A PROOF METHODOLOGY FOR VERIFYING REQUEST PROCESSING PROTOCOLS

by

Christos N. Nikolaou
Edmund M. Clarke, Jr.
Stephan A. Schuman

TR-04-82

January, 1982

A Proof Methodology for Verifying Request Processing Protocols

Christos N. Nikolaou

Edmund M. Clarke, Jr.

Center for Research in Computing Technology

Harvard University

Stephen A. Schuman

Massachusetts Computer Associates, Inc.

At Harvard University, this research was supported in part by NSF Grant MCS-7908365 and by Contract N00039-78-G-0020 with the Naval Electronics Systems Command.

At Massachusetts Computer Associates, Inc., this research was supported in part by the U.S. Army CORADCOM, through the Scientific Services Program under Delivery Order No. 1704 from Battelle Columbus Laboratories.

Table of Contents

Abstract	1
1. Introduction	1
2. Sequential Request Processing Protocol	5
2.1. The overall system	5
2.2. the DRIVER	7
2.3. The Server	8
2.4. The INTERFACE	9
3. Description of the Formal Tools	11
3.1. History variables	12
3.2. A Proof System for Safety Properties	16
3.3. Using Temporal Logic to prove liveness properties	19
3.3.1. The strong fairness axiom	22
3.3.2. Proving liveness properties	25
4. Analysis of the SRP protocol	27
5. Conclusion	37
References	39

List of Figures

Figure 1-1:	A network of communicating Virtual Machines	2
Figure 2-1:	Overall Structure and Data Flow for two neighboring nodes supporting the SRP protocol	5
Figure 2-2:	The INTERFACE process	9
Figure 4-1:	Annotated AGENT and SURROGATE processes at node *n*	27
Figure 4-2:	Annotated INTERFACE process at node n	27

Abstract

In this paper, we view computer networks as distributed systems that provide their users with a set of services. A given target network configuration is conceived as a network of communicating virtual machines and its behavior is modelled through a system of communicating sequential processes. We thus develop a proof methodology which enables us to express and verify partial and total correctness assertions about the system in a simple and natural way. Global invariants are used to establish invariant properties of the whole system and temporal logic to express eventuality properties.

1. Introduction

We view a computer network as a distributed system which provides its users with a set of services. Every user can access any node in the network, and submit a request for getting a particular service (e.g. data retrieval, file transfer, etc.). The network will eventually satisfy this request and will also give a response to the user (this response may be either data retrieved or a message indicating the completion of the user's request). In what follows we develop distributed algorithms, running on each node of the network, which hide from the user the distributed nature of a computer network by handling both local and remote requests in a uniform way. In section 2 we present an algorithm whereby a node can handle at most one request for a particular service at any given time; we also assume that the underlying communication medium linking the various nodes of the network is perfectly reliable. These assumptions are relaxed in [10]; there we assume that every node is able to handle a bounded number of requests for a particular service at any given time. We furthermore assume that the medium is unreliable permitting information to be lost, duplicated or reordered.

At each node, the algorithms are implemented as a system of Communicating Sequential Processes (CSP). CSP is a language of concurrent programming developed by Hoare in [6] which we will subsequently be using with some simple extensions. For an alternative implementation of the algorithms using ADA, the reader is referred to [11].

Abstractly, we wish to conceive of some given target network configuration, as a network of communicating Virtual Machines (VMs) (Fig 1-1). The individual nodes of a particular network correspond to fully independent (autonomous) processors, each of which is capable of executing a complete CSP program. We can thus model a broad number of computer networks such as:

1. Inter-Virtual-Machine Transmission (e.g. the "VMCF" capability provided by IBM's VM/370 system) where seperate VMs are actually multiprogrammed onto the same physical processor (and so may communicate by accessing common memory supplied by the operating system).

2. Input/Output-Bus Transmission (e.g. between a central processor and some intelligent peripheral), where a number of distinct (but quite specialized) processors communicate by direct, very high-speed

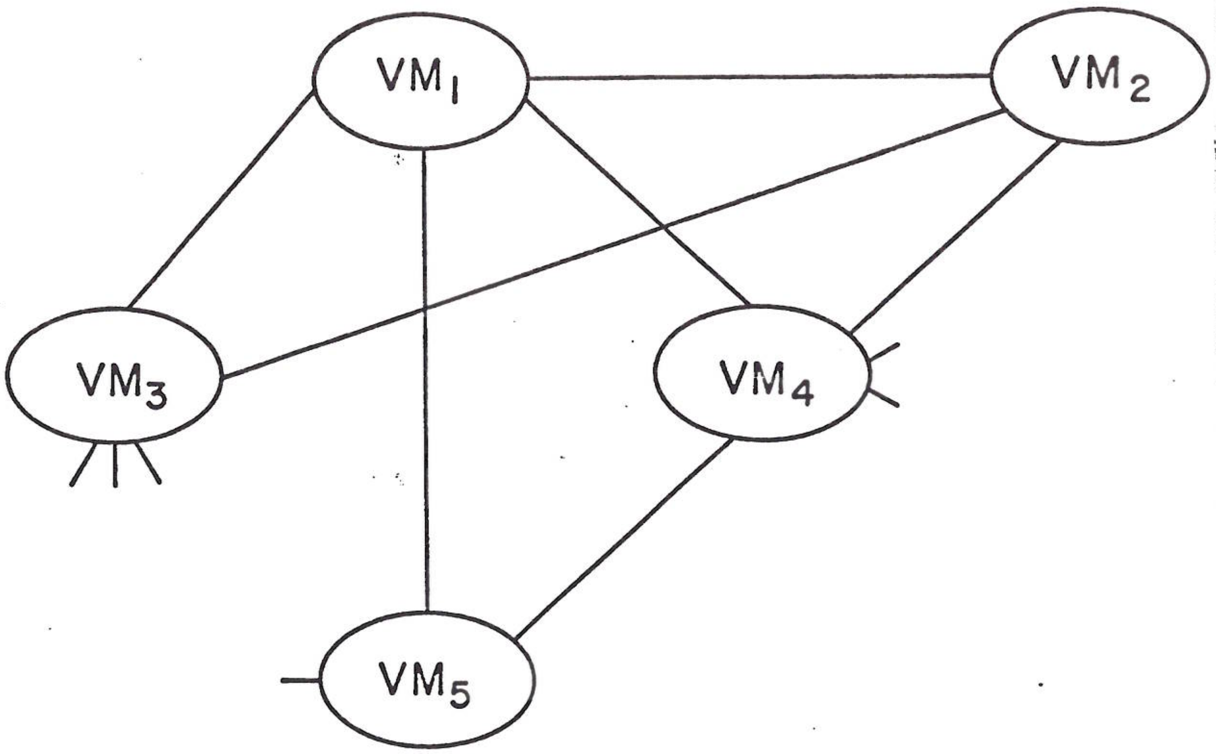

Figure 1-1: A network of communicating Virtual Machines

connections.

3. Local-area Network Transmission (e.g. the Xerox Ethernet or the Cambridge Digital Communication Ring, see [12]), where several autonomous (and often logically specialized) processors are connected together over relatively short distances by dedicated cables.

4. Long-Haul Network Transmission (e.g. the ARPANET and various public or private packet-switching networks) where many large-scale general-purpose processors are interconnected by means of phone lines, microwave relays nor even satellite communication.

The paper is organized as follows: in section 2 we describe a simple distributed algorithm for sequential request processing, assuming a perfectly reliable underlying communication medium. In the subsequent section 3 we present an axiom system for expressing the partial correctness properties of the sequential request processing protocol and its temporal logic extensions for handling the verification of total correctness properties. In the final section of this chapter we prove that responses always correspond to their associated requests, and that all requests are processed exactly once by the system.

2. Sequential Request Processing Protocol

The strategy employed in this section is based on the property that no more than one remote request of each service is in progress from the same node at any given time, so as to avoid saturation of the communication medium or overloading of the corresponding server node. As such, this property is necessarily established on the driver side of the protocol defined below.

2.1. The overall system

The overall system is represented as an array of CSP processes, each one standing for one node of the network:

```
    NETWORK :: [ NODE [ n : 1 . . N ] ]
```

The interconnection pattern of the nodes define the sets Γ(n), whose elements are the indices of the neighboring nodes to node *n*. Function **target_node,** residing in every node of the network is able to select a neighboring node, according to the service requests received.

The overall structure and associated data-flow for two neighboring nodes are depicted in Fig. 2-1. There are three processes running in parallel on every node:

```
    NODE[n] : : [ DRIVER | | INTERFACE | | SERVER ]
```

The DRIVER process receives the requests from the users and forwards them to the INTERFACE. The INTERFACE selects (using the target-node function) the appropriate node to service the request (which might very well be the local node) and transmits the request to the chosen node *k*. The INTERFACE process of node

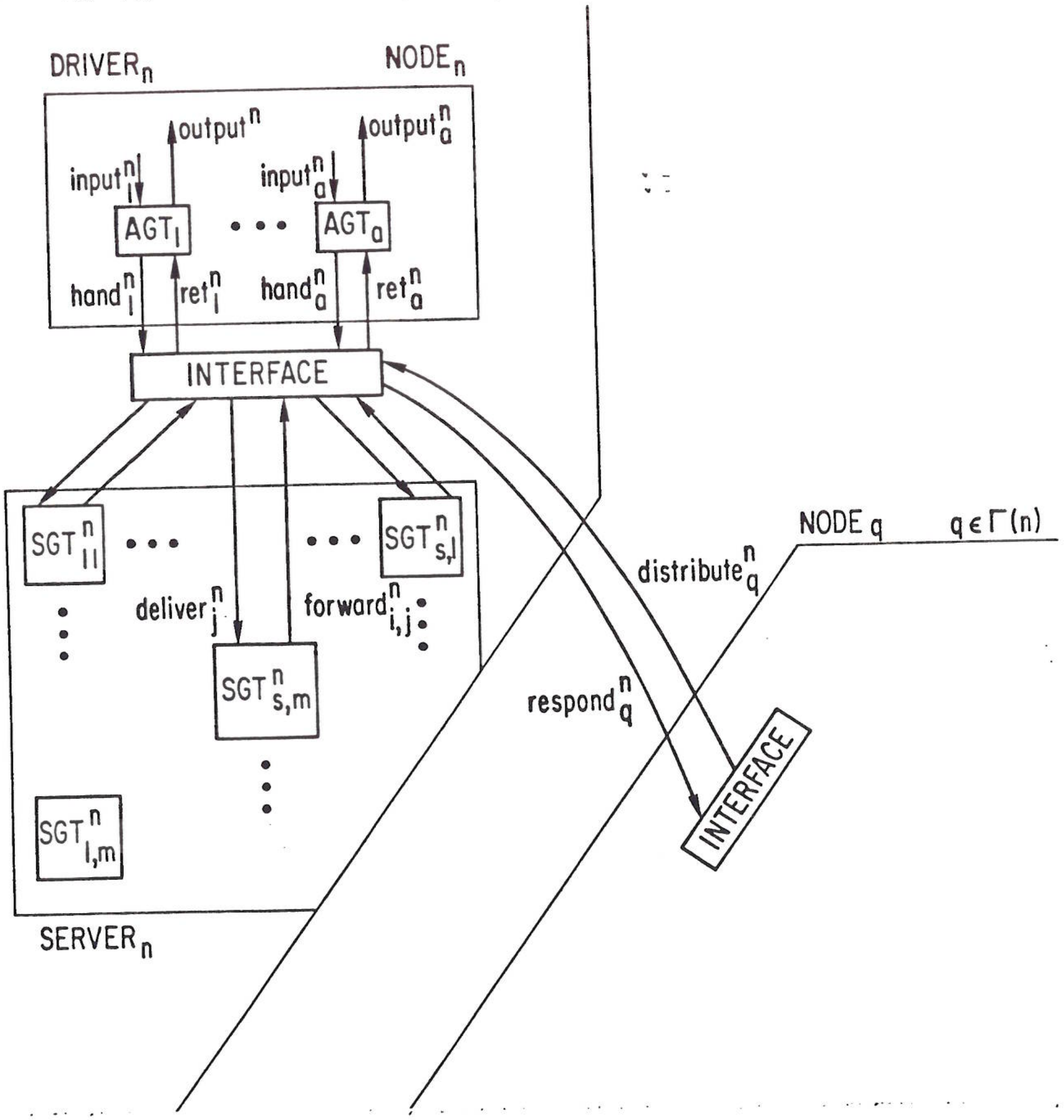

Figure 2-1: Overall Structure and Data Flow for two neighboring nodes supporting the SRP protocol

k delivers the request to the SERVER, which then forwards the response to the INTERFACE process of node *k*. Finally, the response is shipped back to node *n*, the INTERFACE process of which, returns the results to the DRIVER, that makes them available to the user. We next describe the DRIVER, INTERFACE and SERVER processes in detail.

2.2. the DRIVER

The DRIVER is defined as an array of processes, called AGENTS, each one responsible for one type of service, among those available to users at node n:

```
    DRIVER : : [ AGENT [ a : 1 . . A ] ]
```

AGENT[a] is the only process which is responsible for all requests of "type *a*" submitted at node *n*. Its body is shown below:

```
    AGENT[a] : :  *[ USER ? input(p) →
                                        INTERFACE ! hand(p);
                                        INTERFACE ? return(r);
                                        USER      ! output(r)
                   ]
```

In the above program, a user who can access node *n* and request service of "type *a*", is able to communicate with AGENT[a] and pass a parameter list *p* to an object of type *input*. These parameters are then passed to the INTERFACE. A typed message is again used (*hand (p,a)*) for reasons which will become apparent in the analysis section of the SRP protocol. The result parameter list *r* is then placed by the INTERFACE in an object of type *return* and is deliverd to the appropriate agent, which passes it back to the (unknown) user.

2.3. The Server

Similarly, the server is defined as an array of processes, called SURROGATES, whose sole responsibility is to (locally) process a request for a given type of service. However, this time the array is two-dimensional to account for the fact that for each type of service, D (= $|\Gamma(n)|$) surrogates are required, the number of neighbors to node n:

```
SERVER :: [ SURROGATE [ s : 1 . . S ; m : 1 . . D ] ]
```

The number of services S, supplied by the server of node n, does not, of course, have to be equal to A, the number of agents. For simplicity however, we assume that, on every node, $S = A$ and, moreover, that there is an one-to-one natural mapping between agent indices and types of services at the server's side. We next give the code for the SURROGATE process:

```
SURROGATE[s,m] : :
          *[INTERFACE? deliver (p, source_node) →
                       process(p,r);
                       INTERFACE!forward(r, source_node)
          ]
```

In a cyclic fashion, a surrogate simply receives a new request from the INTERFACE, processes it and forwards the results list r back to the INTERFACE. It also forwards the identity of the node, which this request originated from. The INTERFACE will thus be able to deliver the response to the right agent.

2.4. The INTERFACE

The communication medium is assumed to be perfectly reliable and messages always reach their destination without being lost, duplicated, reordered or corrupted. The body of the INTERFACE process of node *n* is given in figure 2-2.

Basically, INTERFACE has to nondeterministically select among the following six alternatives:

1. receive a new request from one of the agents. Using function *target_node*, INTERFACE determines the destination node of the request and by setting the appropriate entry of the flag array *ERC* (standing for *E*nable *R*equest *C*hannel) it enables communication with the target node. Furthermore, it disables communication with all agents, by turning off flag *EA* (standing for *E*nable *A*gent) until the parameter list p_1 is transmitted to its destination. Finally, it temporarily stores the identity of the agent, which originated the last request, in the variable *aid*.

2. if the appropriate flag permits, transmit to some neighboring node *k* a request, together with the name of the agent (i.e. the type of service) responsible for it. Once the communication is accomplished, disable transmission of requests to node *k* (reset *ERC[k]*) and enable communication with the agents again (set flag *EA*).

3. receive a request from a neighboring node and deliver it to the

```
INTERFACE ::
*[ [] (a : 1..A) EA; AGENT[a] ? hand(p1) →
                                        q := target_node(p1);
                                        aid := a;
                                        ERC[q] := true;
                                        EA := false

   [] (k : 1..D) ERC[k]; NODE[k] ! distribute(p1, aid) →
                                        ERC[k] := false;
                                        EA := true

   [] (k : 1..D) NODE[k] ? distribute(p2, agt) →
        [ [] (m : 1..D) SURROGATE[agt, m] ! deliver(p2, k) → skip ]

   [] (s : 1..S ; m : 1..D) ES ;
        SURROGATE? forward(r1, source_node) →
                                        EAC[source_node] := true;
                                        sid := s;
                                        ES := false

   [] (k : 1..D) EAC[k]; NODE[k] ! respond (r1, sid) →
                                        EAC[k] := false;
                                        ES := true

   [] (k : 1..D) NODE[k] ? respond(r2,sgt) →
                                        AGENT[sgt] ! return(r2)

   ]
```

Figure 2-2: The INTERFACE process

appropriate surrogate.

4. if enabled, receive a response from a surrogate and prepare for its transmission to its source node. Flags *ES* (*E*nable *S*urrogate) and *EAC* (*E*nable *A*nswer *C*hannel) function exactly as flags *EA* and *ERC* above.

5. if enabled, transmit a response to its source node; then prepare for receiving another response from some surrogate.

6. finally, receive a response from a neighboring node and deliver it to the appropriate agent. Note that according to the aforementioned convention, the first surrogate index (here called *sgt*) is also the agent index.

3. Description of the Formal Tools

In general, we want to prove *safety* and *liveness* properties about network protocols. As the folklore goes, safety properties are properties which assert that something bad never happens, while liveness properties are properties that assert that something good will eventually happen. In the context of network protocols there are two classes of properties that we want to verify:

1. that the responses that the users receive, match their requests and that every request is processed *at most once* by the network (safety).

2. that every message (be it a request or a response) is eventually delivered

through the network to the appropriate node (or user) (liveness).

To reason about safety and liveness properties of programs written in CSP we shall use a mixture of informal deliberation - whenever we think that a formalization of the presented arguments is more or less straightforward - together with proof theoretic deductions. Several partial correctness proof systems for CSP have already been proposed in the literature ([1], [LG], [9], [3]). We will be using the proof system proposed by Apt, Francez and De Roever in [1] with the addition of history variables to record the history of communication exchanges between every potentially (syntactically) matching pair of processes. History variables are also used in [9] and [7]. Our formal tool for proving liveness properties will be temporal logic.

3.1. History variables

Our *assertion language* is a first order language (see, for example, [Shoenfield]) in which certain variables are designated as *history variables.* Let us consider a pair of i/o commands of the form: $P[e_1]\ !\ T(t_1, \ldots, t_n)$ and $P[e_2]\ ?\ T(x_1, \ldots, x_n)$. Assume that the output command appears in the body of the process $P[i]$ and that the input command appears in the body of the process $P[j]$. Here, e_1 and e_2 are integer expressions, $x_1, \ldots, x_n$ are variables local to process $P[j]$, $t_1, \ldots, t_n$ are terms composed by variables local to process $P[i]$ and T is the type of the message to be exchanged. We say that these two i/o commands *match potentially* if j is in the range of values obtained by e_1 and i is in the range of values obtained by e_2.

The reader is reminded that this is not standard CSP, in [6] e_1 and e_2 are only allowed to be integer constants. However, we use this extension in our network protocols and we feel that it is quite useful. To each syntactically matching communication pair we assign a history variable, which, by convention, has the same name as the type of the message exchanged. At any given (global) state, each history variable contains the sequence of messages exchanged through the communication pair (channel) that it is attached to.

Following Hailpern [5], we now introduce notation for describing histories. Let α and β be arbitrary history variables. The length of α is denoted by $|\alpha|$. If the first element of α is α_1, the second is α_2, . . . , and the last one is α_n, then we can write:

$$\alpha = <\alpha_1 \alpha_2 \ldots \alpha_n>$$

We denote the i-th element of history α by α_i, and concatenation of sequences by juxtaposition:

$$\alpha = <\alpha_1 \ldots \alpha_n><\beta_1 \ldots \beta_m> = <\alpha_1 \ldots \alpha_n \beta_1 \ldots \beta_m>$$

We write $\alpha \leq \beta$ if α is an *initial subsequence* of β. This means that $|\alpha| \leq |\beta|$ and the two sequences are identical in their first $|\alpha|$ elements. Let α^- be derived by α by omitting the last element of α. For the following CSP program,

$*[\ S_1;\ P_1\ ?(!)\ \alpha(x_1,\ \ldots\ ,\ x_n);\ S_2;\ P_2\ ?(!)\ \beta(x_1,\ \ldots\ ,\ x_m);\ S_3\ ]$

we may clearly deduce the inequalities given below:

$$0 \leq |\alpha| - |\beta| \leq 1 \qquad (1)$$

Furthermore, in the following specialization

$*[\ S_1;\ P_1\ ?\ \alpha(x_1,\ \ldots\ ,\ x_n);(\rightarrow)\ P_2\ !\ \beta(x_1,\ \ldots\ ,\ x_n);\ S_3\ ]$

one can make the stronger statement shown below:

$$\alpha^- \leq \beta \leq \alpha \tag{2}$$

We shall be dealing with histories of structured CSP objects, for which selection operators are defined in the obvious way: if $n_1, \ldots, n_k$ are the component names of the object $A = T(v_1, \ldots, v_k)$, then $A.n_1 = v_1, \ldots, A.n_k = v_k$. If history α is composed of objects of type T as above, then we extend the selection operator to histories in the following way: if $\alpha = <A_1 \ldots A_1>$ then $\alpha.n_i = <A_1.n_i \ldots A_m.n_i>$. We shall also want to form subhistories of α by selecting some of its objects. We will usually be given a function f and the selection criterion will be that the value of this function applied to some particular component n_j of all the elements should be equal to a predetermined constant v:

```
$sel(v, \alpha, n_j, f) \triangleq$
**if** $\alpha = <>$ **then** $<>$
**else if** $v = f(first(\alpha.n_j))$ **then**
$(first(\alpha).n_1, \ldots first(\alpha).n_{j-1}, first(\alpha).n_{j+1}, \ldots first(\alpha).n_k)sel(v, rest(\alpha), n_j, f)$
**else** $sel(v, rest(\alpha), n_j, f)$
```

where <> is the empty history and $rest(\alpha)$ and $first(\alpha)$ have the obvious meanings. We will frequently omit the third or the fourth argument of **sel** or even both, whenever their values are obvious from the context. Assume that the objects of history α belong to a totally ordered set (e.g. assume that each request, submitted to a particular agent, is uniquely identified by a code number; α may be the history of codes associated to these requests). Then α is *nondecreasing* if $\alpha_i \leq \alpha_j$, whenever $i<j$. We denote this property by $Nd(\alpha)$. Similarly α is *monotonically increasing* if $\alpha_i < \alpha_j$, whenever $i<j$. Again we write $Mi(\alpha)$. Let $S(\alpha)$ denote the set of elements of α. Then, for monotonic histories α and β the following lemma holds:

Lemma 1: *If $Mi(\alpha)$ and $Mi(\beta)$ then from $S(\alpha^-) \subseteq S(\beta) \subseteq S(\alpha)$ we can deduce $\beta \leq \alpha$.*

Proof: For the sake of contradiction, assume that the lemma holds for the first *i-1* elements of both histories. Now suppose that $\beta_i \neq \alpha_i$. By hypothesis, $\beta_i \in S(\alpha)$. Hence, let $\beta_i = \alpha_j$. We distinguish the following cases:

1. $j<i$. By our assumption, there is $\beta_k = \alpha_j = \beta_i$ in violation of the monotonicity of β.

2. $j>i$. By the monotonicity of α, $\alpha_j > \alpha_i$. Therefore, α_i cannot be the last element of α and by hypothesis, $\alpha_i \in \beta$. Let $\beta_k = \alpha_i < \alpha\text{-}(j = \beta_i)$. By the monotonicity of β, $k<i$. By our assumption, however, $\beta_k = \alpha_k = \alpha_i$ in violation of the monotonicity of α.

Q.E.D.

For nondecreasing histories α and β, we define the **merge** operation recursively as follows:

```
$Merge(\alpha, \beta) \triangleq$
  if $\alpha = <>$ then $\beta$
  else if $\beta = <>$ then $\alpha$
  else if $first(\alpha) < first(\beta)$
      then $first(\alpha) first(\beta)\ Merge(\ rest(\alpha), rest(\beta))$
      else $first(\beta) first(\alpha)\ Merge(\ rest(\alpha), rest(\beta))$
```

Observe that if $\alpha_i = \beta_j$, for some i and j, both of them are included in Merge(α, β).

3.2. A Proof System for Safety Properties

To reason about safety roperties of CSP programs, one has first to provide proofs for component processes and then to deduce properties of a parallel program by analyzing the proofs for components. Apt, Francez and De Roever in [1] provide a method, called *cooperation test*, to tie separate proofs together into a meaningful whole. In this section, we only present those axioms and proof rules which are explicitly used in the analysis of our algorithms. The reader is referred to [1] for the complete axiom system. In what follows, by α we denote an i/o command of the form: $p[e]\ ?\ T(x_1, \ldots, x_n)$ or $p[e]\ !\ T(t_1, \ldots, t_n)$.

We first define the concept of **bracketing**: (adopted from [2])

Definition 1: *A process is bracketed if the brackets "<" and ">" are interspersed in its text so that for each program section <S> (to be called* **a bracketed section***), S is of one of the following forms:*

1. $S_1 \, ; \, a \, ; \, S_2$ *or*

2. $a \rightarrow S_1$,

and S_1 *and* S_2 *do not contain any i/o statements.*

For the cooperation tests a *global invariant I* (possibly referring to variables of *all* processes and auxiliary variables) is introduced. Every process is annotated with *brackets* in such a way that a *bracketed section* contains at most one communication command. The invariant is only required to hold outside bracketed sections. We now repeat the definition of the *potential matching* for the convenience of the reader:

Definition 2:

1. *Two communication commands of the form* $p[e] \,?\, T(x_1, \ldots x_n)$ *and* $p[e'] \,!\, T(t_1, \ldots, t_n)$ *are* **potentially matching** *if they appear in a q-process and p-process respectively. (Hence* $p[e] \,?\, T(x_1, \ldots, x_n)$ *and* $p[e'] \,!\, T(t_1, \ldots, t_n)$ *are always potentially matching).*

2. *Two bracketed sections* $\{pre_1\}\ S_1\ \{post_1\}$, $\{pre_2\}\ S_2\ \{post_2\}$ *are* **potentially matching** *if the corresponding communication commands are.*

In the above definition, by *p*-process (or *q*-process) we mean a process that is a member of an array of process p (or q). The notion of *potential matching* replaces that of syntactic matching in [1]. A potential match will be an *actual* (semantic) *match* provided the following two conditions hold:

1. The sections S_1 and S_2 are taken from $q[i]$ and $p[j]$ respectively, and the current values of e and e' are j and i respectively, and

2. There is a global state in a computation sequence when the controls of $q[i]$ and $p[j]$ are about to execute S_1, S_2 respectively.

The cooperation test has to establish the postconditions of the potentially matching bracketed sections and the invariant, given that the precondition and the invariant held initially. The invariant is also used to rule out all potentially matching pairs of bracketed sections, which *do not* actually (semantically) match. The cooperation test is:

$$\{ pre_1 \wedge pre_2 \wedge I \wedge e = j \wedge e' = i \}\ S_1 \| S_2\ \{ post_1 \wedge post_2 \wedge I \}$$

whenever S_1 and S_2 are potentially matching bracketed sections taken from $q[i]$ and $p[j]$ respectively.

In case S_1 and S_2 do not semantically match, the conjunction of the precondition and the invariant will be inconsistent. To establish the cooperation test we use the following additional axiom:

communication

$$\{\, T(i, j) = \alpha \}\ p[i]\ ?\ T(x_1, \ldots, x_n) \parallel p[j]\ !\ T(t_1, \ldots, t_n)$$
$$\{ \bigwedge_{l=1}^{n} x_l = t_l \wedge T(i, j) = \alpha <(t_1, \ldots, t_n)>\}$$

provided $p[i]\ ?\ T(x_1, \ldots, x_n)$ and $p[j]!\ T(t_1, \ldots, t_n)$ are taken from $p[j]$ and $p[i]$ respectively. Here $T(i, j)$ is the history of the unidirectional channel permitting flow of information from process i to process j.

We can now give the proof rule for parallel composition:

parallel composition

proofs of $\{p_i\}\ P[i]\ \{q_i\}\ (i = 1, \ldots, m)$ *cooperate*

$\{p_1 \wedge \ldots \wedge p_m \wedge I\}\ P[1] \parallel \ldots \parallel P[m]\ \{q_1 \wedge \ldots \wedge q_m \wedge I\}$

provided no variable free in I is subject to change outside a bracketed section.

We have now concluded the presentation of our proof system for safety properties.

3.3. Using Temporal Logic to prove liveness properties

Temporal logic is an instance of modal logic [HC], which considers a *universe* that consists of many similar *states* (or *worlds*) and a basic *accessibility* relation between the states, $R(s, s')$, which specifies the possibility of getting from one state s to another state s'. As Manna and Pnueli point out in [8], "the main notational idea is to avoid any explicit mention of either the state parameter or the accessibility relation. Instead we introduce two special operators that describe properties of states which are accessible from a given state in a universe."

The two *modal operators* introduced are $\Box$ (called the *necessity operator*) and $\Diamond$ (called the *possibility operator*. Let $|W|_s$ denote the *truth value* of formula w in state s. Then:

$$|\Box W|_s = \forall s' \, [\, R(s, s') \supset |W|_{s'} \,] \tag{3}$$

$$|\Diamond W|_s = \exists s' \, [\, R(s, s') \wedge |W|_{s'} \,] \tag{4}$$

A *modal formula* is a formula constructed from proposition symbols, predicate symbols, function symbols, individual constants and individual variables, the classic logic operators and quantifiers, and the modal operators. The *truth value of a modal formula* at a state in a universe is found by repeated use of 3 and 4 above for the modal operators and evaluation of any subformula on the state itself. A formula w which is true in all states of every universe is called *valid*, i.e. w is valid if for every universe U and every state $s \in U$, $|W|_s$ is true. Henceforth, we will assume that the accessibility relation R is *reflexive* and *transitive*, i.e.

$$\forall s \; R(s, s)$$

$$\forall s_1, s_2, s_3 \, [\, R(s_1, s_2) \wedge R(s_2, s_3) \supset R(s_1, s_3) \,]$$

In [10] we present a complete axiom system for propositional modal logic and its extension for formulas involving quantifiers.

Our language uses a set of basic symbols consisting of individual variables and constants, and proposition, function and predicate symbols. The set is partitioned into two subsets: global and local symbols. The *global symbols* have a uniform interpretation over the complete universe and do not change their value or meaning from one state to another. The *local symbols*, on the other hand, may assume different meanings and values in different states of the universe. A *universe* consists of a nonempty domain of interpretation D, an assignment over D to all global symbols a set of *states (worlds)* S where each state $s \in S$ specifies an assignment over D to all *local* symbols, and a specification of the accessibility relation R. A *model* (U, s_0) is a universe U with one of the states of U, $s_0 \in S$, designated as the initial or reference state. Manna and Pnueli characterize temporal logic as follows:

> The framework of temporal logic is a modal framework in which we impose further restrictions on the models of interpretation ([PRI], [RU]). The interpretation given by temporal logic to the basic accessibility relation is that of the passage of time. A world s' is accessible from a world s if through development in time, s can change to s'. We concentrate on histories of development which are linear and discrete. Thus, the models of temporal logic consist of ω-sequences, i.e. infinite sequences of the form $\sigma = s_0, s_1, \ldots$ In such a sequence, s_j is accessible from s_i *iff* $i \leq j$. Due to the discreteness of the sequences we can refer not only to states that lie in the future of a given state, but also to the (unique) immediate future state or *next state.* This leads to the introduction of an additional operator, the *next instant* operator denoted by $\circ$ ([8]).

Consider, now, a program

$P :: [P_1 \| \ldots \| P_n]$,

where each P_i denotes a single process containing no nested processes within its body. Assume that each statement in every P_i has a unique label identifying that statement. Let L^i denote the set of labels of P_i, and let $\vec{y}^{\,i}$ be the vector of variables local to process P_i. An *execution state* of P has the form $s = < l^1_{i_1}, \ldots, l^n_{i_n}, \vec{n} >$ where $l^j_{i_j} \in L^j$ is the label of the next statement to be executed by process P_j and $\vec{n}$ gives the current values of the local variables $\vec{y} = < \vec{y}^{\,1}, \ldots, \vec{y}^{\,n} >$. For $l \in L^i$, the predicate *at* l holds (at a given state) iff control of P_i resides at l (i.e. the statement labelled by l is the next candidate for execution in P_i). Similarly, the predicate *after* l holds at a given state, iff control of P_i resides at the state *immediately* after execution of the statement labelled l.

Every partial correctness assertion of the form $\{p\}\ S\ \{q\}$ can now be converted to an equivalent temporal logic formula as follows:

$$(at\ S \wedge p) \supset \Box\ (after\ S \supset q)$$

3.3.1. The strong fairness axiom

The issue of fairness is briefly discussed in [7], the paper defining CSP. There it is argued that it is the programmer's responsibility to prove that his (or her) program behaves correctly, without making any assumptions whatsoever, about the fairness introduced by the implementation. We felt, however, that ensuring a fair execution of the SRP protocol through explicit programming in CSP, would dramatically increase their already high complexity and would distract us from the reliability issues that we want to study in this thesis. On the other hand, it would be impossible to establish the desired liveness properties of the protocols without making any assumptions about fairness.

Francez and De Roever ([4]) distinguish two kinds of fairness: *weak* fairness - associated with eventual advance of each process - and *strong* fairness - associated with eventual occurance of each pending communication. We adopt the *strong* fairness assumption, which we next formalize, as in [4]. In the sequel, for simplicity of notation, we consider only one array of processes which we call *p*.

Let $L \in \mathbf{L}^{i}$ be a label of an alternative, repetitive or a communication command in process P_i. Let m be the number of guards g_a, $a = 1, \ldots, m$ in the alternative or the repetitive command. By convention, we assume that this number is always 1 in the case of a communication command. For example, L could label the following alternative command:

```
L : [ [] b1; C1 → L1 : S1
        .
        .
        .
      [] bm; Cm → Lm : Sm
    ]
```

Let $B(g_a)$ denote the *boolean* part of g_a (**true**, if not explicitly mentioned or if L is labelling a communication command). Also, let

$$C(g_a) = \langle p,q \rangle,\ 1 \leq p,\ q \leq n \text{ and } p = i \text{ or } q = i,$$

be the *communication* part of g_a ; here p denotes the index of the source process and q the index of the destination process. Because we allow computed targets, function $C(.)$ depends on the current state of the system; therefore it is rather a semantic function than a syntactic one, as it is the case in [4]. By convention we take $p = q = i$ for those branches having no communication part. Next, we define the set of *communication expectations* for L:

$$E^i_{L,s} = \{ C(g_a) / B(g_a) = \textbf{true} \ \textit{in state } s,\ a = 1, \ldots, m\}$$

Clearly, if L labels a communication command, $E^i_L = \{ C(g_1) \}$ where g_1 is the command. Henceforth, we will omit the state subscript of the communication expectation sets, but we will keep in mind that these sets are always state dependent.

In [4], the strong fairness axiom is expressed as follows:

$$\forall i\ (1 \le i \le n).\ \forall L\ (L \in \mathbf{L}^i).\ \forall a\ (1 \le a \le m(L)).$$
$$\forall j\ (1 \le j \le n).\ \forall L'.\ (L' \in \mathbf{L}^j).$$
$$\{ \Box \Diamond [\ at\ L \wedge at\ L' \wedge B(g_a) \wedge C(g_a) \in E^j_{L'}] \supset \Diamond\ at\ L_a \}$$

This axiom, expressing eventual occurrence of every matched communication, can be understood as follows:

Let P_i denote a given process, L a label of a given guarded command in P_i, and g_a denote a given guard. Let P_j denote a given process, and L' denote a given guarded command in P_j. Then if it is infinitely often the case that the communication part of g_a is matched with (an element of) $E^j_{L'}$, then eventually this specific communication will occur. Note again the special case of $i = j$ which induces fair choice among local alternatives.

3.3.2. Proving liveness properties

Again deviating from the original semantics of CSP, as presented in [7], we will consider liveness properties of non-terminating processes only. Henceforth, we assume that processes are composed by an optional initial terminating part, and a non-terminating repetitive command. Processes are allowed to be defined as arrays of other processes, but only at top level.

We next introduce the predicate *matching(L, a)* which becomes true iff there is at least one communication command in some process P_j matching the communication command in the guard g_a of the statement labelled L in process P_i

matching (L, a) $\triangleq$

$$\exists\, j\,(1 \leq j \leq n).\ \exists L'\,(L' \in \mathrm{L}^j)\,[\ at\ L' \wedge B(g_a) \wedge C(g_a) \in E^j_{L'}\,]$$

For the terminating constructs to be used, we now recursively define the predicate *terminate(L)*. The general scheme to be used for proving termination properties will be:

$$at\ L \wedge terminate\ (L) \supset \Diamond\ after\ L$$

The predicate *terminate(L)* is defined as follows:

skip

$$terminate\ (L) \equiv \Diamond\ after\ L$$

assignment

$$terminate\ (L) \equiv \diamond\ after\ L$$

composition

Let $L{:}\ S_1\,;\,S_2\,;\,.\,.\,.\,;\,S_n$. Then:

$$terminate(L) \equiv terminate(S_1) \wedge \bigwedge_{i=2}^{n} (at\ S_i \supset terminate(S_i))$$

alternative command

$$terminate(L) \equiv$$

$$\exists\ a\ (1 \leq a \leq m(L)).\ matching(L,\ a) \wedge \bigwedge_{a=1}^{m} (\ matching(L,\ a) \supset terminate(S_a\,))$$

terminating repetitive command

$$terminate\ (L) \equiv \exists\ a\ (1 \leq a \leq m(L)).\ matching(L,\ a) \supset$$
$$\bigwedge_{a=1}^{m} (matching(L,\ a) \supset terminate(S_a\,)) \wedge \diamond\ (\bigwedge_{a=1}^{m} \neg\ B(g_a\,))$$

The predicate *terminate*, as defined above, expresses the necessary and sufficient conditions for termination of the aforementioned constructs, in *all* possible execution sequences.

We next give a condition asserting that a non-terminating repetitive command labelled *L*, will never be blocked within its body, and that, moreover, control will be at *L* infinitely often:

$$\square \bigvee_{a=1}^{m} B(g_a\,) \wedge \square\ (\ \bigwedge_{a=1}^{m} (\ matching(L,\ a) \supset terminate(S_a\,))$$
$$\supset \square \diamond\ at\ L \wedge \square\ (\neg\ after\ L)$$

Observe that the condition above is also established in the trivial case, when the i/o commands of the guards, though infinitely often enabled, are never actually executed (because there are no matching i/o commands in other processes). In that case $\Diamond \Box$ *at* L holds.

This completes our description of the formal tools.

4. Analysis of the SRP protocol

To uniquely identify user's requests we assign to them a code *c*, so that no two requests have the same code. Moreover, the codes are monotonically increasing with time. The channel histories are named after the type of messages exchanged through that channel. Indices are also used whenever deemed necessary, to uniquely identify histories. Figures 4-1 and 4-2 show the annotated versions of the SRP processes.

In order to establish the safety properties of the SRP protocol we use the proof methodology outlined in section 3. Thus we form a global invariant *I*, which we should prove holding at all times. *I* is the conjunction of several clauses, that we list below:

$$Mi(input_a^n.C) \tag{5}$$

We first assert that the codes asssociated to incoming requests to be handled by agent *a* are monotonically increasing.

$$hand_a^n \leq input_a^n \tag{6}$$

The above inequality states that at node *n* and agent *a* the *hand*-history is an initial subsequence of the *input*-history. Similarly:

```
AGENT[a] : : {$input^n_a$ = <> ∧ $hand^n_a$ = <>
               ∧ $return^n_a$ = <> ∧ $output^n_a$ = <>}
  $AL^n_a$      : *[ USER? input(p,c) →
                     $AL1^n_a$ :    INTERFACE ! hand (p,c);
                     $AL2^n_a$ :    INTERFACE ? return (r,c);
                     $AL3^n_a$ :    USER ! output (r,c)
               ]


SURROGATE [s,m] : : { $deliver^n_{s,m}$ = <> ∧ $forward^n_{s,m}$ = <> }
     $SL^n_{s,m}$ : *[ INTERFACE ? deliver(p, source_node,c) →
               $SL1^n_{s,m}$ : process(p,r);
               $SL2^n_{s,m}$ : INTERFACE ! forward (r, source_node, c)
              ]
```

Figure 4-1: Annotated AGENT and SURROGATE processes at node *n*

```
INTERFACE : : { EA = true ∧ ES = true ∧ $\bigwedge_{i=1}^{n}$ ERC[i] = false
                ∧ $\bigwedge_{i=1}^{n}$ EAC[i] = false
                ∧ $\bigwedge_{q=1}^{|\Gamma(n)|}$ ( distribute$_q^n$ = <> ∧ respond$_n^q$ )

ILⁿ :  *[ [](a:1..A) EA; < AGENT[a] ? hand($p_1$,$c_1$) →
                                  q := target_node($p_1$,$c_1$);
                                  aid := a;
                                  ERC[q] := true; EA := false >

        [](k: 1..D) ERC[k]; < NODE[k] ! distribute($p_1$, aid, $c_1$) →
                                  ERC[k] := false ; EA := true >

        [](k: 1..D) NODE[k] ? distribute($p_2$, agt, $c_2$) →
          IL1ⁿ : [ [](m : 1..D)
              SURROGATE[agt, m] ! deliver($p_2$, k, $c_2$) → skip]

        [] (s : 1..S ; m : 1..D) ES;
              < SURROGATE[s, m] ? forward($r_1$, source_node, $c_3$) →
                    sid := s;
                    EAC[source_node] := true; ES := false >

        [] (k : 1..D) EAC[k]; < NODE[k] ! respond($r_1$, sid, $c_3$) →
                    EAC[k] := false; ES := true >

        [] (k : 1..D) NODE[k] ? respond($r_2$, sgt, $c_4$) →
                 IL2ⁿ : AGENT[sgt] ! return($r_2$, $c_4$)
      ]
```

Figure 4-2: Annotated INTERFACE process at node n

$$output_a^n \leq return_a^n \tag{7}$$

For the m-th SURROGATE process of service type *s* at node *dn* we have:

$$forward_{s,m}^{dn}.C \leq deliver_{s,m}^{dn}.C \tag{8}$$

In the following four clauses we describe the behaviour of the INTERFACE processes at two neighboring nodes *sn* and *dn.*

$$sel(a,\ distribute_{dn}^{sn}).C \leq sel(dn,\ hand_a^{sn},\ P,\ target\_node).C \tag{9}$$

Here, we select those requests listed in history *distribute* which come from agent *a* only, at source node *sn,* and we state that the thereby formed subhistory of their codes, is an initial subsequence of that same agent's *hand*-subhistory, containing codes of requests with destination node *dn.* Hereafter, for economy of notation, we will omit the third and fourth arguments of *sel,* when selecting from the *hand* history according to the destination node (or the *hand* history for that matter).

$$\underset{m \in |\Gamma(dn)|}{Merge}(sel(sn,\ deliver_{t,m}^{dn}).C) \leq sel(t,\ distribute_{dn}^{sn}).C \tag{10}$$

In 10 we consider the histories of all channels which deliver requests to all surrogates of type *t,* $t \in \{1, \ldots, S\}$ at node *dn.* We form their subhistories by selecting only those requests (of type *t*) coming from node *sn* and we merge them. We then state that the thus formed history is an initial subsequence of the *distribute*-subhistory containing requests of type *t.*

The following two clauses are essentially symmetric to 9 and 10.

$$sel(t,\ respond_{sn}^{dn}).C \leq \underset{m \in |\Gamma(dn)|}{Merge}(sel(sn,\ forward_{t,m}^{dn}).C) \tag{11}$$

$$sel(dn,\ return_t^{sn}.R\ ,target\_node).C \leq sel(t,\ respond_{sn}^{dn}\ ).C \qquad (12)$$

Again, for economy of notation, we shall omit the third and fourth arguments in *sel* above, when selecting from the *return* history according to destination node (or the *output* history for that matter).

The last two clauses of I express an invariant property of the flags *EA, ERC* and *ES, EAC* of INTERFACE

$$EA \otimes \bigotimes_{k=1}^{n} ERC[k] \qquad (13)$$

$$ES \otimes \bigotimes_{k=1}^{n} EAC[k] \qquad (14)$$

where "⊗" denotes exclusive OR.

In order to establish the truth of the invariant I we have to prove that I holds:

1. initially

2. outside the bracketed sections of every process.

Clearly, I holds initially because all histories are empty and EA and ES are **true** whereas all elements of ERC and EAC are **false**.

We now examine the truth value of each individual clause of I. When agent a

receives a request from the user the *input* history is increased by one more element by virtue of the communication axiom. So 6 now holds as a strict inequality. When the agent *a* communicates with the INTERFACE, the *hand* history is increased, and 6 still holds possibly as inequality. Since nowhere else in the system the *input* and *hand* histories are affected the truth of 6 is established. Observe that the processes pairs USER - AGENT[a] and AGENT[a] - INTERFACE, whose communications we used to establish 6, trivially pass their communication tests. We can similarly establish the truth of 7 and 8.

The truth of 13 can be established by inspection of the first two alternatives of the INTERFACE process. If we assume that 13 holds at the beginning of INTERFACE's infinite loop, then the first group of alternatives is enabled and the second disabled. Moreover, after execution of one of the alternatives in the first group, *EA* becomes false and *exactly one* of the *ERC*s becomes **true**, so 13 continues to hold. One of the alternatives of the second group is now enabled, namely the *q*-th branch; after its execution *EA* is reset to **true** and *ERC[q]* to **false** thus maintaining the truth of 13. The truth of 14 can be similarly established.

Let $\alpha = sel(a, distribute^{sn}_{dn}).C$ and $\beta = sel(dn, hand^{sn}_{a}).C$. If both α and β are empty then 9 holds trivially. If β is nonempty, consider β_i. The fact that β_i has been put into β implies that INTERFACE has communicated with the agent *a* and that subsequently the target node *dn* (value of *q*) is determined, communication with all agents has been disabled (*EA* set to **false**), and that exactly the *dn*-th branch of the second group of alternatives of INTERFACE has been enabled. Thus, if history α is altered, because of communication with node *dn*, then β_i will have to be appended to α. A simple induction on the length of α and β then shows that $\alpha_i = \beta_i$, thus establishing the truth of 9. The truth of 12 is similarly established.

The following lemma is used to show 10:

Lemma 1: *:*

1. The subhistory $\alpha^m = sel(sn, deliver^{dn}_{t,m}).C$ increases monotonically.

2. 10 is an invariant over all computation sequences.

Proof: To be given in the final draft of the paper.

We finally prove 11 through the following lemma:

Lemma 2: *11 is invariant over all computation sequences.*

Proof: To be given in the final draft of the paper.

Let us observe now a useful general property of histories, which follows as a direct generalization of the above lemma:

Lemma 3: *Consider the monotonic histories $\alpha, \beta^1, \dots, \beta^m$ and history γ. Assume that all $S(\beta^i)$ are pairwise disjoint, that every element of γ belongs in exactly one β^i and that $S(\alpha^-) \subseteq S(\gamma) \subseteq S(\alpha)$. Then $\gamma \leq \alpha$.*

We are now in a position to prove the following theorem:

Theorem 1:

1. Every request submitted to a node's driver reaches exactly one surrogate, at most once, at the server's side:

$$\underset{m \in \Gamma(dn)}{Merge} \left(sel(sn,\ deliver^{dn}_{t,m}).C \right) \leq sel(dn,\ input^{sn}_{t}).C$$

2. At every node, and for every agent responses correspond to requests:

$$output^{sn}_{t}.C \leq input^{sn}_{t}.C$$

Proof:

1. An immediate consequence of J and 10.

2. By applying lemma 3 for the histories $input^{sn}_{t}.C$, $sel(t,\ respond^{dn}_{sn}).C$ for all $dn \in \Gamma(sn)$, and history $return^{sn}_{t}.C$, we get:

$$return^{sn}_{t}.C \leq input^{sn}_{t}.C$$

which together with 7 yields the desired result.

Q.E.D.

The second part of the present analysis is devoted to liveness properties of the SRP protocol. Our goal is to prove that if an unbounded number of requests are submitted to every node of the network, then the output is an unbounded number of responses. We formalize the continuous availability of the USER processes by the following assertion:

$$\Box \ ((at AL^{n}_{a} \supset \Diamond \ at\ AL1^{n}_{a}) \wedge (at\ AL3^{n}_{a} \supset \Diamond \ after\ AL3^{n}_{a}))$$

We first prove that INTERFACE cannot be blocked at $IL1^{n}$ or $IL2^{n}$:

Lemma 4:

$$\diamond \; at\ IL1^n \supset \diamond \; after\ IL1^n \tag{15}$$

$$\diamond \; at\ IL2^n \supset \diamond \; after\ IL2^n \tag{16}$$

Proof: To be given in the final draft of the paper.

An immediate consequence of lemma 4 and the non-terminating repetitive command condition is the following lemma:

Lemma 5:

$$\Box \diamond \; at\ IL^n \tag{17}$$

Proof: Using the generalization rule we can transform clauses 13 and 14 of the invariant:

$$\Box \, (EA \otimes \bigotimes_{k=1}^{n} ERC[k]\,) \text{ and,}$$

$$\Box \, (ES \otimes \bigotimes_{k=1}^{n} EAC[k]\,)$$

i.e. at any time, one of the first and second group of alternatives and one of the fourth and fifth group of alternatives will be enabled. On the other hand, lemma 4, and the assignment and composition liveness axioms establish the termination of the statements following all the guards of the INTERFACE. We can thus apply the non-terminating repetitive command proof rule and conclude 17.

Q.E.D.

The next lemma shows that *EA (ES)* is enabled infinitely often.

Lemma 6:

$$\Box \Diamond EA \qquad (\Box \Diamond ES)$$

Proof: Assume the contrary. Then, from some state on, *EA (ES)* should be **false**: $\Diamond \Box (\neg EA)$ $(\Diamond \Box (\neg ES))$. Because of the 13 (14) -th clause of the invariant, exactly one *ERC[k]) (EAC[k])* should be **true** from some state on:

$$\Diamond \Box \exists k\, (1 \leq k \leq N).\, (ERC[k]) \quad (\exists k\, (1 \leq k \leq N)\, \Diamond \Box (ERC[k])) \quad (18)$$

According to lemma 5, at node *k* the third group of alternatives of INTERFACE is infinitely often enabled; hence, we can apply the strong fairness axiom and deduce that communication between the interfaces of nodes *n* and *k* will occur; by the assignment livenes axiom $\Diamond EA$ $(\Diamond ES)$ in contradiction with our assumption.

Q.E.D.

We are finally ready to prove the main liveness property of the SRP protocol.

Theorem 2: *At every node, if the users are continuously submitting requests and always willing to accept their responses, the input and output histories are unbounded:*

$$\forall sn\, (1 \leq sn \leq N).\, \forall t\, (1 \leq t \leq A). \qquad (19)$$

$$\Box\, ((at\ AL_t^{sn} \supset \Diamond\ at\ AL1_t^{sn}) \wedge (at\ AL3_t^{sn} \supset \Diamond\ after\ AL3_t^{sn})$$

$$\supset u(input_t^{sn}) \wedge u(output_t^{sn})$$

Proof: Consider agent *t*, who initially is at AL_t^{sn}. Because of 19, $\Diamond$ *at* $AL1_t^{sn}$. By lemma 6 and the fairness axiom, communication between the agent *t* and INTERFACE has to occur; thus $\Diamond at\ AL2_t^{sn}$. If *dn = target_node(p)* the *dn*-th branch is enabled from the second group of alternatives of INTERFACE, and communication with NODE[dn] will occur, again by virtue of the strong fairness axiom. Thus, at NODE *dn*, INTERFACE is at $IL1^{dn}$; by lemma 4 communication with some SURROGATE[t,m] will occur, and again $\diamond at\ IL^{dn}$, by lemma 5 SURROGATE[t,n] will go to $SL2^{dn}_{t,m}$, after processing the request and by lemma 6 communication with the INTERFACE will occur, therefore enabling the *sn*-th branch of the fifth group of alternatives in INTERFACE. Again by lemma 5 nodes *sn* and *dn* will communicate and the response will reach node *sn* with INTERFACE at that node being at $IL2^{sn}$. By lemma 4 communication with agent *t* will occur, therefore bringing agent *t* at $AL3^{sn}_t$; by 19,

$$after\ AL3^{sn}_t \supset \diamond\ at\ AL^{sn}_t$$

Thus, we proved that:

$$at\ AL^{sn}_t \supset \diamond\ at\ AL^{sn}_t$$

which means that:

$$\forall sn\ (1 \leq sn \leq N).\ \forall t\ (1 \leq t \leq A)\ \Box \diamond\ at\ AL^{sn}_t \qquad (20)$$

From 20, because of the non-terminating repetitive nature of AGENT, we can deduce 19.

Q.E.D.

5. Conclusion

In this paper we presented a distributed algorithm for request processing at remote sites of a network. We verified the preservation of the exactly once semantics applied to every request submitted to the network. We believe that the proof methodology used is of wider applicability to systems of processes modelling network protocols. One could envision a systematic way, whereby given a system of processes and their communication channels, one deduces properties of these channel histories. It should be interesting to see to what extend the verification of at least certain classes of process systems (e.g. those modelling network protocols) can be mapped to a calculus of these processes communication histories.

References

[1]

Apt K. R., Francez N., De Roever W. P.
A proof system for Communicating Sequential Processes.
ACM Transactions on Programming Languages and Systems 2(3):359-385, July, 1980.

[2]

Apt K. R.
Formal justification of a proof system for communicating sequential processes.
submitted for publication , 1981.

[3]

Chen C. Z., Hoare C. A. R.
Partial Correctness of Communicating Sequential Processes.
In *Proceedings of the 2nd symposium on distributed systems.* , Paris, April, 1981.

[4]

Francez N., De Roever W. P.
Fairness in communicating processes.
1980.
Extended Abstract.

[5]

Hailpern B.
Verifying Concurrent Processes using Temporal Logic.
PhD thesis, Stanford University, August, 1980.

[6]

Hoare C. A. R.
A calculus for total correctness of communicating processes.
1981.
unpublished manuscript.

[7]

Hoare C. A. R.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[8]

Manna Z., Pnueli A.
Verification of concurrent programs: the temporal framework.
Technical Report STAN-CS-81-836, Stanford University, June, 1981.

[9]

Misra J., Chandy K. M.
Proofs of networks of processes.
IEEE Transactions on software Engineering SE-7(4):417-426, July, 1981.

[10]

Nikolaou C. N.
Reliability Issues in Distributed Systems.
PhD thesis, Harvard University, 1982.

[11]

Schuman S. A., Clarke E. M., Nikolaou C. N.
Programming distributed applications in ADA: a first approach.
In *Proceedings of the 10-th International Conference on Parallel Processing.* ,
Bellaire, Michigan, August, 1981.

[12]

Spector A. Z.
Performing Remote Operations Efficiently on a local Computer Network.
Technical Report STAN-CS-80-831, Stanford University, January, 1981.
pre-publication draft.