A METHODOLOGY FOR VERIFYING REQUEST PROCESSING PROTOCOLS

Christos N. Nikolaou
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Edmund M. Clarke, Jr.
Department of Computer Science
Carnegie-Mellon University

Nissim Francez
Technion, Haifa, Israel

Stephen A. Schuman
Massachusetts Computer Associates, Inc.
Wakefield, Massachusetts

ABSTRACT

In this paper, we view computer networks as distributed systems that provide their users with a set of services, in a way which hides the distinction between those services which are local and those which are remote. We conceive of a given target network configuration as a network of communicating virtual machines and its behavior is modelled by a system of communicating sequential processes. Network protocols are described by a high level concurrent language (CSP) and a methodology is developed which permits the verification of 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 histories to record the sequence of communication exchanges between every matching pair of processes. Eventuality properties are expressed using linear temporal logic.

1. Introduction

We view a computer network as a distributed system which provides its users with a set of services. Every user is associated with a particular node in the network; however, a user can submit a request (e.g. data retrieval, file transfer, etc.) to any node capable of processing that request. In general, a user will not know the topology of the network or even whether the request is to be processed locally or remotely. The network will eventually satisfy the request and give a response to the user (either data retrieved or a message indicating the completion of the user's request). Thus, distributed systems able to handle remote procedure calls (see [13]) are naturally modelled by our request processing protocols and provided the motivation for the research reported in this paper. A system with the remote procedure call (RPC) capability can, in a way transparent to the user, execute the body of a called procedure at a remote site and have the results returned to the user's site. Implementation, performance and reliability problems

This research was supported by NSF Grant MCS-815553 while the first three authors were at Harvard University. At Massachusetts Computer Associates, Inc., this research was supported by the U.S. Army CORADCOM, through the Scientific Services Program under Delivery Order No. 1704 from Battele Columbus Laboratories.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

for this class of protocols have been addressed by others (see, e.g. [13]). We focus on the modelling and verification issues in this paper; an extension of our study to fault-tolerant protocols may be found in [14].

At each node, the algorithms are implemented as a system of Communicating Sequential Processes (CSP). CSP is a language for concurrent programming developed by Hoare [8] which we shall subsequently be using with the simple extension of allowing a dynamic determination of the target of communication ([4]). For an alternative implementation of the algorithms using ADA, the reader is referred to [16]. High-level concurrent languages, like CSP or ADA, are preferable to a transition graph or state-table specification of a finite-state-machine for describing complicated protocols, because all the powerful verification tools developed for high-level programs can then be used to prove the desired protocol properties.

The individual nodes of a particular network correspond to fully independent (autonomous) processors, each of which is capable of executing the program of a CSP process. We can thus model a broad number of computer networks such as:

- Inter-Virtual-Machine Transmission (e.g. the "VMCF" capability provided by IBM's VM/370 system) where separate VMs are actually multiprogrammed on the same physical processor (and so may communicate by accessing common memory supplied by the operating system).
- 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, high-speed connections.
- Local-area Network Transmission (e.g. the Xerox Ethernet or the Cambridge Digital Communication Ring, see [18]), where several autonomous (and often logically specialized) processors are connected together over relatively short distances by dedicated cables.
- 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 or even satellite communication.

Note that for message passing systems, an intermediate CSP process may be necessary to model the function of the message buffers.

In this paper, we show how network protocol properties can be described using histories to record the sequence of communications between processes and linear temporal logic to express liveness properties. The communication axiom that we use allows

histories to be updated implicitly and permits us to avoid introducing histories explicitly into the program text. In [6] a similar approach is used to deal with correctness properties of network protocols; the programming language, however, uses monitors and buffered message-passing for synchronization (unlike the rendez-vous mechanism of CSP). Hence, issues of total correctness and fairness are handled quite differently. In [17] the use of temporal logic is proposed, to specify properties of finite state machines; however, no attempt is made to translate these specifications into a procedural description in a high-level language.

The paper is organized as follows: in section 2 we describe a simple distributed algorithm for sequential request processing (SRP), assuming a perfectly reliable underlying communication medium. In section 3 we adopt an axiom system for expressing safety properties in CSP and enhance it by temporal logic extensions for handling the verification of liveness properties. In the final section we show how the formal tools presented in section 3 can be applied to verify the desirable properties of the SRP protocol, i.e. the responses always correspond to their associated requests, and that all requests are processed exactly once by the system (*exactly once semantics*). The SRP protocol is extended in [14] where we assume that every node is able to handle a bounded number of requests for a particular service at any given time. We further assume that the medium is unreliable permitting information to be lost, duplicated or reordered. It is again shown using the techniques introduced in this paper, that the extended protocol (called the Concurrent Request Processing Protocol) satisfies the same properties, i.e. correct correspondence of requests to responses and exactly once semantics.

2. The Sequential Request Processing (SRP) 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 concurrent CSP processes, each one standing for one node of the network:

$$NETWORK :: [NODE [n : 1 \dots N]]$$

We assume a static (fixed) interconnection pattern of the nodes, given by means of the nonempty sets $Nb(n)$ whose elements are the indices of the neighboring nodes to node n . The meaning of $j \in Nb(n)$ is that there is a bidirectional communication channel between node i and node j . A routing function **target_node**, residing in every node of the network is able to select a neighboring node, according to the service requests received. The communication medium is assumed to be perfectly reliable and messages always reach their destination without being lost, duplicated, reordered or corrupted.

The overall structure and associated data-flow for two neighboring nodes are depicted in Fig. 2-1. A node consists of three groups of processes: **AGENTS**, an **INTERFACE** and **SURROGATES**. The **AGENT** group of processes receives the requests from the users and forwards them to the **INTERFACE**. The **INTERFACE** selects (using the **target_node** function) the appropriate node k to service the request (which might very well be the local node) and transmits the request to the chosen node. The **INTERFACE** process of node k delivers the request to the **SURROGATE** group of processes, 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 appropriate **AGENT** which makes them available to the user.

Figure 2-1: Two neighboring nodes

For proof purposes, we assign to requests a code c , so that no two requests have the same code; codes are monotonically increasing with time. The bracketing of statements is required by the proof in section 4 and will be explained later.

2.2 The AGENTS

AGENT[a], $a = 1 \dots A$, is the only process which is responsible for all requests of "type a " submitted at node n . Its body is shown below:

```

AGENT[a]::
AL1_a^n : * [<USER?input(p,c)> →
              AL1_a^n : <INTERFACE ! hand(p,c)>;
              AL2_a^n : <INTERFACE ? return(r,c)>;
              AL3_a^n : <USER ! output(r,c)>
            ]

```

A user that can access node n and request service of "type a " is able to communicate with **AGENT**[a] and pass a parameter list p as part of an object of type **input**. These parameters are then passed to the **INTERFACE**. A typed message is again used (**hand**(p,c)) 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 delivered to the appropriate agent, which passes it back to the user.

2.3. The SURROGATES

The sole responsibility of the **SURROGATES**, is to (locally) process a request for a given type of service. Their array is two-dimensional to account for the fact that for each type of service $D_n = |Nb(n)| + 1$ surrogates are required, to serve the neighbors to node n and the local agent of the same type.

$$SURROGATE[s : 1 \dots S; m : 1 \dots D_n]$$

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, at every node, $S = A$ and, moreover, that there is a natural one-to-one mapping between agent indices and types of services at the server's side. We next give the code for the **SURROGATE** process:

SURROGATE $_{[s,m]}::$

```

 $SL_{s,m}^n$ : *[ <INTERFACE ? deliver ( $p$ , source__node,  $c$ )> →
     $SL1_{s,m}^n$ : process ( $p, r$ );
     $SL2_{s,m}^n$ :
    <INTERFACE ! forward( $r$ , source__node,  $c$ )>
]

```

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

2.4 The INTERFACE

The body of the **INTERFACE** process of node n is given in figure 2-2. **INTERFACE** nondeterministically selects among the following alternatives, not all of which are simultaneously enabled:

- receive a new request ($hand(p_1, c_1)$) from one of the agents. The value of function **target__node** determines the destination node of the request and is stored in the variable **OUTGOING**. Thus, the **INTERFACE** enables communication with the target node and disables communication with all agents, until the parameter list p_1 is transmitted to its destination.
- the value of **OUTGOING** indicates that the request is local; hence it is passed to a **SURROGATE** of the same service type. Communication with the agents is then enabled by setting the value of **OUTGOING** to zero.
- if the value of **OUTGOING** 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, enable communication with the agents again.
- receive a request from a neighboring node and deliver it to the appropriate surrogate.
- if enabled, receive a response from a surrogate and prepare for its transmission to its source node. Variable **INCOMING** functions exactly as **OUTGOING** above.
- The response to a local request is made available; return the result to the appropriate local agent and reenables reception of responses.
- if enabled, transmit a response to its source node; then prepare for reception of another response.
- 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.

Note that some of the i/o guards use computed indices, the CSP extension mentioned in the introduction.

INTERFACE : : **OUTGOING** := 0; **INCOMING** := 0;

```

 $IL^n$ : *[ □ ( $a:1 \dots A$ ) OUTGOING = 0;
    <AGENT[ $a$ ] ? hand( $p_1, c_1$ ) →
    OUTGOING := target__node( $p$  sub 1,  $c$  sub 1);
    aid :=  $a$ >

    □ ( $m:1 \dots D_n$ ) OUTGOING =  $n$ ;
    <SURROGATE[aid,  $m$ ] ! deliver( $p_1, n, c_1$ )
    → OUTGOING := 0 >

    □ ( $k \in Nb(n)$ ) OUTGOING =  $k$ ;
    <NODE[ $k$ ] ! distribute( $p_1, aid, c_1$ ) →
    OUTGOING := 0>

    □ ( $k \in Nb(n)$ ) <NODE[ $k$ ] ? distribute( $p_2, agt, c_2$ )> →
     $IL1^n$ : [□ ( $m:1 \dots D_n$ )
    <SURROGATE [agt,  $m$ ] ! deliver( $p_2, k, c_2$ )> →
    skip ]

    □ ( $s:1 \dots S$ ;  $m:1 \dots D_n$ ) INCOMING = 0;
    <SURROGATE [ $s, m$ ] ? forward( $r_1$ , source__node,  $c_3$ )
    →
    sid :=  $s$ ; INCOMING := source__node >

    □ INCOMING =  $n$ ; <AGENT[sid] ! return( $r_1, c_3$ ) →
    INCOMING := 0 >

    □ ( $k \in Nb(n)$ ) INCOMING =  $k$ ;
    <NODE[ $k$ ] ! respond( $r_1, sid, c_3$ ) →
    INCOMING := 0>

    □ ( $k \in Nb(n)$ ) <NODE[ $k$ ] ? respond( $r_2, sgt, c_4$ )> →
     $IL2^n$ : <AGENT[sgt] ! return( $r_2, c_4$ )>
]

```

Figure 2-2: The **INTERFACE** process

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:

- that the responses that the users receive, match their requests and that every request is processed *at most once* by the network (safety).
- 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 for more subtle cases. Several partial correctness proof systems for CSP have already been proposed in the literature ([1], [3], [10], [12]). We will be using the proof system proposed in [1] and extended in [4] with the addition of histories to record the sequence of communication exchanges between every matching pair of processes. Our formal tool for proving liveness properties will be linear temporal logic [11]. For lack of space, we give only a short survey of the formal systems involved. The reader who is unfamiliar with program verification techniques is encouraged to consult the references.

3.1 Histories

Our *assertion language* is a first order language with equality, in which certain variables are designated as *histories*. Let us consider a pair of i/o commands of the form: $P[e_1] ! T(t_1, \dots, t_n)$ and $P[e_2] ? T(x_1, \dots, x_n)$, taken from processes $P[i]$, $P[j]$ respectively. Here, e_1 and e_2 are integer expressions, $x_1, \dots, x_n$ are variables local to process $P[j]$, $t_1, \dots, t_n$ are terms composed by variables local to process $P[i]$ and T is the type of the message to be exchanged. We associate with such a pair of i/o commands a history T_{ij} . A similar use of histories appears in [6]. We shall use standard notation for operations on sequences, such as concatenation, initial subsequence ($\leq$), etc. We also denote by α^- the history derived from a nonempty history α by deleting its last element. At any given (global) state, each history contains the (finite) sequence of messages exchanged through the communication pair (channel) that it is attached to.

We deal with histories of structured CSP objects, for which selection operators are defined in the obvious way: if $n_1, \dots, n_k$ are the component names of the object $A = T(v_1, \dots, v_k)$, then $A.n_1 = v_1, \dots, A.n_k = v_k$. 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 $\alpha = \langle A_1 \dots A_m \rangle$ then $\alpha.n_i = \langle A_1.n_i \dots A_m.n_i \rangle$. We also form projections of α by selecting some of its objects and deleting projected components of the selected object. In our case the selection criterion is the equality to a constant v of the value of a given function f applied to some particular component n_j .

```

sel ( $\alpha, n_j, f, v$ )  $\triangleq$ 
  if  $\alpha = \langle \rangle$  then  $\langle \rangle$ 
  else if  $v = f(\text{first}(\alpha.n_j))$  then
    ( $\text{first}(\alpha).n_1, \dots, \text{first}(\alpha).n_{j-1},$ 
      $\text{first}(\alpha).n_{j+1}, \dots, \text{first}(\alpha).n_k$ )
    sel( $\text{rest}(\alpha), n_j, f, v$ )
  else sel( $\text{rest}(\alpha), n_j, f, v$ )

```

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 and that α is the history of codes associated with these requests). Then α is *nondecreasing* if $\alpha_i \leq \alpha_j$, whenever $i < j$. We denote this property by $Nd(\alpha)$. Similarly $Mi(\alpha)$ denotes that α is *monotonically increasing*. For nondecreasing histories α and β , we define the *merge* operation as usual with preservation of duplicate elements. Let $S(\alpha)$ denote the set of elements of α .

Lemma 1: If $Mi(\alpha)$ and $Mi(\beta)$ then $S(\alpha^-) \subseteq S(\beta) \subseteq S(\alpha)$ implies $\beta \leq \alpha$.

3.2 A Proof System for Safety Properties

To reason about safety properties of CSP programs, one has first to provide proofs of component processes and then to deduce properties of a parallel program by analyzing the proofs for components. Local properties of a process not defined in terms of other processes are deduced using assertions about local variables and proof techniques for sequential programs. Global safety properties on the other hand, can be expressed through a global invariant I . The invariant may not hold at all global states; program parts that violate I appear *bracketed* in the annotated programs. In [1] a method is provided, called *cooperation test*, to tie separate proofs of processes together into a meaningful whole. This work has been extended in [4] to include computed targets as used here. Every possible process communication has to be checked through a cooperation test. For such a test we assume that the local precondition and I hold before the communication occurs; we then have to establish that the local postcondition and I hold after the communication. The interested reader is referred to [1].

An axiom system with similar ideas is also presented in [10]. The only new extension added here is the following version of the **communication axiom**, relating histories to communications:

$$\{p[t/x] \wedge T_{ij} = \alpha \mid p[i] ? T(x) \parallel p[j] ! T(t) \mid p \wedge T_{ij} = \alpha \leq T(t) \}$$

provided $p[i]?T(x)$ and $p[j]!T(t)$ are taken from $p[j]$ and $p[i]$ respectively. Here T_{ij} is the history of the unidirectional channel permitting flow of information of type T from process i to process j , p is the postcondition local to process j , and x and t are vectors of variables and terms respectively.

3.3 Using Temporal Logic to Prove Liveness Properties

Linear temporal logic was originally introduced as a formal system for reasoning about sequences of events that are totally ordered in time. Recently, this logic has been proposed in [11] and [15] as an appropriate formal system for reasoning about parallel programs. The logic permits the description of a program's execution history without the explicit introduction of time. Moreover, important correctness properties such as mutual exclusion, deadlock freedom, absence of starvation and fairness can be elegantly expressed in this system.

Two *temporal operators* $\Box$ (globally) and $\Diamond$ (eventually) are introduced into standard first order logic. The set of well formed formulae is the smallest set containing the atomic formulae and closed under universal quantification over global variables, boolean connectives, and the above temporal operators. An *interpretation* is a pair $\langle S, i \rangle$ where S is a sequence of states and i specifies the present state. We define the truth of a formula f in an interpretation $\langle S, i \rangle$ (denoted by $S, i \models f$) in the usual inductive way, if f does not contain any temporal operators. The truth of formulae containing temporal operators is defined as follows:

$$S, i \models \Box f \text{ iff } \forall k \geq i, S, k \models f$$

i.e. f is globally true of the interpretation $\langle S, i \rangle$, if f holds in all states occurring after the i -th one.

$$S, i \models \Diamond f \text{ iff } \exists k \geq i, S, k \models f$$

i.e. f is eventually true of the interpretation $\langle S, i \rangle$ if there is a state k occurring after the i -th one where f holds. Finally, f is *valid* iff for all state sequences S and for all $i \geq 0$, $S, i \models f$.

Consider the program,

$$P :: [P_1 \parallel \dots \parallel P_n]$$

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 . For $\ell \in L^i$, 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 . For a formal definition of these notions for CSP, see [9].

3.4 Proving liveness properties under fairness assumptions

The issue of fairness is only briefly discussed in [8], where it is argued that it is the programmer's responsibility to prove that his (or her) program behaves correctly, without making any fairness assumptions whatsoever. Unless, however, the details of the various schedulers were explicitly modelled - something that would have unduly complicated the code of the protocol - it would be impossible to establish the desired liveness properties of the SRP protocol without making any assumptions about fairness.

In [5] and later in [9], several kinds of fairness are distinguished: fairness associated with eventual advance of each process, fairness associated with eventual occurrence of each pending communication, etc. We adopt a weak version of the *channel* fairness assumption, as explained below. In the sequel, for simplicity of notation, we consider only one array of processes which we call P .

Let $\ell \in L^i$ be a label of an alternative, repetitive or a communication command in process P_i . Let m be the number of guards g_a , $a = 1, \dots, m$ 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:

$$\ell : [\Box b_1; C_1 \rightarrow \ell_1; S_1 \Box \dots \Box b_m; C_m \rightarrow \ell_m; S_m]$$

The channel fairness axiom can be described as follows (the reader is referred to [9] for its formal statement in linear temporal logic):

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 another given process, and ℓ' denote a given guarded command in P_j . If it is continuously the case that the boolean part of g_a is true and that the communication part of g_a matches to the communication part of some guard of ℓ' whose boolean part is also true, then eventually this specific communication will occur.

Again deviating from the original semantics of CSP, as presented in [8], 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.

The predicate $\text{matching}(\ell, a)$ is 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 . For the terminating constructs to be used, it is possible to recursively define a predicate $\text{terminate}(\ell)$ ([14]) satisfying:

$$\text{at } \ell \wedge \text{terminate}(\ell) \supset \Diamond \text{ after } \ell$$

As an example of the use of terminate , we give a condition asserting that a non-terminating repetitive command labelled ℓ , will never be blocked within its body, and moreover, that control will be at ℓ infinitely often:

$$\Diamond \text{ at } \ell \wedge$$

$$\begin{aligned} & \Box \left[\text{at } \ell \supset \bigvee_{a=1}^m b_a \wedge \left(\bigwedge_{a=1}^m (\text{matching}(\ell, a) \supset \text{terminate}(S_a)) \right) \right] \\ & \supset \Box \Diamond \text{ at } \ell \wedge \Box (\neg \text{after } \ell) \end{aligned}$$

Observe that the condition above is also established in the trivial case, where 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 $\Diamond \Box \text{ at } \ell$ holds.

4. Analysis of the SRP Protocol

In this section, we specify and prove some of the more inter-

esting safety properties of SRP. Initially, all histories are assumed to be empty. We form a global invariant I , and show that it holds whenever processes are outside their bracketed sections. The global invariant describes the message flow through the various nodes of the network and illustrates a *conservation* property, namely that if a message is received, then it should have been sent by the appropriate sender in the right order. That every sent message is indeed received is the liveness property of SRP, to be shown in the sequel. The reader is encouraged to consult figure 2-1 while interpreting the invariant. I is the conjunction of several clauses, that we list below:

$$Mi(\text{input}_a^n, C) \quad (1)$$

We first assert that the codes associated with incoming requests to be handled by agent a at node n are monotonically increasing.

$$\text{hand}_a^n \leq \text{input}_a^n \quad (2)$$

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

$$\text{output}_a^n \leq \text{return}_a^n \quad (3)$$

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

$$\text{forward}_{s,m}^{dn}.C \leq \text{deliver}_{s,m}^{dn}.C \quad (4)$$

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

$$\text{sel}(\text{distribute}_{dn}^{sn}, AID, id, a).C \leq \quad (5)$$

$$\text{sel}(\text{hand}_a^{sn}, P, \text{target_node}, dn).C$$

where id stands for the appropriate identity function. Here, we select those requests listed in history *distribute* which come from agent a , at source node sn . We state that the thereby formed projection, is an initial subsequence of the history containing codes of requests coming from agent a with destination node dn (for $dn \neq sn$). Clause (6) refers to locally served requests:

$$\text{Merge}_{m=1}^{D_n} (\text{sel}(\text{deliver}_{a,m}^{sn}, SN, id, sn).C) \leq \quad (6)$$

$$\text{sel}(\text{hand}_a^{sn}, P, \text{target_node}, sn).C$$

$$\text{Merge}_{m=1}^{D_n} (\text{sel}(\text{deliver}_{a,m}^{dn}, SN, id, sn).C) \leq \quad (7)$$

$$\text{sel}(\text{distribute}_{dn}^{sn}, AID, id, a).C$$

In (7) we consider the histories of all channels which deliver requests to all surrogates of type a , $a \in \{1, \dots, S\}$ at node dn . We form their projections by selecting only those requests (of type a) coming from node sn and merge them. We then state that the thus formed history is an initial subsequence of the *distribute*-subhistory containing requests of type a . The following three clauses are essentially analog to 5, 6 and 7.

$$sel(respond_{sn}^{dn}, AID, id, a).C \leq \quad (8)$$

$$\begin{aligned} & \text{Merge}_{m=1}^{D_n} (sel(forward_{a,m}^{dn}, SN, id, sn).C) \\ & sel(return_a^{sn}, R, \text{target_node}, sn).C \leq \quad (9) \end{aligned}$$

$$\begin{aligned} & \text{Merge}_{m=1}^{D_n} (sel(forward_{a,m}^{sn}, SN, id, sn).C) \\ & sel(return_a^{sn}, R, \text{target_node}, dn).C \leq \quad (10) \\ & sel(respond_{sn}^{dn}, AID, id, a).C \end{aligned}$$

In order to establish the truth of the invariant I we have to prove that I holds initially and outside the bracketed sections of every process. Clearly, I holds initially because all histories are empty.

To illustrate how the invariance of I is established we prove the invariance of (5). Let

$$\alpha = sel(distribute_{dn}^{sn}, AID, id, a).C \quad (11)$$

and

$$\beta = sel(hand_a^{sn}, P, \text{target_node}, dn).C \quad (12)$$

In order to establish (5) we actually prove the invariance of the stronger assertion (once the relevant histories cease to be empty):

$$\begin{aligned} & (OUTGOING \neq dn \supset \beta^- < \alpha = \beta) \wedge \\ & (OUTGOING = dn \supset \beta^- = \alpha < \beta) \end{aligned}$$

There are two families of communications that affect the histories of α and β . The first is a communication between some agent a in node sn and the **INTERFACE** of the same node. We assume that the invariant holds before the communication occurs. Inspection of the code for the **INTERFACE** (first alternative) shows that $OUTGOING \neq dn$ at this point. Therefore we can immediately obtain $\beta^- < \alpha = \beta$. The effect of the communication is to increase β 's length by one element and set $OUTGOING$ to dn . Thus after the addition of an element to β , the relation $\beta^- = \alpha < \beta$ clearly holds and the first conjunct holds vacuously. The second communication is between the **INTERFACE** of node sn and (the **INTERFACE** of) a different node dn . Inspection of the third alternative of the **INTERFACE** shows that $OUTGOING = dn$ at this point. Thus from the invariant we obtain $\beta^- = \alpha < \beta$. The effect of the communication is to extend α by one element (which by local reasoning can be shown to be the last element of β) and to set $OUTGOING$ to zero; we obtain $\beta^- < \alpha = \beta$ and the second conjunct holds again vacuously. This establishes the invariance of (5). The reader familiar with [1] will recognize this argument as an instance of a cooperation test. The rest of the clauses are similarly established and will be left for an extended version of the paper.

Once the invariance of I is established, one may prove the main safety property of SRP as expressed by

Theorem 1:

1. Every request submitted to a node's driver can reach at most one surrogate at the server's side:

$$\begin{aligned} & \text{Merge}_{m=1}^{D_n} (sel(deliver_{a,m}^{dn}, SN, id, sn).C) \\ & \leq sel(input_a^{sn}, P, \text{target_node}, dn).C \quad (13) \end{aligned}$$

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

$$output_a^{sn}.C \leq input_a^{sn}.C \quad (14)$$

Proof: The basic ideas behind the details of the proof are the application of the transitivity of the " $\leq$ " relation, as well as the known properties of merging and projecting.

1. (2) can be rewritten as follows, given that sel and component selection preserves the prefix relationship of two histories:

$$sel(hand_a^{sn}, P, \text{target_node}, dn).C \leq \quad (15)$$

$$sel(input_a^{sn}, P, \text{target_node}, dn).C$$

Combining (5) and (15) we get:

$$sel(distribute_{dn}^{sn}, AID, id, a).C \leq \quad (16)$$

$$sel(input_a^{sn}, P, \text{target_node}, dn).C$$

(7) and (16) yield (13) by transitivity of " $\leq$ ". Observe that if $dn = sn$ (local service) then (13) results from (6) and (15).

2. Using (4), the fact that $forward_{a,m}^{dn}.C$ and $deliver_{a,m}^{dn}.C$ are monotonically increasing, and that $Merge$ and sel preserve the prefix relationship of histories, we obtain:

$$\text{Merge}_{m=1}^{D_n} (sel(forward_{a,m}^{dn}, SN, id, sn).C) \leq \quad (17)$$

$$\text{Merge}_{m=1}^{D_n} (sel(deliver_{a,m}^{dn}, SN, id, sn).C)$$

Combination of (13), (17), (8) and (9) yield:

$$sel(return_a^{sn}, R, \text{target_node}, dn).C \leq \quad (18)$$

$$sel(input_a^{sn}, P, \text{target_node}, dn).C$$

Note, again, that if $sn = dn$ then (18) results from (13), (17) and (9). Local reasoning, based on the code of the **INTERFACE**, shows that no request is dispatched to more than one node. Thus, $sel(return_a^{sn}, R, \text{target_node}, dn).C$, for $dn \in Nb(sn)$ are disjoint histories, and clearly the same holds for $sel(input_a^{sn}, P, \text{target_node}, dn).C$. It follows that we can infer (14) from (18).

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 a node of the network, then the output at this node is an unbounded number of responses. We formalize the continuous availability of the **USER** processes by the following assertion:

$$\Box((at AL_a^n \supset \Diamond at AL1_a^n)$$

$$\wedge (at AL3_a^n \supset \Diamond after AL3_a^n))$$

The truth of this assertion should be established by an appropriate axiom characterizing the user program and the channel fairness axiom. Since we want to avoid referring to the user program, we take this assertion as a starting point.

In [14] it is proved that **INTERFACE** cannot be blocked when it has a message to output to either an agent or a surrogate:

Lemma 2:

$$\begin{aligned} & \Box [\text{at } IL1^n \supset \Diamond \text{ after } IL1^n] \\ & \Box [\text{OUTGOING} = n \supset \Diamond \text{OUTGOING} = 0] \\ & \Box [\text{at } IL2^n \supset \Diamond \text{after } IL2^n] \\ & \Box [\text{INCOMING} = n \supset \Diamond \text{INCOMING} = 0] \end{aligned}$$

An immediate consequence of lemma 2 and the non-terminating repetitive command condition is the following lemma:
Lemma 3:

$$\Box \Diamond \text{at } IL^n \quad (19)$$

Proof: Inspection of the code of **INTERFACE** shows that **OUTGOING** and **INCOMING** range over the set $Nb(n) \cup \{0, n\}$. Hence at any time, one of the first, second or third group of alternatives and one of the fifth, sixth or seventh group of alternatives will be enabled. By using lemma 2 and the fact that all of the statements which follow guards in the **INTERFACE** are loop-free, we can apply the non-terminating repetitive command proof rule and conclude (19).

Q.E.D.

The next lemma shows that communication of the **INTERFACE** with the agents (surrogates) is enabled infinitely often.

Lemma 4: $\Box \Diamond \text{OUTGOING} = 0$, $(\Box \Diamond \text{INCOMING} = 0)$

Proof: Assume the contrary. Therefore, from some state on:

$$\begin{aligned} & \Diamond \Box \exists k \in Nb(n) \bigcup \{n\} (\text{OUTGOING} = k) \\ & (\Diamond \Box \exists k \in Nb(n) \bigcup \{n\} (\text{INCOMING} = k)) \end{aligned}$$

If $k=n$ then a contradiction is reached by lemma 2; otherwise, according to lemma 3, at node k the fourth (eighth) group of alternatives of **INTERFACE** is infinitely often enabled; hence, we can apply the channel fairness axiom and deduce that communication between the interfaces of nodes n and k will occur. Immediately after this communication **OUTGOING** (**INCOMING**) will be assigned zero, 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 infinitely often submitting requests always ready to accept their responses, then the collection of agents infinitely often produce outputs to the users.*

$$\forall sn (1 \leq sn \leq N). \forall t (1 \leq t \leq A). \quad (20)$$

$$\Box ((\text{at } AL_t^{sn} \supset \Diamond \text{at } AL_t^{sn}) \wedge (\text{at } AL_t^{sn} \supset \Diamond \text{after } AL_t^{sn}) \supset$$

$$\Box \Diamond \text{at } AL_t^{sn})$$

Proof: Consider agent t , who initially is at AL_t^{sn} . Because of 23, $\Diamond \text{at } AL_t^{sn}$. By lemma 4 and the fairness axiom, communication between the agent t and **INTERFACE** has to occur; thus $\Diamond \text{at } AL_t^{sn}$. If $dn = \text{target_node}(p)$ 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 channel fairness axiom. Thus at **NODE** $[dn]$, **INTERFACE** is at $IL1^{dn}$, by lemma 2 communication with some **SURROGATE**

$[t, m]$ will occur, and again $\Diamond \text{at } IL^{dn}$ by lemma 2, **SURROGATE** $[t, m]$ will go to $SL2_{t, m}^{dn}$ after processing the request, and by lemma 4 communication with the **INTERFACE** will occur, therefore enabling the sn -th branch of the fifth group of alternatives in **INTERFACE**. Again by lemma 3 nodes sn and dn will communicate and the response will reach node sn with **INTERFACE** at that node being at $IL2^{sn}$. By lemma 2 communication with agent t will occur, therefore bringing agent t at $AL3_t^{sn}$. By (20):

$$\Box [\text{after } AL3_t^{sn} \supset \Diamond \text{at } AL_t^{sn}]$$

Thus, we proved that:

$$\Box [\text{at } AL_t^{sn} \supset \Diamond \text{at } AL_t^{sn}]$$

which means that:

$$\forall sn (1 \leq sn \leq N). \forall t (1 \leq t \leq A) \Box \Diamond \text{at } AL_t^{sn}$$

Q.E.D.

5. Conclusion

We believe that the proof methodology used in this paper is of wide applicability to systems of processes modelling network protocols. For example, one could envision a systematic way, whereby given a system of processes and their communication channels, one deduces invariances of these channel histories similar to the ones given in section 4. We believe that the verification of certain classes of concurrent systems (e.g. those modelling network protocols) can be reduced to a calculus of process communication histories. Further research is needed in this area.

6. Acknowledgements

Brent Hailpern and Anders Ravn read early drafts of the paper and made several helpful remarks and corrections.

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., Extended naming conventions in communicating processes. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, January 1982.
- [5] Francez N., De Roever W. P. Fairness in communicating processes. 1980. Extended Abstract.
- [6] Hailpern B. *Verifying Concurrent Processes Using Temporal Logic*. PhD thesis, Stanford University, August, 1980.
- [7] Hoare C. A. R. A calculus for total correctness of communicating processes. *Science of Programming*, 1, 1982
- [8] Hoare C. A. R. Communicating Sequential Processes. *CACM* 21(8):666-677, August 1978.
- [9] Kuiper R., De Roever W.P. *Fairness assumptions for CSP in a temporal logic framework*. Technical Report, dept. of Computer Science, University of Utrecht, 1982.
- [10] Levin G., Gries D. A Proof Technique for Communicating Sequential Processes. *Acta Informatica* 15:281-302, 1981.
- [11] Manna Z. Pnueli A. *Verification of concurrent programs: the temporal framework*. Technical Report STAN-CS-81-836, Stanford University, June, 1981.

- [12] Misra J. Chandy K. M. Proofs of networks of processes. *IEEE Transactions on software Engineering* SE-7(4):417-426, July, 1981.
- [13] Nelson B.J. *Remote Procedure Call*. Report CSL-81-9, Xerox Palo Alto Research Center, 1981.
- [14] Nikolaou C. N. *Reliability Issues in Distributed Systems*. PhD thesis, Harvard University, 1982.
- [15] Owicki S., Lamport L. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems* 4(3):455-495, July 1982.
- [16] Schuman S. A., Clark 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.
- [17] Schwartz R.L., Melliar-Smith P.M. From State Machines to Temporal Logic: Specification Methods for Protocol Standards. To appear in *IEEE Transactions on Networks*.
- [18] 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.