COMPUTING LABORATORY

Proving Correctness Properties of a Replicated Synchronous Program

L.V. Mancini and G. Pappalardo

TECHNICAL REPORT SERIES
No 266  September, 1988
Proving Correctness Properties of a Replicated Synchronous System

L.V. Mancini and G. Pappalardo

This paper is to appear in the Computer Journal

Abstract

Replicated synchronous programs executed by a set of possibly faulty processors find a natural application in the fault-tolerance technique known as N-Modular Redundancy (NMR). We present an axiomatic approach to correctness proofs for replicated synchronous programs and illustrate its application by studying an agreement protocol for NMR distributed computing.
Bibliographical details

MANCINI, Luigi Vincenzo

Proving correctness properties of a replicated synchronous program. [By] L.V. Mancini and G. Pappalardo

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1988.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 266)

Added entries

PAPPALARDO, Giuseppe
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 266

Abstract

Replicated synchronous programs executed by a set of possibly faulty processors find a natural application in the fault-tolerance technique known as N-Modular Redundancy (NMR). We present an axiomatic approach to correctness proofs for replicated synchronous programs and illustrate its application by studying an agreement protocol for NMR distributed computing.

About the author

Mr. L.V. Mancini has been at the Computing Laboratory since May, 1985 as a Research Associate.

Mr. G. Pappalardo has been at the Computing Laboratory since April, 1986 as a Research Associate.

Suggested keywords

AXIOMATIC PROOFS MAJORITY VOTING
DISTRIBUTED AGREEMENT MODULAR REDUNDANCY
DISTRIBUTED SYSTEMS RELIABILITY
FAULT TOLERANCE REPLICATED PROCESSING
FORMAL CORRECTNESS SYNCHRONOUS PROGRAMS
HOARE LOGIC

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.64404
U.D.C. 681.322.06 519.687
Proving Correctness Properties of a Replicated Synchronous Program

L. V. Mancini$^{1,2}$ and G. Pappalardo$^{1,3}$

(1) Computing Laboratory - University of Newcastle upon Tyne
(2) Dipartimento di Informatica, Università di Pisa, Italy
(3) Università di Reggio Calabria, Italy

ABSTRACT

Replicated synchronous programs executed by a set of possibly faulty processors find a natural application in the fault-tolerance technique known as N-Modular Redundancy (NMR). We present an axiomatic approach to correctness proofs for replicated synchronous programs and illustrate its application by studying an agreement protocol for NMR distributed computing.

Index Terms: synchronous programs, Hoare logic, formal correctness, axiomatic proofs, fault tolerance, reliability, modular redundancy, distributed agreement, majority voting, replicated processing, distributed systems.
1. Introduction

Synchronous distributed programming is enjoying a growing interest thanks to its conceptual simplicity and to the appealing complexity bounds that it can attain. Theoretical investigation in this area has mainly centred upon computation models (e.g. PRAMs and bounded-degree networks(18)), but little work has been done on program verification.

Synchronous distributed programs may find an important application in fault-tolerant distributed computing (7,20), and it is especially in this field that the lack of a verification technique is apt to be regretted, for correctness is threatened by faulty hardware as well as by programming errors.

In this paper we adapt Hoare logic (4), the classical technique for proving sequential programs correct, to a class of synchronous replicated programs. We have been encouraged to do so because we have found that for the application considered extant verification techniques for parallel programming are unnecessarily complex, in that they are intended to cope with arbitrary relative speeds of the processors; our technique, instead, exploits the advantage of dealing with synchronous processors and therefore affords simpler proofs.

As a case study we shall consider Join, a fault-tolerant synchronous replicated algorithm for N-modular redundant distributed systems. N-Modular Redundancy (NMR), which is one of the best-known fault-tolerance techniques for centralized processing, can also be applied to distributed systems (1,2,7,17,19,20). Such systems are networks of nodes, each made up of N voter-task combinations in a classical NMR configuration and connected to the others through redundant communication channels. As a means of preventing the so-called sequencing failure (11), i.e. a failure of the tasks within a node to process input requests in the same order, we propose a fully decentralized synchronous algorithm called Join. Join is to be executed by the modules of a NMR node and achieves its objective under the only assumption that a majority of the modules are non-faulty, thus tolerating arbitrary behaviour of the faulty ones. Note that Join relies on authentication techniques less heavily than the agreement algorithm considered in (10), which results in quite a different approach to the formal correctness proof.

The paper is structured as follows. In Section 2 we discuss a distributed architecture for replicated processing; in Section 3 the Join algorithm is introduced and explained informally; Section 4 shows the algorithm at work through an example; Section 5 presents an axiomatic approach to correctness proofs for (simple) replicated synchronous programs, and demonstrates it in a formal proof of the correctness of Join.
2. An Architecture for Replicated Processing

The system under consideration can be depicted as an arbitrary directed graph of NMR nodes connected by redundant (logical) links. Each node consists of $N$ modules; if an edge in the graph leads from node $N_m$ to node $N_n$, then every module of $N_m$ is connected to every module of $N_n$, so that there are $N^2$ links between the two nodes (Fig. 1); messages exchanged over a link are unique. The generic module $k$ ($1 \leq k \leq N$) contains a voter-task combination $V_k$, $T_k$. The function of $V_k$ is to perform majority voting on incoming messages and enqueue them in the voted message queue $vmq_k$. The function of $T_k$ is to pick messages from $vmq_k$, process them and forward $N$ copies of the result to the NMR nodes attached to $T_k$; we assume that each $T_k$ maintains some state information determined by its past input and affects its subsequent behaviour.

![System architecture for $N=3$](image)

In the discussion below we regard the links departing from a task as part of the task; thus a failure of $T_1$ of $N_m$ in Fig. 1 can be a failure of $T_1$ proper or a communication failure affecting any of the three links leaving it.

A correct NMR system must behave like its non-replicated, non-faulty counterpart, tolerating as many as $LN/2J$ faulty modules per node. We claim that a sufficient condition for this is that every node in the network satisfies the local correctness criterion LOC below.

**LOC** If input to a given node satisfies EVN, then the outputs from at least $\lceil N/2 \rceil$ modules in the node are equal and are correct with respect to the input.

where

**EVN** If a message $m$ is received by a given node on at least $\lceil N/2 \rceil$ of the links directed to a module, then $m$ will eventually be received on at least $\lceil N/2 \rceil$ of the links directed to every module in the node.
The above claim can be proved by induction on the structure of the network, but we shall only try to justify it with the help of Fig. 1: assume that $T_1$ and $T_2$ of $N_m$ generate a correct output, then all voters of $N_n$ will receive at least two copies of a correct message and at most one copy of a wrong one (from $T_3$), i.e. input to $N_n$ is correct and satisfies EVN; thus if $N_n$ satisfies LOC, at least two modules of $N_n$ will issue a correct output; the same argument may now be invoked to show that nodes fed by $N_n$ (and by nodes like it) will also generate a correct output.

Let us now consider how a generic NMR node may fulfill LOC. If at least $\lceil N/2 \rceil$ modules in the node are non-faulty and if EVN holds, then at least $\lceil N/2 \rceil$ voters will be able to construct the same set of messages for their non-faulty tasks; yet, this is not sufficient for the tasks to generate the same output, as required by LOC; since the tasks are devices with a state, in order that they generate the same output they must also process voted messages in the same order. This condition will henceforward be termed $\text{SEQ}$.

In a given node all tasks of non-faulty modules process input messages in the same order and violations of it will be called sequencing failures. In the sequel it is shown that if input to a NMR node with at least $\lceil N/2 \rceil$ non-faulty modules satisfies EVN, then the node can satisfy $\text{SEQ}$ by executing the $Join$ algorithm; as discussed above, this entails that the node satisfies the local correctness criterion LOC and that networks of such nodes are correct.

The condition $\text{SEQ}$ is particularly hard to meet in a distributed processing environment owing to the non-determinacy arising from concurrency, as it can be appreciated from the following example. Let $N_n$ be a NMR node which receives results from two different nodes $N_m$ and $N_1$ (Fig. 2). Suppose that $N_m$ and $N_1$ send their messages at about the same time to $N_n$, and that messages can experience variable delays during transmission. It is thus possible that voters of $N_n$ receive messages in different orders: this will cause voted messages to be enqueued in different orders in the $\text{vmq}$ queues of $N_n$.

![Fig. 2.](image-url)
3. The Join Algorithm

In order to ensure the condition \texttt{SEQ} given \texttt{EVN}, the modules of a NMR node must agree by a distributed algorithm on the order in which messages in the \texttt{vmq} queues will be processed. This algorithm must be reliable, and guarantee the agreement also in the presence of components subject to arbitrary faults; for this purpose the Join algorithm proposed here exploits the signed message algorithm for interactive consistency of [5]. Join is better specified by refining the module structure of Fig. 1 as in Fig. 3 (of course, the implementation may in practice be widely different). Thus we assume that each module contains a Join unit, which extracts messages from \texttt{vmq} and appends them, in such a way that \texttt{SEQ} is respected, to the queue \texttt{omq} that feeds the task \( T \).

![Fig. 3. A refined module structure](image)

3.1 Data Types

In this section some notation and concepts are introduced for use both in the presentation of the algorithm and later in the formal correctness proof upon which this paper is centred: here, instead, the level of formality is deliberately kept to a minimum. The message queues \texttt{vmq} and \texttt{omq} are seen as global variables by procedure Join. We assume that messages have the form \(<id,info>\), where \( id \) is a unique identifier, so that, as mentioned above, messages received by a module and inserted in its queues are unique; it thus makes sense to extend to message queues some set operations. The following notation is used in the sequel:

- \([\ ]\) the empty queue,
- \([m]\) the queue consisting of the single message \( m \),
- \( m \in q \) true iff message \( m \) is contained in the queue \( q \),
- \( q_1 \subseteq q_2 \) true iff \( m \in q_1 \) implies \( m \in q_2 \) (\( q_1 \) and \( q_2 \) may be sets or queues),
- \( q_1 + q_2 \) the concatenation (postfixing) of queue \( q_2 \) to \( q_1 \),
- \( q_1 - q_2 \) the queue obtained from \( q_1 \) by deleting the messages that are in \( q_2 \).
The local variables used by Join are:

\begin{verbatim}
var myids, cids: set of identifier;
  ids: array[1..N] of set of identifier; \quad \{N is the degree of replication\}
\end{verbatim}

We let

\begin{align*}
identifiers(q) &= \{id | \langle id,info \rangle \in q\} \\
union(ids) &= \bigcup_{n=1}^{N} ids[n] \\
msgin(id,q) &= \text{TRUE} \quad \text{iff there is a message } m = \langle id,info \rangle \text{ such that } m \in q
\end{align*}

and, assuming a total order is defined over identifiers,

\begin{verbatim}
messages(idset,q) = [ ] if idset = \emptyset; otherwise:
messages(idset,q) =
  let mid = max(idset) in
  if \langle mid,info \rangle \in q \text{ then } [\langle mid,info \rangle] + messages(idset - \{mid\},q)
  \quad \text{else } messages(idset - \{mid\},q)
\end{verbatim}

I.e., \textit{messages(idset,q)} is the queue formed by the messages of \textit{q} whose identifiers are in \textit{idset}, inserted in a position that respects the order over identifiers.

3.2 The Program

Procedure Join is started at given time intervals by all non-faulty processors of a NMR node simultaneously and is carried out synchronously, i.e. every statement is executed by all non-faulty Join units at the same time, which is determined by a planned scheduling; this implies that non-faulty modules must have synchronized clocks.

Procedure Join is shown in Fig. 4 and commented below (a more complete Pascal specification can be found in [9]).

1: myids := identifiers(umq);
2: InteractiveConsistency(myids,ids);
3: cids := union(ids);
4: Wait;
5: Transfer(umq,omq,cids)

Fig. 4. The Join procedure

1. The set identifiers(umq) of the identifiers of the messages in umq is assigned to myids.
2. Each unit engages with the others in a byzantine agreement upon the value of myids. The version of byzantine agreement used, being based on signatures [16] is guaranteed to succeed if there is at least one non-faulty unit in the node (we assume that communication between
units satisfies all the requirements laid down in [5] for consensus to be reached). By the agreement, all non-faulty units compute exactly the same array \( ids \) and, if unit \( n \) is non-faulty, then \( ids[n] \) is \( n \)'s local value of \( myids \), i.e. the set of message identifiers received by \( n \).

(3) The set \( \text{union}(ids) \) is computed and assigned to \( cids \). It will be the same for all non-faulty units, like the array \( ids \) from which it is computed.

(4) A timer is started which expires after an interval equal to the maximum delay \( \Delta \) that messages directed to the node can experience. Let \( n \) be a non-faulty unit and \( id \) the generic identifier in its \( cids \); then either (a) \( id \) is also in \( ids[n] = myids \), or \( id \) is not in \( ids[n] \) but in some \( ids[k] \) for \( k \neq n \); if so, \( k \) may be either (b) non-faulty or (c) faulty. In the cases (a) and (b), a message \( m \) of identifier \( id \) must be in all the \( umqs \) once \( \Delta \) has elapsed; this is because \( m \) was in at least a \( vmq \) before the timer started, it must therefore reach all modules in a majority of copies (by EVN), and this must happen within \( \Delta \), which is an upper bound on communication delays. As to case (c) we make the assumption (HOPE) that when step 5 starts either no \( vmq \) or all \( vmqs \) will contain a message of identifier \( id \).

(5) The messages whose identifiers are in \( cids \) are removed from \( vmq \) and appended to \( omq \) in accordance with the order defined over identifiers. From (3) above we know that \( cids \) is the same for all non-faulty units; from (4) we know that (whichever of the cases (a), (b) and (c) applies) if an identifier is in \( cids \), then the corresponding message will be either in the \( vmq \) of all non-faulty modules or in none's when step 5 starts. This ensures that all non-faulty units transfer to \( omq \) the same messages in the same order, as required by SEQ. Some care is needed about the assumption made under (4) for case (c), i.e.

**HOPE** If \( id \) is contributed to \( cids \) by a faulty unit, then when the Transfer step starts either all non-faulty units have a message of identifier \( id \) in their \( vmq \) or none has it.

Indeed, a faulty unit need not cheat about everything; if an \( id \) in \( cids \) originates from a faulty unit, it may well be that the unit has actually received a message that has not yet reached any non-faulty unit. If so, for the same reasons as in (a) and (b), the message will be in every non-faulty module's \( vmq \) when Transfer starts, which is consistent with HOPE. On the other hand, a faulty unit could contribute a forged identifier \( id \) to \( cids \), but it is very unlikely, though not impossible, that, when Transfer starts, a message carrying the forged \( id \) has been received only by some of the non-faulty modules. HOPE amounts exactly to ruling out this possibility.

A NMR system has anyway a certain failure probability, whose increase due to the chance that HOPE is violated is likely to be so negligible to be justifiably ignored. It is not difficult, however, to cope with this chance; two possible approaches are: (i) taking the majority of array \( ids \), rather than the union, or (ii) requiring sender modules to authenticate identifiers [10], so that faulty modules within receiving nodes cannot forge identifiers when performing Join. The tradeoff involved in the choice of one of these solutions is better understood by observing that units
engaged in Join may be viewed as byzantine generals trying to decide which messages to pass to processing tasks. Once constructed the array ids, they should take the decision by majority voting; if they instead take the risk of passing all the identifiers in ids, they are trading correctness for efficiency: a choice that performance evaluation might indeed prove convenient!

4. The Algorithm in Action

In order to clarify the algorithm of the previous section, we shall observe it at work in a concrete situation. We shall assume the topology of Fig. 2 and a degree of replication equal to three as in Fig.1; we also assume that message identifiers are pairs <node, n> specifying that a message is the n-th sent by node.

Let the three queues umq of Nn be empty at time t0 (Fig. 4.a) and N1 and Nm issue the messages of identifiers (l,1), (l,2), and (m,1) respectively between t0 and t1, the instant at which Nn starts performing the Join algorithm. It should be recalled that owing to different communication delays: (i) a queue may contain messages which another queue has not yet received, and (ii) different queues may receive messages in a different order. Thus the situation of the umqs at t1 may well be that depicted in Fig. 4.b.

At t1 the Join algorithm is simultaneously started by the three Join units of node Nn: each unit computes the set myids of its identifiers, and exchanges myids with the other units, recording unit n's value of myids in ids[n] (procedure InteractiveConsistency). Assuming that only unit 2 is faulty, the byzantine agreement used ensures that units 1 and 3 compute the same arrays ids, e.g. those in Fig. 4.c; note that ids[2] has taken the fictitious value (0,5) in the non-faulty units 1 and 3 owing to a particularly malicious behaviour of the faulty unit 2, but more likely ids[2] will simply be empty. The units compute then the sets cids shown in Fig 4.d.

At time t2 a timer is set; on its expiry after Δ, the maximum communication delay, unit 1 will have received and voted the message of identifier (m,1) and unit 3 will have received and voted the message of identifier (l,2). Thus, at this stage each non-faulty unit holds in its queue umq a voted copy of the message corresponding to each correct identifier in cids (Fig. 4.e).Transferring messages from umq to omq according to the fixed priority l<m will yield the queues omq shown in Fig. 4.f and ensure that messages sent to node Nn are processed in the same order by its non-faulty modules 1 and 3.
Fig. 4. Join in action
5. Proving the Join Algorithm Correct

5.1 A Hoare Logic for Replicated Synchronous Programs

We shall now introduce a generalization of Hoare logic for replicated synchronous computations in the presence of faults. Such computations are performed by $N$ processes, of which $N_f$ are non-faulty and execute synchronously the same program $s$, whereas the other $N-N_f$ are faulty and may exhibit arbitrary behaviour; $s$, which must be of the form $s_1s_2\ldots s_n$ where the $s_i$s are either assignments or procedure calls, is said to be executed synchronously if none of the non-faulty processes starts to perform $s_i + 1$ before the others have finished $s_i$. For short, $s$ will be referred to as simple replicated synchronous program (SRSP), although synchronization and replication are features of the execution model, not of the program itself.

We assume that a fault predicate non-faulty is true for $1 \leq n \leq N$ if process $n$ is non-faulty during the entire execution of a SRSP $s$. Let $x, y, \ldots$ be the variables occurring in $s$; each non-faulty process will have its own copy of each variable; the copy is denoted by subscripts; e.g. $x_n$ is process $n$'s copy of $x$. A state of the SRSP $s$, given a fault predicate, is an assignment of values from the appropriate domains to the variables of non-faulty processes, i.e. a tuple

\[
< x_{n_1} = v_{1}, \ldots, x_{n_{N_f}} = v_{N_f} \\
y_{n_1} = v_{1}, \ldots, y_{n_{N_f}} = v_{N_f} \\
\ldots \\
>
\]

where non-faulty($n_i$) holds for $1 \leq i \leq N_f$.

An assertion for a SRSP is a formula of a first-order calculus whose lexicon, in addition to suitable function and predicate symbols, contains non-faulty and the variables $x_n, \ldots, y_n, \ldots$ for values of $n$ such that non-faulty($n$) is true. Assertions about a SRSP are evaluated on its states.

The notation $(P) s \{Q\}$, where $s$ is a SRSP and $P$ and $Q$ are assertions, has the following interpretation: if execution of $s$ starts in a state satisfying $P$ (called the precondition) and if $s$ terminates, it will do so in a state satisfying $Q$ (the postcondition). Thanks to the simplified syntax of SRSPs, a proof system for reasoning about them must only contain an axiom for assignment, a rule for sequential composition and the usual consequence rules; these are to be employed in conjunction with the definitions of procedure calls semantics in terms of pre- and postconditions.

The SRSP proof system employed in the sequel consists of an axiom and two inference rules (whose premiss and conclusion are separated by a horizontal line).

(as) assignment axiom \[\{P(e_m, e_n, \ldots)\} x = e \{P(x_m, x_n, \ldots)\}\]

where $x_m, x_n, \ldots$ are all subscripted instances of $x$ occurring free in $P$, no such instance is bound in $P$ and $e_n$ is obtained from $e$ by subscripting its variables with $n$. 

- 9 -
(sq)  sequential composition rule
\[
\frac{\{P\} s_1 \{Q\}, \{Q\} s_2 \{R\}}{\{P\} s_1 s_2 \{R\}}
\]

(cn) consequence rules
\[
\frac{P \rightarrow Q, \{Q\} s \{R\}}{\{P\} s \{R\}} \quad \frac{\{P\} s \{Q\}, Q \rightarrow R}{\{P\} s \{R\}}
\]

5.2 The Join algorithm as a SRSP

In the formal proof of the next section we shall assume that the variable omq is never read by the processing tasks $T_1$ (see Fig. 3), which makes it equal to the output history from the Join units; this makes it easier to state the requirement SEQ formally, i.e. that all these histories have to be the same. A purist’s approach would have been to give the procedure Transfer below a fourth auxiliary parameter that would be equal to omq but would not be shared with any external process. More care is needed about the other shared variable of Join, viz. vmq, to which also voters $V_1$ have access (Fig. 3). After obtaining a proof within the above proof system, as though Join were executed in isolation, it will be necessary to check that no assertion used in the proof and containing vmq is invalidated by the concurrent activity of the voters. (This is the well-known non-interference criterion for proofs about concurrent programs introduced in [13]).

The semantics of the procedures called by Join is defined below. Note that "\rightarrow" denotes logical implication. Note also that both here and, later, in the proof subscripts are often implicitly assumed to take only legal values, i.e. integers in the range $1..N$ that correspond to non-faulty processes. E.g. we may write $A(x_n)$ instead of $\forall n. \text{non-faulty}(n) \rightarrow A(x_n)$ and omit the range of $k$ in $\bigcup_k V_k$.

\[
\text{(ic)} \quad \{ \text{myids}_n = MY_n \land I \}
\]
InteractiveConsistency(myids,ids)
\[
\begin{align*}
\{ & (\forall k. \text{non-faulty}(k) \rightarrow \text{ids}_n[k] = MY_k) \\
& (\forall k. 1 \leq k \leq N \rightarrow \text{ids}_n[k] = ID_k) \\
& I \}
\end{align*}
\]
where no subscripted instances of ids occur free in I.

\[
\text{(wt)} \quad \{ V_n \subseteq \text{vmq}_n \land \text{cids}_n = \bigcup_k \text{identifiers}(V_k) \cup \text{OTHER\_IDS} \\
& \bigcup_k \text{identifiers}(V_k) \cap \text{OTHER\_IDS} = \emptyset \\
& I \}
\]
Wait
\[
\begin{align*}
\{ & \bigcup_k V_k \subseteq \text{vmq}_n \\
& (id \in \text{OTHER\_IDS} \land \text{msgin}(id, \text{vmq}_n) \rightarrow \text{msgin}(id, \text{vmq}_k)) \\
& I \}
\end{align*}
\]
(tr) \{P(omq_n + messages(cids_n, vmq_n), \text{vmq}_n - messages(cids_n, vmq_n))\}
Transfer(vmq, omq, cids)
\{P(omq_n, \text{vmq}_n)\}

It should be noted that a suitable invariant (viz. I) has been included in (ic) and that formal parameters have been given the same name as the actual parameters occurring in the calls made by Join, which allows the above definitions to be used in the proof of next section without further manipulations.

(ic) and (tr) are straightforward formalizations of the behaviour of the corresponding procedures, which have been described in Section 3.2. Concerning (wt), note that it captures both property EVN (Section 2) and assumption HOPE (Section 3.2) but does not specify the time interval after which Wait returns; however, this deficiency is apt to be encountered in most formal treatments, for only quite outlandish program semantics cater for the specification of time intervals.

5.3 The Formal Proof

There are two correctness requirements for the Join algorithm:

SEQ The output histories issued by non-faulty modules should be all equal;

LIV Any message present in a voted message queue of a non-faulty module before Join starts should be ready for the task when Join has terminated.

Note that SEQ is a safety property: it states that nothing bad happens (viz. that the output histories never differ), but it does not ensure that something good happens: the output histories might all be null and still satisfy SEQ. A liveness properties like LIV is therefore needed to guarantee that actual progress does take place. Liveness properties occurring in the literature are usually termination or requirements that something good should eventually happen. The latter must be treated outside Hoare logic, e.g. by temporal logic [14]. In our running example, though, termination follows straightforwardly, once it is assumed for the individual statements, so we shall not deal with it. Besides, something good has to happen not "eventually" at some indefinite time, but exactly on termination of Join, as stated in LIV, which makes recourse to temporal logic unnecessary.

Formally SEQ and LIV can be expressed as

{HP: V_n \subseteq \text{vmq}_n \land \text{omq}_n = O}
Join
{TH: \(\forall m. \ m \in \bigcup_k V_k \rightarrow m \notin \text{vmq}_n\)}
\land \text{omq}_n = O + \text{NewO} \land \text{NewO} \supseteq \bigcup_k V_k}

In the usual style of axiomatic proofs for sequential programs [3], we give below a proof outline (assertions are tagged with a label for further reference)
{HP: \( V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \) }
myids: = \text{identifiers(\text{vmq})};
{A: \( \text{myids}_n = \text{MY}_n \land \text{identifiers}(V_n) \subseteq \text{MY}_n \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \) }
InteractiveConsistency(myids,ids);
{IC: \( (\forall k. \text{non-faulty}(k) \rightarrow \text{id}_n[k] = \text{MY}_k) \land (\forall k. 1 \leq k \leq N \rightarrow \text{id}_n[k] = \text{ID}_k) \land \text{identifiers}(V_n) \subseteq \text{MY}_n \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \) }
cids: = \text{Union(ids)};
{U: \( \text{cids}_n = \cup_k \text{identifiers}(V_k) \cup \text{OTHER} \_ \text{IDS}_n \land \cup_k \text{identifiers}(V_k) \cap \text{OTHER} \_ \text{IDS}_n = \emptyset \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \) }
Wait;
{W: \( (\forall m. m \in \cup_k V_k \rightarrow m \in \text{vmq}_n \land m \in \text{messages(\text{cids}_n,\text{vmq}_n)}) \land \text{messages(\text{cids}_n,\text{vmq}_n)} = \text{NewO} \land \text{omq}_n = O \) }
Transfer(\text{vmq,omq,cids});
{TH: \( (\forall m. m \in \cup_k V_k \rightarrow m \notin \text{vmq}_n) \land \text{omq}_n = O + \text{NewO} \land \text{NewO} \supseteq \cup_k V_k \) }

The formal proof that TH follows from HP is worked out below through a series of steps aimed at progressively establishing that intermediate assertions used in the proof outline hold after termination of the statements preceding them; in doing so we shall implicitly be applying the rules for sequential composition (sq) and consequence (cn). Each step is numbered: those involving program statements are justified by references to Sections 5.1 and 5.2; those that are "pure" logical implications are proved valid not within a proof system (which would be an exercise of little consequence) but rather by (metalinguistic) reasoning on their validity in the assumed interpretation, which includes the functions and predicates introduced in Section 3.1; in other words, to prove that A → B is valid we prove that if A is valid then also B is valid.

(0.1) \( \text{HP} \rightarrow \text{HP1} \) where
\( \text{HP1: } \text{identifiers}(V_n) \subseteq \text{identifiers(\text{vmq}_n)} \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \) 

(1) \( \text{(HP1)} \) by (as)
\( \text{myids: = identifiers(\text{vmq})} \)
\( \{A0: \text{identifiers}(V_n) \subseteq \text{myids}_n \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \} \)

(1.1) \( \text{A0} \rightarrow \text{A} \) where
\( \text{A: } \text{myids}_n = \text{MY}_n \land \text{identifiers}(V_n) \subseteq \text{MY}_n \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = O \)
(2) \[ \text{InteractiveConsistency}(\text{myids}, \text{ids}) \] 
\( \{ \text{IC:} \) \( (\forall k. \text{non-faulty}(k) \rightarrow \text{ids}_n[k] = \text{MY}_k) \) 
\( \land (\forall 1 \leq k \leq N \rightarrow \text{ids}_n[k] = \text{ID}_k) \) 
\( \land \text{identifiers}(V_n) \subseteq \text{MY}_n \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = 0 \}\)

(2.1) \( \text{IC} \rightarrow \text{IC1} \) where

\text{IC1:} \( \text{union}(\text{ids}_n) = \bigcup_k \text{identifiers}(V_k) \cup \text{OTHER\_IDS} \) 
\( \land \bigcup_k \text{identifiers}(V_k) \cap \text{OTHER\_IDS} = \emptyset \) 
\( \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = 0 \)

**Proof:** It is sufficient to prove that the first two lines of IC1 are true if IC is. This follows from
(i) \( \bigcup_k \text{identifiers}(V_k) \subseteq \text{union}(\text{ids}_n) \) and
(ii) \( \text{union}(\text{ids}_n) \) is the same for all \( n \),
which follow from the first and second conjunct of IC respectively.

(3) \( \{ \text{IC1} \} \) \( \text{by (as)} \)
\( \text{cids:} = \text{union}(\text{ids}) \)
\( \{ \text{U:} \) \( \text{cids}_n = \bigcup_k \text{identifiers}(V_k) \cup \text{OTHER\_IDS} \) 
\( \land \bigcup_k \text{identifiers}(V_k) \cap \text{OTHER\_IDS} = \emptyset \) 
\( \land V_n \subseteq \text{vmq}_n \land \text{omq}_n = 0 \}\)

(3.1) \( \text{U = U1} \) where
\( \text{U1:} \) \( \text{U} \land \text{cids}_n = \bigcup_k \text{identifiers}(V_k) \cup \text{OTHER\_IDS} \) 
\( \land \bigcup_k \text{identifiers}(V_k) \cap \text{OTHER\_IDS} = \emptyset \)
(4) \{U1\} \text{ by (wt)}

Wait

\{W0: \ \cup_k V_k \subseteq \text{vmq}_n \}

\wedge (id \in \text{OTHER \_IDS} \land \text{msgin}(id,\text{vmq}_n)) \rightarrow \text{msgin}(id,\text{vmq}_k)

\wedge \text{cids}_n = \cup_k \text{identifiers}(V_k) \cup \text{OTHER \_IDS}

\wedge \cup_k \text{identifiers}(V_k) \cap \text{OTHER \_IDS} = \emptyset

\wedge \text{omq}_n = O\}

(4.1) \text{W0} \rightarrow \text{W} \text{ where}

\text{W:} \quad (\forall m. \ m \in \cup_k V_k \rightarrow

m \in \text{vmq}_n \land m \in \text{messages(cids}_n,\text{vmq}_n))

\wedge \text{messages(cids}_n,\text{vmq}_n) = \text{NewO} \land \text{omq}_n = O

Proof

first conjunct of W:

(a) \quad m = <id,info> \in V_k \text{ assumption of the conjunct}

(b) \quad m \in \text{vmq}_n \text{ for all } n \text{ by (a) and the first line of W0}

(c) \quad id \in \text{cids}_n \text{ for all } n \text{ by (a) and the third conjunct of W0}

(d) \quad m \in \text{messages(cids}_n,\text{vmq}_n) \text{ for all } n \text{ by (b) and (c)}

second conjunct of W (we prove that messages(cids}_n,\text{vmq}_n) \text{ is the same for all } n:

(a) \quad m = <id,info> \in \text{messages(cids}_h,\text{vmq}_h) \text{ for some } h \text{ assumption of the conjunct}

(b) \quad id \in \text{cids}_h \text{ by (a)}

(c) \quad id \in \text{cids}_n \text{ for all } n, \text{ and} \text{ by (b) and the third line of W0}

(e1) \quad \text{either id} \in \cup_k \text{identifiers}(V_k)

(d1) \quad m \in V_k \text{ for some } k \text{ by (c1)}

(e1) \quad m \in \text{vmq}_n \text{ for all } n \text{ by (d1) and the first line of W0}

(f1) \quad m \in \text{messages(cids}_n,\text{vmq}_n) \text{ for all } n \text{ by (c) and (e1)}

(c2) \quad \text{or id} \in \text{OTHER \_IDS}

(d2) \quad \text{msgin}(id,\text{vmq}_h) \text{ by (a)}

(e2) \quad \text{msgin}(id,\text{vmq}_n) \text{ for all } n \text{ by (c2), (d2) and the second line of W0}

End of Proof

(4.2) \text{W} \rightarrow \text{W1} \text{ where}

\text{W1:} \quad (\forall m. \ m \in \cup_k V_k \rightarrow m \notin \text{vmq}_n - \text{messages(cids}_n,\text{vmq}_n))

\wedge \text{omq}_n + \text{messages(cids}_n,\text{vmq}_n) = O + \text{NewO} \land \text{NewO} \supseteq \cup_k V_k
Transfer(vm, omq, cids)

\( \langle TH: \quad (\forall m. m \in \bigcup_k V_k \rightarrow m \notin vmq_n) \)
\( \wedge \quad omq_n = O + NewO \wedge NewO \supseteq \bigcup_k V_k \rangle \)

As stated previously, it should now be checked that voters do not violate the non-interference criterion. For this purpose it is sufficient to prove that in the assertions used in the proof no subformulae directly containing instances of omq are affected. It is easy to realize that, since voters add but do not remove messages from the omqs, the noted subformulae are all invariant to the voters' activity, except

\[ id \in OTHER\_IDS \wedge msgin(id, vmq_n) \rightarrow msgin(id, omq_k) \]

appearing in W0, for which, again, assumption HOPE of Section 3.2 has to be invoked.

6. Concluding Remarks

It is certainly out of the scope of this paper to evaluate Join as a solution for NMR distributed processing. The following consideration, however, may be worth making. Maintaining the modules of a NMR node in step may seem a burden; but in fact in most implementations, the modules are likely to be close enough to use the same physical clock. This should be contrasted with approaches that require even tighter synchronization, such as SIFT [20], where nodes are required to maintain their clocks synchronized.

Formal verification of NMR systems has already been considered [8,12] in the literature, but the present paper is, to our knowledge, the first attempt in a logical system that permits syntax-directed reasoning, i.e. one where properties of a program are inferred from those of its components. Temporal logic [14,15] and other logic systems [6] for parallel programming have been found unsuitable for synchronous replicated programming because of their assumption of independent processor speeds; indeed, this assumption is exactly what makes parallel programs and proofs inherently complex. Our approach, instead, attains greater simplicity by trying to reflect the greater simplicity of synchronous programming. But more research seems necessary, in the light of the excellent time performance of synchronous parallel algorithms and of the trend towards machine architectures for their execution. An open issue is e.g. the treatment of "general" synchronous programs that unlike our simple ones may contain arbitrary programming constructs.

Acknowledgements

The authors are grateful to Prof. B. Randell for his comments and suggestions. This work was supported by the Royal Signals and Radar Establishment of the U.K. Ministry of Defence.
References


