORTING   procinterface

l, v        : VAR Value
MemLoc?(l) : bool
MemVal?(v) : bool

arg            : VAR Arg
MemLoc(arg)    : bool = (EXISTS l : MemLoc?(l) AND arg = (:l:))
MemVal(arg)    : bool = (EXISTS v : MemVal?(v) AND arg = (:v:))
MemLocVal(arg) : bool = (EXISTS l, v : MemLoc?(l) AND
                                MemVal?(v) AND arg = (:l, v:))

InitVal, WriteOK, BadArg, MemFailure : Value

ValDiff : AXIOM InitVal /= WriteOK AND InitVal /= BadArg AND
                InitVal /= MemFailure AND WriteOK /= BadArg AND
                WriteOK /= MemFailure AND BadArg /= MemFailure

SpecVal: AXIOM MemVal(InitVal) AND NOT MemVal(WriteOK) AND
               NOT MemVal(BadArg) AND NOT MemVal(MemFailure) AND
```

```
               NOT MemLoc(InitVal) AND NOT MemLoc(WriteOK) AND
               NOT MemLoc(BadArg) AND NOT MemLoc(MemFailure)
END membasic
```

2.2 Inner Memory

The RPC-Memory Specification mentions in section 2:

> The memory behaves as if it maintains an array of atomically read and written locations ...

Hence we model the access to the inner memory by a sequence of atomic read and write events, using the PVS predefined type `sequence` which gives infinite sequences (i.e., a function from `nat` to a certain type). To include finite sequences in the model, a special atomic "undefined" event `UE` with name `undef` is declared.

The type `Atoms` consist of events that are either undefined or represent read or write actions. In the subtype `AtomsRW` the undefined ones are excluded. Note that the argument of read and write events is a list of a location and a value. Functions `loc` and `val` are defined to select these elements.

```
innermem : THEORY
BEGIN
IMPORTING membasic

undef       : Name
UndefDiff   : AXIOM undef /= read AND undef /= write
UndefExists : AXIOM (EXISTS E : etype(E) = atomic AND
                                name(E) = undef)

UE  : { E | etype(E) = atomic AND name(E) = undef }

Atoms : TYPE = { E | etype(E) = atomic AND
                     (name(E) = undef OR
                      ((name(E) = read OR name(E) = write) AND
                        MemLocVal(argu(E)) ) ) }

AtomsRW : TYPE = { At : Atoms | name(At) /= undef }
A       : VAR AtomsRW
loc(A)  : (MemLoc?) = car(argu(A))       % first of argument
val(A)  : (MemVal?) = car(cdr(argu(A)))  % second of argument
```

By axiom `TotalOrderSeq`, it is assumed that a sequence of atomic events corresponds to the causal ordering of these events. Finally, predicate `ReadCorrect` expresses that an atomic read event either reads the initial value if there is no preceding write event or it reads the value of the last preceding write event.

```
AtomSeq                : VAR sequence[Atoms]
i, j, j0, j1, j2, k, n : VAR nat
```

```
loca                   : VAR (MemLoc?)

TotalOrderSeq : AXIOM j1 < j2 IFF AtomSeq(j1) < AtomSeq(j2)

% define value read, using two auxiliary predicates

NoWriteBefore(j,loca,AtomSeq) : bool =
    NOT (EXISTS j0 : j0 < j AND name(AtomSeq(j0)) = write AND
                     loc(AtomSeq(j0)) = loca)

NoWriteBetween(j0,j,loca,AtomSeq) : bool =
    NOT (EXISTS j1 : j0 < j1 AND j1 < j AND
                     name(AtomSeq(j1)) = write AND
                     loc(AtomSeq(j1)) = loca)

ReadCorrect ( AtomSeq ) : bool = (FORALL j :
  name(AtomSeq(j)) = read IMPLIES
   ((val(AtomSeq(j)) = InitVal AND
                  NoWriteBefore(j,loc(AtomSeq(j)),AtomSeq)) OR
    (EXISTS (j0 | j0 < j) : name(AtomSeq(j0)) = write AND
                  argu(AtomSeq(j)) = argu(AtomSeq(j0)) AND
                  NoWriteBetween(j0,j,loc(AtomSeq(j)),AtomSeq) )))
END innermem
```

Equivalent Formulation of the Inner Memory Clearly it is easy to make mistakes in specifications, in the sense that the specification does not express the intended behavior. For instance, in the specification above one could forget the initial case. Therefore it is important to have a way to increase the confidence in the specification, e.g. using simulations. In PVS one can try to prove putative theorems, expressing properties that should hold for a correct specification.

Here we give an alternative formulation of the inner memory, in terms of a state-based implementation, and show that it is equivalent to the formulation above. In theory `inmemimpl` we define the state of the inner memory as a function from memory locations to memory values. The behavior of the inner memory is then modeled by a sequence of states. An atomic read does not change the state and returns the value of the location involved in the current state, whereas an atomic write changes the value of the location (represented by the `WITH` construct in PVS by which a new value can be assigned to an element of a sequence or a function).

```
inmemimpl : THEORY
BEGIN
IMPORTING innermem
State     : TYPE = [ (MemLoc?) -> (MemVal?) ]
InitState : State = (LAMBDA loca : InitVal)
mval      : VAR (MemVal?)
```

```
ss        : VAR sequence[State]

MemSeqImpl(AtomSeq,ss) : bool =
 ss(0) = InitState AND
 (FORALL i :
  IF  name(AtomSeq(i)) = read
  THEN val(AtomSeq(i)) = ss(i)(loc(AtomSeq(i))) AND
       ss(i+1) = ss(i)
  ELSIF name(AtomSeq(i)) = write
  THEN ss(i+1) = ss(i) WITH [(loc(AtomSeq(i))) := val(AtomSeq(i))]
  ELSE ss(i+1) = ss(i)
  ENDIF )

MemImpl(AtomSeq) : bool = (EXISTS ss : MemSeqImpl(AtomSeq,ss))
```

The aim is to show that the alternative formulation `MemImpl` is equivalent to the specification `ReadCorrect` of `innermem`. In order to prove that `ReadCorrect` implies `MemImpl`, a sequence of states is constructed by the recursive function `ConstrStatSeq`. Such a recursive definition in PVS requires a `MEASURE` function which should be shown to decrease for recursive calls (which is trivial here).

```
ConstrStatSeq(AtomSeq) (i) : RECURSIVE State =
    IF i = 0
    THEN InitState
    ELSIF name(AtomSeq(i-1)) = write
    THEN ConstrStatSeq(AtomSeq)(i-1)
             WITH [(loc(AtomSeq(i-1))) := val(AtomSeq(i-1))]
    ELSE ConstrStatSeq(AtomSeq)(i-1)
    ENDIF
    MEASURE (LAMBDA n : n)

impllem1 : LEMMA ReadCorrect(AtomSeq) IMPLIES MemImpl(AtomSeq)
```

For the proof in the other direction we have to show, for each read action that does not read the initial value, the existence of a preceding write with the same value. This is done by the following recursive function.

```
FirstWriteBefore(AtomSeq,loca) (j) : RECURSIVE nat =
    IF j=0 THEN j+1       % some value
    ELSIF name(AtomSeq(j-1)) = write AND loc(AtomSeq(j-1)) = loca
    THEN j-1
    ELSE FirstWriteBefore(AtomSeq,loca)(j-1)
    ENDIF
    MEASURE (LAMBDA n : n)

impllem2 : LEMMA MemImpl(AtomSeq) IMPLIES ReadCorrect(AtomSeq)
```

```
memimplcor : THEOREM ReadCorrect(AtomSeq) IFF MemImpl(AtomSeq)

END inmemimpl
```

Of course one could very well imagine that one starts with the operational description of `inmemimpl` and then proves the formulation as given in `innermem`. We have presented the theories here in the order in which they were written originally. In subsequent theories the predicate `ReadCorrect` is used.

2.3 Memory

Next we specify the memory component in the theory `memory`. This theory is parameterized by a natural number `N` to represent the component number. First a few simple properties are defined, expressing the values returned after a call of a read or write event.

```
memory [ N : nat ] : THEORY
BEGIN
IMPORTING innermem

MRead : bool = (FORALL E1 :
 etype(E1) = call AND name(E1) = read AND comp(E1) = N
  IMPLIES (EXISTS E2 : proced(E1,E2) AND
          IF MemLoc(argu(E1))
          THEN argu(E2) = (:MemFailure:) OR MemVal(argu(E2))
          ELSE argu(E2) = (:MemFailure:) OR argu(E2) = (:BadArg:)
          ENDIF ) )

MWrite : bool = (FORALL E1 :
 etype(E1) = call AND name(E1) = write AND comp(E1) = N
  IMPLIES (EXISTS E2 : proced(E1,E2) AND
          IF MemLocVal(argu(E1))
          THEN argu(E2) = (:MemFailure:) OR argu(E2) = (:WriteOK:)
          ELSE argu(E2) = (:MemFailure:) OR argu(E2) = (:BadArg:)
          ENDIF ) )
```

Note that we allow a `MemFailure` if the argument is not correct, although this is not completely clear from the informal formulation. The implementation given later, however, allows this possibility.

Property `MReadInternal` specifies, using predicate `NoReadBetween`, that a successful read procedure performs a single atomic read on the inner memory. A similar specification is given for write procedures, with the difference that a successful write procedure may perform one or more atomic write events.

```
NoReadBetween(E1,E2,j,AtomSeq) : bool =
 NOT (EXISTS (j0 | j0 /= j) :
        E1 < AtomSeq(j0) AND AtomSeq(j0) < E2 AND
```

```
         name(AtomSeq(j0)) = read AND name(AtomSeq(j)) = read AND
         ident(AtomSeq(j0)) = ident(AtomSeq(j)) AND
         loc(AtomSeq(j0)) = loc(AtomSeq(j)) )
```

```
MReadInternal (AtomSeq ) : bool = (FORALL E1, E2:
 comp(E1) = N AND name(E1) = read AND proced(E1,E2) AND
 MemVal(argu(E2))
   IMPLIES (EXISTS j :
    name(AtomSeq(j)) = read AND ident(AtomSeq(j)) = ident(E1) AND
    argu(E1) = (:loc(AtomSeq(j)):) AND
    argu(E2) = (:val(AtomSeq(j)):) AND
    E1 < AtomSeq(j) AND AtomSeq(j) < E2 AND
    NoReadBetween(E1,E2,j,AtomSeq)))
```

```
MWriteInternal ( AtomSeq ) : bool = (FORALL E1, E2 :
      comp(E1) = N AND name(E1) = write AND
      proced(E1,E2) AND argu(E2) = (:WriteOK:)
      IMPLIES (EXISTS j : name(AtomSeq(j)) = name(E1) AND
                          ident(AtomSeq(j)) = ident(E1) AND
                          argu(AtomSeq(j)) = argu(E1) AND
                          E1 < AtomSeq(j) AND AtomSeq(j) < E2))
```

According to the informal specification, an operation that raises a BadArg exception has no effect on the memory. We strengthen this property by expressing that for any internal atomic action there is a causally related procedure which does not raise this exception. Although this is not mentioned in the informal specification, it seems unreasonable to allow spurious internal actions, which might even change the state in case of a write action.

```
MInternalRead ( AtomSeq ) : bool = (FORALL j :
   name(AtomSeq(j)) = read
    IMPLIES (EXISTS E1, E2 : proced(E1,E2) AND comp(E1) = N AND
                             name(E1) = read AND
                             ident(E1) = ident(AtomSeq(j)) AND
                             argu(E1) = (:loc(AtomSeq(j)):) AND
                             argu(E2) /= (:BadArg:) AND
                             E1 < AtomSeq(j) AND AtomSeq(j) < E2))
```

```
MInternalWrite ( AtomSeq ) : bool = (FORALL j :
   name(AtomSeq(j)) = write
    IMPLIES (EXISTS E1, E2 : proced(E1,E2) AND comp(E1) = N AND
                             name(E1) = name(AtomSeq(j)) AND
                             ident(E1) = ident(AtomSeq(j)) AND
                             argu(E1) = argu(AtomSeq(j)) AND
                             argu(E2) /= (:BadArg:) AND
                             E1 < AtomSeq(j) AND AtomSeq(j) < E2))
```

Finally, the commitment of the memory component requires the existence of a sequence of atomic read and write events satisfying the predicates defined above.

```
CMem : bool = MRead AND MWrite AND
              (EXISTS AtomSeq : ReadCorrect(AtomSeq) AND
                                MReadInternal(AtomSeq) AND
                                MWriteInternal(AtomSeq) AND
                                MInternalRead(AtomSeq) AND
                                MInternalWrite(AtomSeq) )
END memory
```

2.4 Simplifying the Memory Specification

The informal specification of the memory component requires that a successful read operation performs exactly one atomic read on the inner memory. This is formalized above by `NoReadBetween` in property `MReadInternal`. In this section we show that this specification is equivalent to a specification that allows multiple atomic reads. First we formulate this simplified specification `CMemSimp` in theory `memsimp`, replacing `MReadInternal` by `MReadInternalSimp`.

```
memsimp [ N : nat ] : THEORY
BEGIN
IMPORTING memory[N]

MReadInternalSimp ( AtomSeq ) : bool = (FORALL E1, E2:
     comp(E1) = N AND name(E1) = read AND proced(E1,E2) AND
     MemVal(argu(E2))
     IMPLIES (EXISTS j : name(AtomSeq(j)) = read AND
                         ident(AtomSeq(j)) = ident(E1) AND
                         argu(E1) = (:loc(AtomSeq(j)):) AND
                         argu(E2) = (:val(AtomSeq(j)):) AND
                         E1 < AtomSeq(j) AND AtomSeq(j) < E2))

CMemSimp : bool = MRead AND MWrite AND
              (EXISTS AtomSeq : ReadCorrect(AtomSeq) AND
                                MReadInternalSimp(AtomSeq) AND
                                MWriteInternal(AtomSeq) AND
                                MInternalRead(AtomSeq) AND
                                MInternalWrite(AtomSeq) )
```

The aim is to prove `CMem IFF CMemSimp`. Observe that `CMem IMPLIES CMemSimp` is trivial, and it remains to show the implication `CMemSimp IMPLIES CMem`. Assuming a sequence `AtomSeq!1` satisfying
`ReadCorrect(AtomSeq!1) AND MReadInternalSimp(AtomSeq!1) AND`
`MWriteInternal(AtomSeq!1) AND MInternalRead(AtomSeq!1) AND`
`MInternalWrite(AtomSeq!1)`,
we have to show the existence of a sequence `AtomSeq` such that

`ReadCorrect(AtomSeq) AND MReadInternal(AtomSeq) AND MWriteInternal(AtomSeq) AND MInternalRead(AtomSeq) AND MInternalWrite(AtomSeq)`.

In our proof, the required sequence is constructed by means of a refinement function `R`. This function does not affect write operations but, to achieve property `MReadInternal(AtomSeq)`, the atomic read operations between a call/return pair of a read procedure are reduced to one. To select such an atomic read, we first define for a certain event `AtomSeq(i)` the set of indices of atomic reads between the same call/return pair with identical attributes.

```
ReadIndexSet(AtomSeq,i) : setof[nat] =
{ j | name(AtomSeq(i)) = read AND name(AtomSeq(j)) = read AND
      ident(AtomSeq(j)) = ident(AtomSeq(i)) AND
      argu(AtomSeq(j)) = argu(AtomSeq(i)) AND
      (EXISTS E1, E2 : proced(E1,E2) AND comp(E1) = N AND
                       name(E1) = read AND
                       ident(E1) = ident(AtomSeq(i)) AND
                       E1 < AtomSeq(i) AND AtomSeq(i) < E2 AND
                       E1 < AtomSeq(j) AND AtomSeq(j) < E2 AND
                       argu(E1) = (:loc(AtomSeq(i)):) AND
                       argu(E2) = (:val(AtomSeq(i)):)}
```

Note that by lemma `UniqueProced` the surrounding procedure `proced(E1,E2)` is unique. Further observe that this set is empty if `AtomSeq(i)` is not a read event or the surrounding procedure does not return its value. Now `R` is defined using the predefined PVS function `choose`, a choice function for nonempty sets, which yields an arbitrary element of the set. A read operation is copied by `R` if its index is selected by this choice function. All other atomic read events are transformed into the undefined event `UE`.

```
R (AtomSeq) : sequence[Atoms] =
     (LAMBDA i : IF (name(AtomSeq(i)) /= read OR
                    (nonempty?(ReadIndexSet(AtomSeq,i)) AND
                          choose(ReadIndexSet(AtomSeq,i)) = i))
                 THEN AtomSeq(i)
                 ELSE UE
                 ENDIF )

ReadInternalImpl : LEMMA MReadInternalSimp(AtomSeq)
                         IMPLIES MReadInternal(R(AtomSeq))

MemSimpTh : THEOREM CMem IFF CMemSimp
END memsimp
```

Theorem `MemSimpTh` has been proved using 14 auxiliary lemmas, and around 300 commands have been used to complete all proofs. (Note that no attempt has been made to minimize the number of commands.)

2.5 Reliable Memory

A simple specification of a reliable memory component, which never raises the `MemFailure` exception, can be obtained by adding a predicate `Reliable` to the memory specification.

```
relmem [ N : nat ] : THEORY
BEGIN
IMPORTING memory[N]

Reliable : bool = (FORALL E: comp(E) = N AND
    etype(E) = return AND (name(E) = read OR name(E) = write)
      IMPLIES argu(E) /= (:MemFailure:) )

CRelMem : bool = CMem AND Reliable
```

When using the specification of the reliable memory it is often convenient to have a more explicit specification. Such a specification can be derived easily from the one above.

```
RelRead : bool = (FORALL E1 :
   etype(E1) = call AND name(E1) = read AND comp(E1) = N
      IMPLIES (EXISTS E2 : proced(E1,E2) AND
                           IF MemLoc(argu(E1))
                           THEN MemVal(argu(E2))
                           ELSE argu(E2) = (:BadArg:)
                           ENDIF ) )

RelWrite : bool = (FORALL E1:
   etype(E1) = call AND name(E1) = write AND comp(E1) = N
      IMPLIES (EXISTS E2 : proced(E1,E2) AND
                           IF MemLocVal(argu(E1))
                           THEN argu(E2) = (:WriteOK:)
                           ELSE argu(E2) = (:BadArg:)
                           ENDIF ) )

CRelExpl : bool = RelRead AND RelWrite AND
                  (EXISTS AtomSeq : ReadCorrect(AtomSeq) AND
                                    MReadInternal(AtomSeq) AND
                                    MWriteInternal(AtomSeq) AND
                                    MInternalRead(AtomSeq) AND
                                    MInternalWrite(AtomSeq) )

RelExplLem : LEMMA CRelMem IFF CRelExpl
END relmem
```

Correctness of Reliable Memory In the theory `relmemcor` we show that a reliable memory component is a correct implementation of a memory component. The main point of the proof is expressed in lemma `RelMemSimp` where we show that the commitment of the reliable memory component implies commitment `CMem` of the memory component. The proof is trivial (and indeed requires one command by the PVS prover), since `CRelMem` equals `CMem AND Reliable`. By `RelMemSimp` and rule `rulecons` of theory `specs` it is easy to prove (two commands in PVS) theorem `RelMemImpl` which expresses that the specification of the reliable memory component implements the specification of the memory component.

```
relmemcor  : THEORY
BEGIN
M : nat
IMPORTING  relmem[M]

RelMemSimp : LEMMA CRelMem IMPLIES CMem

RelMemImpl : LEMMA spec(true,CRelMem) => spec(true,CMem)
END relmemcor
```

2.6 Failing Memory

A failing memory component always returns the `MemFailure` exception, as specified in theory `memfail`.

```
memfail [ N : nat ] : THEORY
BEGIN
IMPORTING memory[N]

FailRead : bool = (FORALL E1 :
   etype(E1) = call AND name(E1) = read AND comp(E1) = N
      IMPLIES (EXISTS E2: proced(E1,E2) AND
                          argu(E2) = (:MemFailure:) ) )

FailWrite : bool = (FORALL E1 :
   etype(E1) = call AND name(E1) = write AND comp(E1) = N
      IMPLIES (EXISTS E2: proced(E1,E2) AND
                          argu(E2) = (:MemFailure:) ) )

CMemFail : bool = FailRead AND FailWrite
END memfail
```

Correctness of Failing Memory Since the specification of the memory component does not restrict the number of times that the `MemFailure` exception can be raised, we can easily prove that a failing component correctly implements a memory component. The main idea is that a failing memory never accesses the inner memory, so we prove `CMem` using an empty sequence of atomic events. Since sequences are infinite in PVS, this is modeled by the sequence which only contains the undefined event `UE`.

```
memfailcor  : THEORY
BEGIN
M : nat
IMPORTING memfail[M]

EmptyAtomSeq : sequence[Atoms] = (LAMBDA (j:nat) : UE )

MemFailSimp : THEOREM CMemFail IMPLIES CMem

MemFailImpl : THEOREM spec(true,CMemFail) => spec(true,CMem)
END memfailcor
```

Here the proof of `MemFailImpl` is again trivially done by two commands. The proof of `MemFailSimp` requires a lot of case distinctions and has been proved by approximately 50 commands. The main point is to show that `CMem` allows the possibility of never accessing the inner memory.

3 Implementing the Memory

3.1 The RPC Component

To specify an RPC component we first formulate a theory with a few basic primitives. Two special values are added to model exceptions. Moreover `RemoteCall` is added as a new procedure name. The argument of this procedure is of type `RPCArg`, a tuple consisting of a name and an argument. The elements of such a tuple are selected by the functions `namearg` and `argarg`.

```
rpcbasic : THEORY
BEGIN
IMPORTING membasic

RPCFailure, BadCall  : Value    % exceptions

RPCValDiff : AXIOM RPCFailure /= BadCall AND
                   WriteOK /= BadCall AND RPCFailure /= WriteOK

RPCSpecVal : AXIOM NOT MemVal?(BadCall) AND
                   NOT MemVal?(RPCFailure)

RemoteCall : Name

RPCArg : TYPE = [ Name, Arg ]
```

```
rpcarg(E)  : RPCArg
namearg(E) : Name = proj_1(rpcarg(E))
argarg(E)  : Arg  = proj_2(rpcarg(E))

END rpcbasic
```

Consider an RPC component with component number `RPC`, which relays calls to a component with component number `N` and relays returns from `N` to the caller. First we specify by `RPCReturn` that it contains a single procedure `RemoteCall` and that a call to the RPC always leads to a return. Commitment `RPCRelay` expresses that a procedure of the RPC either returns `RPCFailure` without any call to `N`, or returns `BadCall` without any call to `N` if the argument is not correct, or leads to one procedure call to `N` where either the value returned is forwarded or `RPCFailure` is raised.

```
rpc [ RPC, N : nat ] : THEORY
BEGIN
IMPORTING rpcbasic

RPCReturn : bool = (FORALL E1 :
  etype(E1) = call AND comp(E1) = RPC
   IMPLIES name(E1) = RemoteCall AND (EXISTS E2 : proced(E1,E2)))

% first three auxiliary definitions used in RPCRelay

CorrectArg(E1) : bool = ProcName?(namearg(E1)) AND
                        ArgNum(namearg(E1)) = length(argarg(E1))

NoCallsBetween(E1,E2) : bool =
 NOT (EXISTS E3 : etype(E3) = call AND comp(E3) = N AND
             ident(E1) = ident(E3) AND name(E3) = namearg(E1) AND
             argu(E3) = argarg(E1) AND E1 < E3 AND E3 < E2)

NoOtherCall(E1,E2,E3) : bool =
 NOT (EXISTS E4 : E4 /= E3 AND etype(E4) = call AND
         comp(E4) = N AND ident(E1) = ident(E4) AND
         name(E4) = namearg(E1) AND argu(E4) = argarg(E1) AND
         E1 < E4 AND E4 < E2)

RPCRelay : bool = (FORALL E1, E2 :
  proced(E1,E2) AND comp(E1) = RPC IMPLIES
    (argu(E2) = (:RPCFailure:) AND NoCallsBetween(E1,E2))
    OR
    IF NOT CorrectArg(E1)
    THEN (argu(E2) = (:BadCall:) AND NoCallsBetween(E1,E2))
    ELSE (EXISTS E3 , E4 : E1 < E3 AND E4 < E2 AND
```

```
            proced(E3,E4) AND comp(E3) = N AND
            ident(E1) = ident(E3) AND
            name(E3) = namearg(E1) AND argu(E3) = argarg(E1) AND
            NoOtherCall(E1,E2,E3) AND
            (argu(E2) = argu(E4) OR argu(E2) = (:RPCFailure:)))
            ENDIF)
```

Note that the informal specification does not explicitly mention that no call is issued in case of a `BadCall`, but it seems reasonable to require this to avoid spurious calls to component `N`. This is also the reason for adding `RPCBackw` which expresses that any procedure of `N` is causally related to a corresponding procedure of `RPC`. Finally we formulate commitment `CRPC` of the RPC component.

```
RPCBackw : bool = (FORALL E3, E4 :
  proced(E3,E4) AND comp(E3) = N
    IMPLIES (EXISTS E1, E2 : E1 < E3 AND E4 < E2 AND
            proced(E1,E2) AND comp(E1) = RPC AND
            ident(E1) = ident(E3) AND
            name(E3) = namearg(E1) AND
            argu(E3) = argarg(E1) AND
            (argu(E2) = argu(E4) OR argu(E2) = (:RPCFailure:)) AND
            CorrectArg(E1) ) )

CRPC : bool = RPCReturn AND RPCRelay AND RPCBackw
END rpc
```

3.2 The Implementation

A memory component with identification `M` is implemented using a clerk component which forwards read and write calls to the RPC component, say with number `RPC`. In the normal situation, this RPC component then relays these calls to a reliable memory component `RM` (see figure 1).
In the theory `clerk` we first give two axioms relating exceptions from an RPC component to special values of a memory component. Further, by `ClerkReturn` the clerk guarantees that a call to `M` always leads to a return.

```
clerk [M, RPC, RM : nat ] : THEORY
BEGIN
IMPORTING memsimp[M], rpc[RPC,RM]

ClerkValDiff: AXIOM BadArg /= RPCFailure AND BadArg /= BadCall AND
                    WriteOK /= RPCFailure AND WriteOK /= BadCall

ClerkSpecialVal : AXIOM NOT MemVal?(RPCFailure) AND
                        NOT MemVal?(BadCall) AND
                        NOT MemLoc?(RPCFailure) AND
                        NOT MemLoc?(BadCall)
```

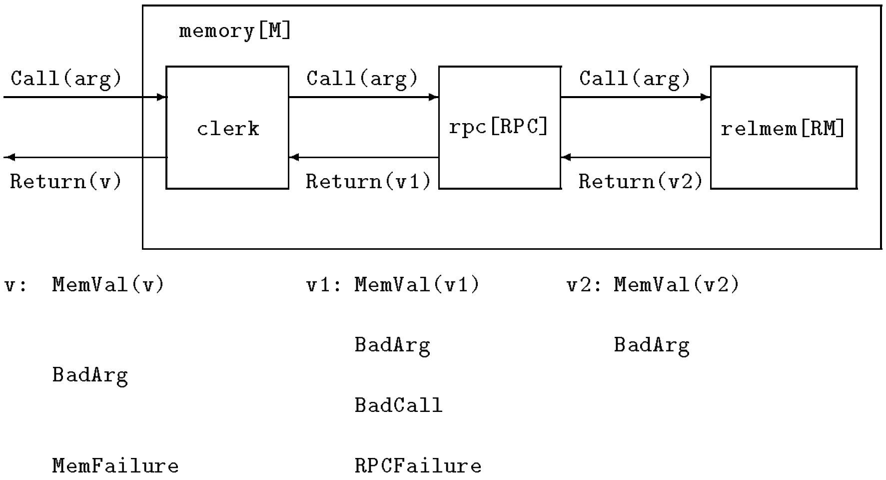

Fig. 1. Implementation of a memory component

```
ClerkReturn : bool =
    (FORALL E1 : etype(E1) = call AND comp(E1) = M
                 IMPLIES (EXISTS E2 : proced(E1,E2)))
```

Next we define abbreviation `ClerkReturnValue` to describe how values received from the RPC component are relayed to the caller. Note that, in addition to the informal description, the clerk transforms the `BadCall` exception into the `BadArg` exception. Property `ClerkRelay` expresses that a procedure call to `M` eventually leads to a procedure call to the RPC component, where the corresponding return value is described by `ClerkReturnValue`. Finally `ClerkBackw` expresses that a call/return pair of `RPC` is surrounded by a corresponding call/return pair of `M`.

```
ClerkReturnValue(E2,E4) : bool =
   (argu(E4) = (:RPCFailure:) AND argu(E2) = (:MemFailure:)) OR
   (argu(E4) = (:BadCall:) AND argu(E2) = (:BadArg:)) OR
   (argu(E4) /= (:RPCFailure:) AND argu(E4) /= (:BadCall:) AND
       argu(E2) = argu(E4))

ClerkRelay : bool = (FORALL E1, E2 :
   proced(E1,E2) AND comp(E1) = M IMPLIES
     (EXISTS E3, E4 : E1 < E3 AND E4 < E2 AND
             proced(E3,E4) AND comp(E3) = RPC AND
             name(E3) = RemoteCall AND ident(E3) = ident(E1) AND
             name(E1) = namearg(E3) AND argu(E1) = argarg(E3) AND
             ClerkReturnValue(E2,E4) ) )
```

```
ClerkBackw : bool = (FORALL E3, E4 :
   proced(E3,E4) AND comp(E3) = RPC IMPLIES
     (EXISTS E1, E2 : E1 < E3 AND E4 < E2 AND proced(E1,E2) AND
             comp(E1) = M AND ident(E3) = ident(E1) AND
             name(E1) = namearg(E3) AND argu(E1) = argarg(E3) AND
             ClerkReturnValue(E2,E4) ) )

CClerk : bool = ClerkReturn AND ClerkRelay AND ClerkBackw
END clerk
```

Correctness of the Implementation The correctness of the implementation is proved in theory `clerkcor`. The aim is to prove theorem `MemImpl`, expressing that the parallel composition of a clerk, an RPC component, and a reliable memory component implements the memory module. This can be shown easily, by rule `specpar`, if `CClerk AND CRPC AND CRelMem IMPLIES memory[M].CMem`. (Different instances of a predicate are distinguished by preceding them with a theory name and an actual parameter.) Observe that the implementation of the memory can retry an atomic read to the inner memory of the reliable memory arbitrarily many times because of `RPCFailure` exceptions. Therefore we use `CMemSimp` instead of `CMem`, which is justified by theorem `MemSimpTh` (see section 2.4). Further, using lemma `RelExplLem` of section 2.5, `CRelMem` is replaced by `CRelExpl`. Then for each property of `CMemSimp` we formulate a lemma, showing on which properties of the implementation it depends. The proof of these lemmas is rather straightforward (in fact, the main work has been done in section 2.4).

```
clerkcor  : THEORY
BEGIN
M, RPC, RM   : nat
IMPORTING clerk[M,RPC,RM], relmem[RM]

RLem : LEMMA ClerkReturn AND ClerkRelay AND RPCRelay AND RelRead
             IMPLIES memory[M].MRead

WLem : LEMMA ClerkReturn AND ClerkRelay AND RPCRelay AND RelWrite
             IMPLIES memory[M].MWrite

ReadInternalLem : LEMMA ClerkRelay AND RPCRelayRead AND
                        memory[RM].MReadInternal(AtomSeq)
                        IMPLIES MReadInternalSimp(AtomSeq)

WriteInternalLem : LEMMA ClerkRelay AND RPCRelayWrite AND
                         memory[RM].MWriteInternal(AtomSeq)
                         IMPLIES memory[M].MWriteInternal(AtomSeq)
```

```
InternalReadLem : LEMMA memory[RM].MInternalRead(AtomSeq) AND
                        RPCBackw AND ClerkBackw AND RelRead
                        IMPLIES memory[M].MInternalRead(AtomSeq)

InternalWriteLem : LEMMA memory[RM].MInternalWrite(AtomSeq) AND
                         RPCBackw AND ClerkBackw AND RelWrite
                         IMPLIES memory[M].MInternalWrite(AtomSeq)

ClerkImpl : LEMMA CClerk AND CRPC AND CRelMem
                  IMPLIES memory[M].CMem

MemImpl : THEOREM (( spec(true,CClerk) //
                     spec(true,CRPC) ) //
                     spec(true,CRelMem) ) =>
                                   spec(true,memory[M].CMem)
END clerkcor
```

Besides the lemmas shown above, 8 auxiliary lemmas have been introduced to split up the proof in small parts. The proofs of all lemmas and the final theorem required approximately 500 commands in the PVS prover.

4 Implementing the RPC Component

Next we consider a lossy RPC component, which is specified informally using timing information. To express these real-time properties, theory `procinterface` is extended with the type `Time` which is equal to the built-in type `real`. Further the timing of events is defined as an attribute.

```
Time  : TYPE = real
time  : [Event -> Time]
```

4.1 A Lossy RPC

We specify a lossy RPC component, with component number `LRPC`, which relays procedures to component `N`. First we specify the values returned in `LRPCRetVal` and the treatment of an incorrect call. The main differences with theory `rpc` are that `lrpc` does not raise the `RPCFailure` exception, a call need not lead to a return, and there is some timing information. We start with a specification which is close to the informal description.

```
lrpc [ LRPC, N : nat ] : THEORY
BEGIN
IMPORTING rpc[LRPC,N]

delta       : Time
```

```
NoRPCFailure : bool = (FORALL E :
   comp(E) = LRPC AND etype(E) = return
        IMPLIES argu(E) /= (:RPCFailure:))

BadCallreturn : bool = (FORALL E1, E2 :
   proced(E1,E2) AND comp(E1) = LRPC AND argu(E2) = (:BadCall:)
        IMPLIES time(E2) < time(E1) + delta)

CallReturnTiming : bool = (FORALL E1, E2, E3, E4 :
   proced(E1,E2) AND comp(E1) = LRPC AND
   proced(E3,E4) AND comp(E3) = N AND ident(E1) = ident(E3) AND
   name(E3) = namearg(E1) AND argu(E3) = argarg(E1) AND
   E1 < E3 AND E4 < E2
    IMPLIES time(E3) < time(E1) + delta AND
    (argu(E2) /= (:BadCall:) IMPLIES time(E2) < time(E4) + delta))

CLRPC : bool = RPCRelay AND RPCBackw AND NoRPCFailure AND
               BadCallreturn AND CallReturnTiming
```

Next we show that `CLRPC` implies a simplified relay property `LRPCRelay`.

```
LRPCRelay : bool = (FORALL E1, E2 :
  proced(E1,E2) AND comp(E1) = LRPC IMPLIES
      IF NOT CorrectArg(E1)
      THEN (argu(E2) = (:BadCall:) AND NoCallsBetween(E1,E2))
      ELSE (EXISTS E3, E4 : E1 < E3 AND E4 < E2 AND
             proced(E3,E4) AND comp(E3) = N AND
             ident(E1) = ident(E3) AND
             name(E3) = namearg(E1) AND argu(E3) = argarg(E1) AND
             NoOtherCall(E1,E2,E3) AND argu(E2) = argu(E4) )
      ENDIF )

LRPCExplLem : LEMMA CLRPC IMPLIES LRPCRelay
END lrpc
```

4.2 The RPC Implementation

An RPC component is implemented by means of a lossy RPC component using an `rpcclerk` component (see figure 2).

In the specification of `rpcclerk`, property `RPCClerkReturn` asserts it contains a single procedure `RemoteCall` and that a call to `RPC` always leads to a return. By property `RPCClerkRelay` we express that this `rpcclerk` component relays remote procedure calls to the lossy RPC component `LRPC` and, as expressed by `RPCClerkReturn`, it relays return values back to the caller, raising an `RPCFailure` if there is no return before a certain deadline. Since the value of this deadline is not yet determined, it is represented by a timing constant `IDel`.

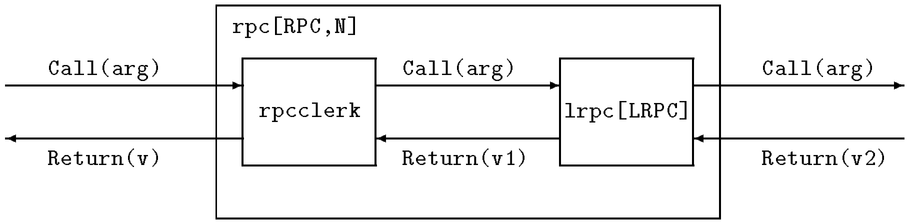

Fig. 2. Implementation of the RPC component

```
rpcclerk [ RPC, LRPC : nat ] : THEORY
BEGIN
IMPORTING rpcbasic

RPCClerkReturn : bool = (FORALL E1 :
  etype(E1) = call AND comp(E1) = RPC
    IMPLIES name(E1) = RemoteCall AND (EXISTS E2 : proced(E1,E2)))

IDel : Time

RPCClerkReturn(E2,E3) : bool =
   (EXISTS E4 : proced(E3,E4) AND time(E4) < time(E3) + IDel AND
                argu(E2) = argu(E4) AND E4 < E2)
   OR
   (NOT (EXISTS E4 : proced(E3,E4) AND
                     time(E4) < time(E3) + IDel) AND
                     argu(E2) = (:RPCFailure:))

NoOtherLRPCCall(E1,E2,E3) : bool =
  NOT (EXISTS E4 : E4 /= E3 AND etype(E4) = call AND
               comp(E4) = LRPC AND E1 < E4 AND E4 < E2 AND
               ident(E3) = ident(E4) AND rpcarg(E3) = rpcarg(E4))

RPCClerkRelay : bool = (FORALL E1, E2 :
  proced(E1,E2) AND comp(E1) = RPC
    IMPLIES (EXISTS E3 : E1 < E3 AND etype(E3) = call AND
                    comp(E3) = LRPC AND ident(E1) = ident(E3) AND
                    rpcarg(E1) = rpcarg(E3) AND
                    RPCClerkReturn(E2,E3) AND
                    NoOtherLRPCCall(E1,E2,E3)))

RPCClerkBackw : bool = (FORALL E3, E4 :
   proced(E3,E4) AND comp(E3) = LRPC
     IMPLIES (EXISTS E1 , E2 : E1 < E3 AND E4 < E2 AND
```

```
                proced(E1,E2) AND comp(E1) = RPC AND
                ident(E1) = ident(E3) AND
                rpcarg(E1) = rpcarg(E3) AND argu(E2) = argu(E4)))

CRPCClerk : bool = RPCClerkReturn AND RPCClerkRelay AND
                   RPCClerkBackw
END rpcclerk
```

Correctness of the RPC Implementation In theory `rpcclerkcor` we verify that the parallel composition of the `rpcclerk` component and the lossy RPC component implements the RPC component under certain conditions. One of these conditions is that every call of a procedure of component `N` returns within `epsilon` time units, as expressed in `CallReturnProp`. This is needed for the proof of `RPCRelay`, where also a condition on `IDel` is needed. The complete proof of all lemmas and the resulting theorem required approximately 250 user interactions.

```
rpcclerkcor  : THEORY
BEGIN
RPC, LRPC, N  : nat
IMPORTING rpcclerk[RPC,LRPC], lrpc[LRPC,N], rpc[RPC,N]

epsilon : Time

CallReturnProp : bool = (FORALL E1 :
  etype(E1) = call AND comp(E1) = N
   IMPLIES
    (EXISTS E2 : proced(E1,E2) AND time(E2) < time(E1) + epsilon))

ImplRPCReturn : LEMMA RPCClerkReturn IMPLIES rpc[RPC,N].RPCReturn

ImplRPCBackw : LEMMA RPCClerkBackw AND rpc[LRPC,N].RPCBackw
                     IMPLIES rpc[RPC,N].RPCBackw

ImplRPCRelay : LEMMA RPCClerkRelay AND LRPCRelay AND
                     rpc[LRPC,N].RPCBackw AND
                     RPCClerkBackw AND CallReturnTiming AND
                     CallReturnProp AND RPCClerkReturn AND
                     IDel >= 2 * delta + epsilon
                     IMPLIES rpc[RPC,N].RPCRelay

ImplRPCImpl : LEMMA IDel >= 2 * delta + epsilon AND CallReturnProp
                    AND CRPCClerk AND CLRPC
                    IMPLIES rpc[RPC,N].CRPC
```

```
RPCImpl : THEOREM IDel >= 2 * delta + epsilon AND CallReturnProp
                  IMPLIES
                  ( spec(true,CRPCClerk) // spec(true,CLRPC) )
                   =>
                   spec(true,rpc[RPC,N].CRPC)
END rpcclerkcor
```

5 Concluding Remarks

Using the tool PVS, all components of the RPC-Memory Specification Problem have been specified and the implementations of the memory component and the RPC component have been verified mechanically. An important aspect of such a formalization is that it reveals ambiguities in the informal specification. For instance, in the current example the treatment of exceptions is not always clear:

- For the memory component it is not clear whether a `MemFailure` can be returned if the argument is not correct. We decided to allow this, motivated by the implementation of section 3.2.
- In the clerk component we extended the informal description by transforming any `BadCall` exception from the RPC component into a `BadArg` return to the caller.

Further, as usual when writing an assertional specification, one also has to state explicitly that spurious events are not allowed. For instance:

- We have assumed that for any internal atomic action there is a causally related procedure which does not raise the `BadArg` exception.
- In the `rpc` component we have added the property that no call is issued in case of a `BadCall`. Similarly, we require that any procedure of `N` is causally related to a corresponding procedure of `RPC`.

Also implicit in the informal specification are the properties if the inner memory (e.g., the meaning of atomicity).

In the RPC-Memory example only a small part of our real-time framework has been used. In [Hoo94a] we show how PVS can be used during the design of real-time systems by an example of the top-down derivation of a distributed real-time control system (a chemical batch processing system). Another example of the formalization of an informal specification can be found in [Hoo95] where part of an ACCESS.bus protocol, connecting a host computer with a dynamically changing number of devices, has been specified and verified.

Acknowledgments

We are grateful to Job Zwiers for the suggestion to use event attributes, thus simplifying our initial solution. The reviewers are thanked for valuable comments

which have led to several improvements. Finally, many thanks goes to the participants of the Dagstuhl seminar on the RPC-Memory Specification Problem for many interesting discussions which have significantly contributed to the current version of our solution.

References

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer-Verlag, 1991.

[Hoo94a] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, 1994.

[Hoo94b] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[JPZ94] W. Janssen, M. Poel, and J. Zwiers. The compositional approach to sequential consistency and lazy caching. Memoranda informatica 94-37, University of Twente, 1994.

[Lam94] L. Lamport. Verification and specification of concurrent programs. In *A Decade of Concurrency*, pages 347–374. LNCS 803, 1994.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.