Using Message Passing for Distributed Programming: Proof Rules and Disciplines

RICHARD D. SCHLICHTING
University of Arizona
FRED B. SCHNEIDER
Cornell University

Inference rules are derived for proving partial correctness of concurrent programs that use message passing. These rules extend the notion of a satisfaction proof, first proposed for proving correctness of programs that use synchronous message-passing, to asynchronous message-passing, rendezvous, and remote procedures. Two types of asynchronous message-passing are considered: unreliable datagrams and reliable virtual circuits. The proof rules show how interference can arise and be controlled.

Categories and Subject Descriptors: C.2.2 [**Computer-Communications Networks**]: Network Protocols—*protocol verification*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.4.4 [**Operating Systems**]: Communications Management—*network communications*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants, pre- and postconditions*

General Terms: Verification

Additional Key Words and Phrases: Message passing, satisfaction proofs, unreliable datagram service, reliable virtual circuit service, remote procedure call, rendezvous

1. INTRODUCTION

Message passing provides a way for concurrent processes to communicate and synchronize. In this paper, proof rules are developed for a variety of message-passing primitives. Two benefits accrue from this. The obvious one is that partial correctness proofs can be written for concurrent programs that use such primitives. This allows these programs to be understood as predicate transformers [6] or invariant maintainers[1] [16], instead of by contemplating all possible execution

[1] An *invariant* is an assertion that is true throughout execution of a program. A program S is an *invariant maintainer* for assertion I if I is an invariant of S.

This work was supported in part by National Science Foundation Grant MCS-81-03605 and MCS-82-02869.

Authors' present addresses: R. D. Schlichting, Department of Computer Science, University of Arizona, Tucson, AZ 85721; F. B. Schneider, Department of Computer Science, Cornell University, Ithaca, NY 14853.

interleavings. The second benefit is that the proof rules shed light on how interference arises when message-passing operations are used and on how this interference can be controlled. This provides insight into techniques to control interference in programs that involve asynchronous activity.

Our work is based on an approach first proposed by Levin and Gries [17] for proving communicating sequential processes (CSP) [13] programs correct. By extending and applying their notion of a satisfaction proof, we have been able to derive proof rules for a variety of communications primitives, including asynchronous message-passing with unreliable datagrams and with reliable virtual circuits, rendezvous, and remote procedures. Moreover, the approach appears to be quite general and therefore could be used to obtain proof rules for other message-passing statements, as well.

In Section 2 we briefly review CSP and satisfaction proofs. In Section 3, proof rules for communications primitives of networks supporting unreliable datagrams [23] are derived. In Section 4, we consider communications primitives of networks that support reliable virtual circuits. Section 5 contains proof rules for rendezvous primitives and remote procedures. Finally, Section 6 discusses related work and Section 7 contains conclusions.

2. SYNCHRONOUS MESSAGE-PASSING AND SATISFACTION PROOFS

2.1 CSP

A CSP program P is a collection of processes $P_1, P_2, \ldots, P_n$:

$P :: [P_1 \| P_2 \| \cdots \| P_n]$.

Processes are executed concurrently; they communicate and synchronize solely by using input and output commands.

An *input command*

inp: A?var

in process B matches an *output command*

out: B!expr

in process A if the type of expr and var are the same. Input and output commands are always executed synchronously in matching pairs, so each has the potential to delay its invoker. This is called *synchronous message-passing*, in contrast to *asynchronous message-passing* where the sender is not delayed. Execution of a pair of input and output commands is equivalent to the assignment var := expr. Thus, a matching pair of input and output commands implements a *distributed* assignment statement.

In CSP, a communications command can appear in the guard of a selection or iteration command.[2] This allows a process to wait for the occurrence of any one of a number of events. A guard consists of a Boolean expression β, which involves only program variables accessible to the process in which the guard appears, and

[2] In the language described in [13] only input commands can appear in guards. It is convenient to consider the more general language in which either an input command or an output command can appear in a guard.

an optional communications statement *com*. The guard is false if β is false. The guard is true if β is true and execution of *com* would not be delayed.[3]

2.2 A Proof System for CSP

Proofs in the programming logic for CSP described in [17] involve three steps. First, each process is annotated with assertions, giving a *sequential proof*. Then, assumptions made in the sequential proof about the effects of receiving messages are validated by performing a *satisfaction proof*. This involves constructing a collection of *satisfaction formulas* and proving them to be valid. Finally, *noninterference* [21] is established to ensure that execution of each process cannot invalidate assertions that appear in the sequential proof of another.

The sequential proof for each process is obtained using a sequential programming logic (such as that in [12]) along with axioms for communications commands (described below). Thus, each process is viewed as a sequential program in isolation. In a sequential proof, the assertion immediately preceding a statement S is called the *precondition* of S and is denoted pre(S); the assertion following S is called the *postcondition* of S and is denoted post(S).

Free variables in assertions are either *program variables* or *auxiliary variables* [4]. For simplicity, distinct variables are assumed to have distinct names. Program variables can appear only in statements in processes that have access to the memory in which those variables are stored.[4] However, any variable can be named in assertions in the sequential proof of any process; this permits the states of different processes to be correlated.

The axiom for an input command inp is

Input Command Axiom. $\{P\}$ inp: A?var $\{R\}$,

and the axiom for an output command out is

Output Command Axiom. $\{Q\}$ out: B!expr $\{U\}$.

Each allows anything to appear as its postcondition; in the parlance of [6], the axioms violate the "Law of the Excluded Miracle." However, this is not a problem. When executed *in isolation*, a communications command cannot terminate, and so the soundness of the axioms for sequential proofs follows.[5] A satisfaction proof will impose further restrictions on the postconditions of communications commands so that soundness is preserved even when communications commands do terminate.

Note that communications commands in guards can be treated in the same way as other communications commands if deadlock is not of concern. Consider the guarded commands

$$b; com \rightarrow S$$

[3] Following [17], we do not adopt Hoare's convention of having a guard containing a communications command to be false if the process named in that communication command has terminated.

[4] In CSP, program variables that can be altered by one process may not be accessed by concurrently executed processes. However, the proof systems described in this paper can be used to reason about programs in which there are shared variables as well as message passing. We exploit this in Section 4.

[5] Recall that this is a partial correctness logic.

and

$$b \rightarrow com; S.$$

The only difference between these is that the latter is more prone to deadlock than the former [17]. Since termination and deadlock are not reflected in a proof system for partial correctness, it is always possible to "translate" a program from one form to the other for the purposes of constructing a proof of partial correctness. A way to adapt such a proof for proving absence of deadlock is given in [17].

In order to understand the obligations for establishing satisfaction, consider matching pair of communications commands inp and out. According to the communications axioms, their execution will leave the system in a state in which $R \wedge U$ is true. Since execution of the matching pair is equivalent to executing var := expr, it suffices for execution to be started in a state that satisfies wp("var := expr", $R \wedge U$) to ensure that the desired postcondition will result [6]. However, the communications axioms stipulate that $P \wedge Q$ is true of the state immediately before the assignment is made. Thus, the truth of the postconditions of a matching pair of communications commands is ensured if

$$(P \wedge Q) \Rightarrow \text{wp}(\text{"var} := \text{expr"}, R \wedge U)$$

or equivalently, if the satisfaction formula[6]

$$\text{Sat}_{\text{synch}}(\text{inp, out}): (P \wedge Q) \Rightarrow (R \wedge U)^{\text{var}}_{\text{expr}}$$

is valid. A satisfaction proof therefore involves

CSP Satisfaction Proof. For every pair of matching communications commands inp and out, prove $\text{Sat}_{\text{synch}}(\text{inp, out})$ valid.

This ensures that the postconditions of communications commands in a sequential proof are true whenever those communications commands terminate.

The third and final step of the proof is to show noninterference. This is necessary because assertions in the proof of one process can refer to variables changed by another.[7] In order to combine the proofs of a collection of processes that will be executed concurrently, it is therefore necessary to show that execution of no process invalidates the proof of another.

An assertion I and atomic action S are *parallel* if S is contained in one process and I is contained in the proof of another. A statement is an atomic action if it involves at most one reference to at most one shared variable. To establish noninterference, it must be shown that execution of every atomic action S parallel to I does not make I false. We say that S *does not interefere with* I if

$$\text{NI}_{\text{synch}}(S, I): \{I \wedge \text{pre}(S)\}\ S\ \{I\}$$

is a theorem in the programming logic. Thus, noninterference is established by proving that no assignment or communications command interferes with any assertion parallel to it.

---

[6] The notation P^x_E denotes the substitution of E for every free occurrence of x in P.

[7] In [17] auxiliary variables can be subject to concurrent access.

For an input command S, the proof that $\mathrm{NI}_{\mathrm{synch}}(S, I)$ is a theorem follows trivially from the Input Command Axiom. However, satisfaction must then be established to ensure the truth of the postcondition of $\mathrm{NI}_{\mathrm{synch}}(S, I)$. This is done by showing that for every output command out that matches input command S and that is parallel to I, the following satisfaction formula is valid.

$$\mathrm{NI\_Sat}_{\mathrm{synch}}(S, \mathrm{out}, I)\text{: } (I \wedge \mathrm{pre}(S) \wedge \mathrm{pre}(\mathrm{out})) \Rightarrow I^{\mathrm{var}}_{\mathrm{expr}}.$$

$\mathrm{NI\_Sat}_{\mathrm{synch}}$ is obtained from $\mathrm{Sat}_{\mathrm{synch}}$ by substituting $I \wedge \mathrm{pre}(S)$ for P, pre(out) for Q, I for R, and true for U. Proving this formula valid corresponds to showing that executing the distributed assignment implemented by communication commands S and out does not invalidate an assertion I in a third process.

Similarly, for S an output command $\mathrm{NI}_{\mathrm{synch}}(S, I)$ is a theorem, but satisfaction must be established. This is done by showing that $\mathrm{NI\_Sat}_{\mathrm{synch}}(\mathrm{inp}, S, I)$ is valid for every input command inp that matches output command S and that is parallel to an assertion I in a third process.

Thus, noninterference is established by proving

CSP Noninterference Proof

For every atomic action S and assertion I parallel to S, prove $\mathrm{NI}_{\mathrm{synch}}(S, I)$.

For every input command S_{inp} and matching output command S_{out}, and every assertion I parallel to S_{inp} and S_{out}, prove $\mathrm{NI\_Sat}_{\mathrm{synch}}(S_{\mathrm{inp}}, S_{\mathrm{out}}, I)$ valid.

The following inference rule then allows proofs of processes to be combined.

Concurrent Execution

$$\frac{\{P1\}S1\{Q1\},\ \{P2\}S2\{Q2\},\ \cdots\ \{Pn\}Sn\{Qn\},\ \text{Satisfaction, Noninterference}}{\{P1 \wedge P2 \wedge \cdots \wedge Pn\}[S1 \,\|\, S2 \,\|\, \cdots \,\|\, Sn]\{Q1 \wedge Q2 \cdots \wedge Qn\}}.$$

3. COMMUNICATION WITH UNRELIABLE DATAGRAMS

Real communications networks, which buffer messages, do not ensure reliable delivery of messages and do not ensure that messages are delivered in the order sent. A message might be discarded because its contents have been detectably corrupted by noise. Two messages from one site to another could travel different routes and therefore might not arrive in the order sent. To mitigate these difficulties, each site in the network typically has software that converts such an *unreliable datagram service* into a facility that ensures that messages are delivered reliably and in order [23]. This network software can be viewed as a concurrent program, where processes synchronize and communicate using the unreliable datagram service. Hence the motivation for developing proof rules for message-passing primitives discussed in this section.

It is useful to distinguish between *sent*, *delivered*, and *received* when describing the status of a message. Execution of a **send** statement causes a message to be *sent*. A message that has been sent might subsequently be *delivered* to its destination, according to the following:

UD1: Messages that are sent are not necessarily delivered.

UD2: Messages between a pair of processes are not necessarily delivered in the order sent.

Once delivered, a message can be *received* by excuting a **receive** statement.

The **send** statement

send expr **to** dest

is executed as follows. First, the value of expression expr is computed. Then, a message with that value is sent to the process named dest. Note that **send** is asynchronous—unlike the output command of CSP, the sender continues while the message is being delivered and received. Also, note that we do not prohibit a process from sending a message to itself. (This would cause the process to block forever in CSP.)

A **receive** statement has the form

receive m **when** β

where m is a program variable and β is a Boolean expression involving m and other program variables. Execution of this statement delays the invoker until a message with value MTEXT (say) has been delivered to the invoking process, and β^m_{MTEXT} = true. Execution of **receive** completes by assigning MTEXT to m.

It is unlikely that such a receive would actually be implemented. The expense and complexity of "peeking" at the contents of delivered messages in order to evaluate β^m_{MTEXT} is too high. On the other hand, the use of β does make it possible to model a variety of communications primitives, such as receive statements, in which strong-typing of the target variable and the message is enforced at run time, and receive statements for named channels. Thus, considering this more general **receive** is sensible.

3.1 Proof Rules

Following the approach outlined in Section 2, a program proof involves three steps: a sequential proof, a satisfaction proof, and a noninterference proof. The proof obligations for each of these steps—including axioms for **send** and **receive**, the satisfaction formula for these primitives, and the noninterference satisfaction formula—are now described.

The state of the system includes information about messages that have been sent but not received, since this can influence execution. In order to *model* this aspect of the state, two auxiliary variables are associated with each process D. The *send multiset* σ_D for process D contains a copy of every message sent to D; the *receive multiset* ρ_D for process D contains a copy of every message received by D.[8] A message can be received only if it has been sent and delivered. Therefore,

Unreliable Datagram Network Axiom. $(\forall D\colon D \text{ a process}\colon \rho_D \subseteq \sigma_D)$.

Lost messages (see UD1 above) are modeled by having them remain forever in $\sigma_D \ominus \rho_D$, where $\ominus$ is the multiset difference operator; out-of-order messages (UD2)

[8] A *multiset*—sometimes called a "bag"—is like a set but can contain more than one instance of the same element.

are modeled because messages available for receipt at any time are in $\sigma_D \ominus \rho_D$, an unordered multiset.

3.1.1 *Axioms for Send and Receive.* Executing

send expr **to** D

is the same as executing the assignment

$$\sigma_D := \sigma_D \oplus \text{expr}$$

where "$\sigma_D \oplus$ expr" denotes the multiset consisting of the elements of σ_D plus an element with value expr. Using wp with respect to postcondition W, we get an axiom for the **send** statement:

Unreliable Datagram Send Axiom. $\{W^{\sigma_D}_{\sigma_D \oplus \text{expr}}\}$ **send** expr **to** D $\{W\}$.

When execution of

receive m **when** β

in process D terminates, β is true. In addition, depending on the particular message received, it may be possible to make some assertion about the state of the sender. An axiom that captures this is

Unreliable Datagram Receive Axiom. $\{R\}$ **receive** m **when** β $\{Q \wedge \beta\}$.

In the course of establishing satisfaction, restrictions are imposed on Q.

3.1.2 *Establishing Satisfaction.* Consider a **receive** statement

r: **receive** m **when** β

in process D. According to the operational semantics for **receive** given above, in order for execution of r to result in the receipt of a message with value MTEXT (say), then (1) β^m_{MTEXT} must be true and (2) $\text{MTEXT} \in (\sigma_D \ominus \rho_D)$, that is, a message with value MTEXT must have been sent to D but not yet received. Thus, immediately before MTEXT is assigned to m, the system state can be characterized by

$$\text{pre}(r) \wedge \beta^m_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D). \tag{3.1.1}$$

Execution of r resulting in receipt of a message with value MTEXT is equivalent to execution of the multiple assignment statement

$$m, \rho_D := \text{MTEXT}, \rho_D \oplus \text{MTEXT}.$$

For this assignment to establish the postcondition $Q \wedge \beta$ of the Unreliable Datagram Receive Axiom, execution must be performed in a state satisfying

$$\text{wp}(\text{"}m, \rho_D := \text{MTEXT}, \rho_D \oplus \text{MTEXT"}, Q \wedge \beta),$$

which is

$$(Q \wedge \beta)^{m, \rho_D}_{\text{MTEXT}, \rho_D \oplus \text{MTEXT}}.$$

Since ρ_D, an auxiliary variable, is not free in β, this is

$$Q^{m, \rho_D}_{\text{MTEXT}, \rho_D \oplus \text{MTEXT}} \wedge \beta^m_{\text{MTEXT}}. \tag{3.1.2}$$

Thus, post(r) will be true when r terminates provided (3.1.1) $\Rightarrow$ (3.1.2), or

$$\text{Sat}_{\text{asynch}}(r)\text{: } (\text{pre}(r) \wedge \beta^{m}_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

is valid. Note that MTEXT is a free variable in $\text{Sat}_{\text{asynch}}(r)$, so it is implicitly universally quantified. This corresponds to the fact that $Q^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$ must be true for any message that could be received by executing r. A satisfaction proof requires:

Unreliable Datagram Satisfaction Proof. For every **receive** statement r, prove $\text{Sat}_{\text{asynch}}(r)$ valid.

3.1.3 *Establishing Noninterference.* As before, noninterference is established by proving additional theorems

$$\text{NI}_{\text{asynch}}(S, I)\text{: } \{I \wedge \text{pre}(S)\}\ S\ \{I\}$$

about every assertion I and every assignment, **send**, and **receive** statement S parallel to it. For S a **receive** in process D, $\text{NI}_{\text{asynch}}(S, I)$ follows trivially from the Unreliable Datagram Receive Axiom, but it is necessary to prove satisfaction. The necessary satisfaction formula $\text{NI\_Sat}_{\text{asynch}}(S, I)$ is obtained by substituting into $\text{Sat}_{\text{asynch}}(r)$ based on $\text{NI}_{\text{asynch}}(S, I)$—that is, using $I \wedge \text{pre}(S)$ for pre(r) and I for Q. This results in:

$$\text{NI\_Sat}_{\text{asynch}}(S, I)\text{: } (I \wedge \text{pre}(S) \wedge \beta^{m}_{\text{MTEXT}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow I^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}.$$

In summary, to establish noninterference

Unreliable Datagrams Noninterference Proof

For every assignment and **send** statement S and every assertion I parallel to S, prove $\text{NI}_{\text{asynch}}(S, I)$.

For every **receive** S and every assertion I parallel to S, prove $\text{NI\_Sat}_{\text{asynch}}(S, I)$ valid.

3.2 Disciplines to Simplify Establishing Satisfaction

Some general techniques that allow satisfaction to be established trivially are now illustrated by means of a series of examples.

Consider the following distributed program, where process A sends a message to process D.

A :: $\{\sigma_D = \Phi\}$
  $x := 16$;
  $\{\sigma_D = \Phi \wedge x = 16\}$
  s: **send** x **to** D
  $\{16 \in \sigma_D \wedge x = 16\}$

D :: $\{\rho_D = \Phi\}$
  r: **receive** m **when** true
  $\{m \in \rho_D\}$

Initially, $\sigma_D = \rho_D = \Phi$ so the above is a valid sequential proof. To establish

satisfaction, $\text{Sat}_{\text{asynch}}(r)$ is constructed and proved valid. It is

$$(\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (m \in \rho_D)^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

$$= (\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow \text{MTEXT} \in (\rho_D \oplus \text{MTEXT}).$$

The consequent is equivalent to true, so the formula is valid and satisfaction is shown. Noninterference follows trivially: no assertion in the sequential proof of one process contains a variable modified by another.

Now suppose post(r) is changed to reflect the fact that the value of the message received is 16:

D:: $\{\rho_D = \Phi\}$
  r: **receive** m **when** true
  $\{m = 16\}$

The sequential proof is still valid—according to the Unreliable Datagram Receive Axiom, anything can be asserted after a **receive**. The new satisfaction formula for r is:

$$(\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (m = 16)^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

$$= (\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (\text{MTEXT} = 16).$$

This is not valid. Without changing post(r), there are two options:

(1) Strengthen β (by using "$m = 16$" for β) so that the resulting satisfaction formula would be valid.
(2) Strengthen pre(r) so that the resulting satisfaction formula would be valid.

Option (1) certainly guarantees that only a message with value 16 will be received. However, this should not be necessary, because only a message with value 16 is sent. Let us pursue option (2).

First, note that if some assertion I is an invariant of a program, then I can be used to strengthen the assertions in a valid sequential proof of that program and the result will remain a valid sequential proof. This forms the basis of a general strategy:

General Strategy for Establishing Satisfaction. Strengthen assertions by adding an invariant of the program as a conjunct to every assertion. By suitable choice of the invariant, the new satisfaction formula can be proved valid.

This strategy is exploited, as follows. Define

$$I_{\text{vals}}: (\forall M: M \in \sigma_D: M = 16).$$

I_{vals} is an invariant of process A; it is true initially ($\sigma_D = \Phi$) and, using the Send Axiom, we find that its truth before execution of s implies its truth after. Since I_{vals} is not interfered with by any statement in D, it is also an invariant of the entire concurrent program. After strengthening assertions in D with I_{vals}, the new satisfaction formula for r is

$$(\rho_D = \Phi \wedge I_{\text{vals}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (m = 16 \wedge I_{\text{vals}})^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$

$$= (\rho_D = \Phi \wedge (\forall M: M \in \sigma_D: M = 16) \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow \text{MTEXT} = 16.$$

This formula is valid, so satisfaction is established. Noninterference follows trivially, as before.

Clearly, the trick was to find a suitable invariant. (This is analogous to a standard technique used when attempting to perform a noninterference proof.) Several standard disciplines for using asynchronous message-passing can be viewed in terms of the invariants they preserve, which are particularly simple. One discipline was used above:

Restricting Postconditions of Receive Statements. An invariant describing the values of messages sent can be used to strengthen the proof of a process containing a **receive** r when the postcondition of r is in terms of the value of the message received and not in terms of the state of the sender.

Invariant I_{vals} is such an invariant. Some other useful disciplines are now described.

Sometimes, execution of a receiver must be synchronized with a sending process. Consider what happens if post(r) in D above is changed to assert that $x = 16$ when the message is received. The sequential proof for D becomes

D:: $\{\rho_D = \Phi\}$
  r: **receive** m **when** true
  $\{x = 16\}$
    ⋮
  computation requiring $x = 16$
    ⋮

The satisfaction formula for r, in light of the new postcondition is

$$(\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow (x = 16)^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$$
$$= (\rho_D = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D)) \Rightarrow x = 16.$$

Again, this formula is not valid, so the proof must be strengthened.

Notice that $x = 16$ is true in process A starting from the time the message is sent to D. Therefore, the following is an invariant of A.

$$I_{\text{mono}}\text{: } 16 \notin \sigma_D \vee x = 16.$$

Moreover, I_{mono} is not interfered with by execution of D. Therefore, I_{mono} is also an invariant of D, and together with I_{vals} it can be used to further strengthen assertions in the proof of D. The new satisfaction formula for r is

$$(\rho_D = \Phi \wedge I_{\text{vals}} \wedge I_{\text{mono}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D))$$
$$\Rightarrow (x = 16 \wedge I_{\text{vals}} \wedge I_{\text{mono}})^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}},$$

which is valid.

To complete the proof, noninterference must be shown. Only the term $x = 16$ in post(r) in D might be invalidated by execution of process A. The statements in A that are parallel post(r) are the **send** and the assignment $x := 16$. It suffices to note that execution of neither of these invalidates post(r).

This example illustrates how, in addition to transferring values from sender to receiver, a **receive** facilitates the "transfer of a predicate"—in this case $x = 16$—

from the proof of the sender to the proof of the receiver. Clearly, transfer of a predicate must be done with care so that subsequent execution by the sender does not invalidate it. As shown above, one safe way to transfer a predicate is to make it a monotonic precondition of a **send**. (An assertion is *monotonic* if once it becomes true it remains so.) Above, $x = 16$ is implied by the precondition of the **send** in A and is also monotonic. Therefore, $x = 16$ will be true when the message is received, regardless of delivery delays. In general

Transfer of Monotonic Preconditions of Send Statements. If P is a monotonic precondition of a **send**, then the following is an invariant:

$$I_{\text{mono}}\text{: "message not sent" } \vee \text{ "}P\text{ true"}$$

This invariant can be used to strengthen the proof of a process containing **receive** r, in order to establish satisfaction when P appears in post(r).

The transfer of monotonic predicates is quite common. Sending an acknowledgment for a message m that has been received by D can be viewed as transferring the predicate $m \in \rho_D$ from D to some other process. Once a message is a member of ρ_D it remains so. Thus, $m \in \rho_D$ is a monotonic predicate for any process.

Transfer of nonmonotonic predicates between processes is also possible, but the structure of the program must ensure the truth of the transferred predicate when the message is received. For example, consider the implications of allowing x to be changed after the message has been sent by process A in the program above. The transferred predicate would no longer be monotonic. Thus, A must be prevented from changing x until D has completed any processing requiring the truth of $x = 16$. To facilitate this, A will wait for an acknowledgement from D. The revised sequential proofs are

A:: $\{\sigma_D = \Phi \wedge \rho_A = \Phi\}$
  $x := 16;$
  $\{\sigma_D = \Phi \wedge \rho_A = \Phi \wedge x = 16\}$
  s: **send** x **to** D;
  $\{16 \in \sigma_D \wedge \rho_A = \Phi \wedge x = 16\}$
  r': **receive** ackmsg **when** ackmsg = 'ack';
  {'ack' $\in \rho_A$}
  $x := 17$

D:: $\{\rho_D = \Phi \wedge \sigma_A = \Phi\}$
  r: **receive** m **when** true;
  $\{x = 16 \wedge 16 \in \rho_D \wedge \sigma_A = \Phi\}$
  ⋮
  computation requiring $x = 16$
  ⋮
  $\{x = 16 \wedge 16 \in \rho_D \wedge \sigma_A = \Phi\}$
  s': **send** 'ack' **to** A;
  {'ack' $\in \sigma_A \wedge 16 \in \rho_D$}
  ⋮
  can no longer assume $x = 16$

The satisfaction formulas for r and r' must now be shown valid. First, for r':

$$(16 \in \sigma_D \wedge \rho_A = \Phi \wedge x = 16 \wedge \text{MTEXT} = \text{'ack'} \wedge \text{MTEXT} \in (\sigma_A \ominus \rho_A))$$
$$\Rightarrow (\text{'ack'} \in \rho_A)^{\text{ackmsg},\rho_A}_{\text{MTEXT},\rho_A \oplus \text{MTEXT}}$$

$$= (16 \in \sigma_D \wedge \rho_A = \Phi \wedge x = 16 \wedge \text{MTEXT} = \text{'ack'} \wedge \text{MTEXT} \in (\sigma_A \ominus \rho_A))$$
$$\Rightarrow \text{'ack'} \in (\rho_A \oplus \text{MTEXT}).$$

This is obviously valid. $\text{Sat}_{\text{asynch}}(r)$ is

$$(\rho_D = \Phi \wedge \sigma_A = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D))$$
$$\Rightarrow (x = 16 \wedge 16 \in \rho_D \wedge \sigma_A = \Phi)^{m,\rho_D}_{\text{MTEXT},\rho_D \ominus \text{MTEXT}}$$

$$= (\rho_D = \Phi \wedge \sigma_A = \Phi \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D))$$
$$\Rightarrow (x = 16 \wedge 16 \in (\rho_D \oplus \text{MTEXT})).$$

This is not valid, so we must strengthen the proof of D.

Note that

$$I_A\colon \sigma_D = \Phi \vee x = 16 \vee \text{'ack'} \in \rho_A$$

is an invariant of process A and is not interfered with by execution of process D. Strengthening assertions in the proof of D with $I_A \wedge I_{\text{vals}}$ yields the satisfaction formula

$$(\rho_D = \Phi \wedge \sigma_A = \Phi \wedge I_A \wedge I_{\text{vals}} \wedge \text{MTEXT} \in (\sigma_D \ominus \rho_D))$$
$$\Rightarrow (x = 16 \wedge 16 \in (\rho_D \oplus \text{MTEXT}) \wedge I_A \wedge I_{\text{vals}}).$$

This is valid—the first disjunct of I_A implies the falsity of the antecedent of the satisfaction formula; the truth of the second, along with I_{vals}, implies the consequent; the third implies $\sigma_A \neq \Phi$ (due to the Unreliable Datagram Network Axion), which means that the antecedent of the satisfaction formula is false.

Showing noninterference here involves showing that execution of A does not interfere with assertions in D that mention x. By design, such assertions appear only between r and s'. The assertion

$$I_{\text{cs}}\colon x = 16 \wedge 16 \in \rho_D \wedge \sigma_A = \Phi$$

appears at both the start and end of that code section and is not invalidated by statements in that portion of D, so it can be included as a conjunct in each intermediate assertion. The only statement parallel to these assertions that could invalidate them is $x := 17$ in A. However,

$$\text{pre}(\text{"}x := 17\text{"}) \Rightarrow \text{'ack'} \in \rho_A,$$

and from the Unreliable Datagram Network Axiom we have

$$\text{'ack'} \in \rho_A \Rightarrow \text{'ack'} \in \sigma_A.$$

Since

$$(I_{\text{cs}} \wedge \text{'ack'} \in \sigma_A) = \text{false},$$

interference is not possible.

In this example, satisfaction was established by strengthening assertions with an invariant based on the following protocol:

(1) Process A maintained the truth of the transferred predicate, $x = 16$, until after D finished any processing requiring the truth of $x = 16$, and
(2) because A had no way of knowing exactly when D no longer required the truth of the transferred predicate, D sent a message—an acknowledgment—to communicate that fact.

The general strategy is

Nonmonotonic Predicates. A message can be used to transfer a predicate from process P_s to process P_r as long as P_s ensures that the predicate is true at the time the message is received. This can be done by ensuring that the transferred predicate is true at the time the message is sent and remains true until P_r returns an acknowledgment to P_s. Thus P_s maintains the following invariant:

I_s: "message not sent" ∨ "acknowledgment received"
∨ "transferred predicate true".

In summary, showing satisfaction for each of the protocols above was simplified by using an invariant to strengthen the proof of the receiver. The basis for such an invariant is that at all times either

(1) Any message that could be received has not yet been sent or has already been received. Thus, MTEXT $\in (\sigma_D \ominus \rho_D)$, a conjunct in the antecedent of the satisfaction formula, is false.
(2) A message has been sent but cannot be received. Thus, pre(r) or β^m_{MTEXT} in the antecedent of the satisfaction formula is false.
(3) A message could be received and $\text{post}(r)^{m,\rho_D}_{\text{MTEXT},\rho_D \oplus \text{MTEXT}}$ is true. Thus, the consequent of the satisfaction formula is true.

3.3 Relationship between Satisfaction and Noninterference

Establishing satisfaction for **receive** r is equivalent to proving that no process invalidates post(r). Therefore, one might expect this obligation to be superfluous, arguing that interference with post(r) by statement S should be detected when proving $\text{NI}_{\text{asynch}}(S, \text{post}(r))$ in the noninterference proof. Unfortunately, because messages are buffered, a statement S in one process can interfere with post(r) even if S and r are not concurrent—that is, even if S cannot be executed immediately after r has completed. To see this, consider the following.

```
SELF :: x := 2;
        {x = 2}
        send 'x_is_two' to SELF;
        x := 3;
        {x = 3}
        receive m when true;
        {m = 'x_is_two' ∧ x = 2}
```

Here, process SELF sends a message to itself and then invalidates the transferred predicate ($x = 2$) before executing a **receive**. The sequential proof given above can be derived from our axioms. There is only one process, so noninterference is trivially established. Yet, the postcondition of the **receive** will not be true when

the **receive** terminates. An attempt to establish satisfaction, however, will fail. (It is possible to construct similar pathologies for programs involving more than one process.)

The example above also illustrates a common misconception about the origin of the "miraculous" postcondition of the **receive**. In Section 2.1 and [17], the claim is made that anything can appear as the postcondition of a CSP communications command because such a statement will not terminate when executed in isolation.[9] However, our **receive** can terminate even when there is only a single process, as illustrated above. (Thus, the Unreliable Datagram Receive Axiom taken by itself is not sound; satisfaction and noninterference proofs are required to ensure soundness.) In general, it seems that a miraculous postcondition arises from the synchronization character of a statement. If a statement can cause an arbitrary delay awaiting an event that is in no way caused by execution of that statement, then the axiom for such a statement can have a miraculous postcondition. The phenomenon is not related to message passing at all. For example, consider the **await** statement

await sem > 0 **then** sem := sem − 1.

We are allowed to deduce

$$\text{pre}(\text{“sem := sem} - 1\text{”}) = \text{sem} > 0.$$

This is miraculous because sem > 0 has nothing to do with the precondition of the **await**. The appearance of sem > 0 in the text of the **await** statement makes it easy to guess what the miracle might be, but sem > 0 is, nevertheless, a miracle. The major complication offered by message passing is that the miracle is often an assertion about the state of another process, which cannot be directly tested and therefore does not appear in the text of the **receive**.

4. COMMUNICATION WITH VIRTUAL CIRCUITS

A *virtual circuit* [23] is a communications channel with the following properties.

VC1. Messages are delivered reliably.
VC2. Messages sent on a virtual circuit are delivered in the order sent.

Constructing concurrent programs using virtual circuits for communication and synchronization is considerably easier than for unreliable datagrams. For this reason, many communications networks provide support for virtual circuits.

A virtual circuit *V* can be viewed as a FIFO queue. The **send** statement

send expr **on** *V*

causes the value of expression expr to be inserted at the end of the queue. The **receive** statement

receive *m* **from** *V*

removes the value at the head of the queue and assigns it to *m*.

[9] Apt, Francez, and de Roever [1] justify the "miraculous" postcondition by considering processes with communications commands deleted in isolation.

4.1 Proof Rules

One of our goals in this paper is to show how proof rules for various message-passing primitives can be derived. The proof rules in Section 3 were derived from "first principles"—that is, along the same lines as was done in [17] for synchronous message passing in CSP. In this section, we use a different approach—we derive proof rules from the proof obligations for a CSP program that simulates a virtual circuit. As before, proofs involve three steps: a sequential proof, a satisfaction proof, and a noninterference proof.

It is convenient to *model* a virtual circuit V in the following way. Variable σ_V records the sequence of messages that have been sent on V and variable ρ_V records the sequence of messages that have been received on V. Some useful operations on sequences $s1$ and $s2$ are

$s1 + \text{val}$ is the sequence that results by appending val to the end of $s1$.
$s1 \leq s2$ is true if $s1$ is a prefix of $s2$.
$s1 - s2$ is the sequence obtained by deleting prefix $s2$ from the beginning of $s1$.
This expression is undefined if $s2$ is not a prefix of $s1$.

$\text{hd}(s1)$ is the first element in $s1$. This expression is undefined if $s1$ is empty.
$\text{last}(s1)$ is the last element of $s1$. This expression is undefined if $s1$ is empty.

The state of each virtual circuit V is maintained by a process N_V. In the CSP simulation of V, processes $P_1, P_2, \ldots$ that communicate and synchronize using V will communicate with N_V.[10]

$N_V :: \sigma_V, \rho_V := \Phi, \Phi;$
do
$\quad \square_i\, r_V^i: P_i?\sigma_V \rightarrow \text{skip}$
$\quad \square_i\, s_V^i: (\sigma_V - \rho_V) \neq \Phi;\ P_i!(\text{hd}(\sigma_V - \rho_V), \rho_V + \text{hd}(\sigma_V - \rho_V)) \rightarrow \text{skip}$
od

A **send** statement in process P_i

s: **send** expr **on** V

is modeled by

$$s': N_V!\sigma_V + \text{expr}.$$

Command s' matches only r_V^i; its execution causes a new element to be appended to σ_V.

A **receive** statement

r: **receive** m **on** V

in process P_i is modeled by

$$r': N_V?(m, \rho_V).$$

Command r' matches only s_V^i; its execution assigns to m the value of the oldest message available for receipt and updates ρ_V accordingly.

[10] Note that in N_V, sending the updated ρ_V back to P_i instead of replacing skip in the command with an assignment statement simplifies the derivation of the proof rule that will follow.

Since messages are received in the order sent (VC2) the following invariant should hold in N_V:

$$I_V\colon \rho_V \leq \sigma_V.$$

This results in a sequential proof of N_V:

$N_V ::$ $\sigma_V, \rho_V := \Phi, \Phi; \{I_V\}$
do $\{I_V\}$
$\quad \square_i\, r_V^i\colon P_i?\sigma_V \rightarrow \text{skip}\ \{I_V\}$
$\quad \square_i\, s_V^i\colon (\sigma_V - \rho_V) \neq \Phi;\ P_i!(\text{hd}(\sigma_V - \rho_V), \rho_V + \text{hd}(\sigma_V - \rho_V)) \rightarrow \text{skip}\ \{I_V\}$
od

I_V implies that in the second guarded command $(\sigma_V - \rho_V) \neq \Phi$ is well defined. That, in turn, ensures that $\text{hd}(\sigma_V - \rho_V)$ in the following output command is well defined.

Axioms for **send** and **receive** are derived from the axioms for the CSP commands used to simulate those statements. This is done in such a way that any proof using the axioms for **send** and **receive** can be translated into a proof of the CSP program that simulates the original.

The Virtual Circuit Send Axiom is derived as follows. For s', the CSP simulation of s in P_i, we have that

$$\{W^{\sigma_V}_{\sigma_V+\text{expr}}\}\ N_V!\sigma_V + \text{expr}\ \{W\}$$

is a theorem from the Output Command Axiom. (The choice of pre- and postconditions is based on the operational equivalence of **send** and the assignment $\sigma_V := \sigma_V + \text{expr}$.) To establish satisfaction, we construct $\text{Sat}_{\text{synch}}(r_V^i, s')$,

$$(I_V \wedge W^{\sigma_V}_{\sigma_V+\text{expr}}) \Rightarrow (I_V \wedge W)^{\sigma_V}_{\sigma_V+\text{expr}},$$

which is valid. Thus,

Virtual Circuit Send Axiom. $\{W^{\sigma_V}_{\sigma_V+\text{expr}}\}$ **send** expr **on** V $\{W\}$.

No satisfaction obligation is incurred when using this axiom, because the satisfaction proof for the CSP simulation of **send** can always be derived from it.

For r', the CSP simulation of r in P_i, we have from the Input Command Axiom that

$$\{R\}\ N_V?(m, \rho_V)\ \{Q\}$$

is a theorem. (This choice of pre- and postconditions is made because a **receive** can delay its invoker and therefore should allow anything to appear as its postcondition.) Therefore, define

Virtual Circuit Receive Axiom. $\{R\}$ **receive** m **on** V $\{Q\}$.

To establish satisfaction, construct $\text{Sat}_{\text{synch}}(r', s_V^i)$:

$$(R \wedge (\sigma_V - \rho_V) \neq \Phi \wedge I_V) \Rightarrow (Q \wedge I_V)^{m,\rho_V}_{\text{hd}(\sigma_V-\rho_V),\rho_V+\text{hd}(\sigma_V-\rho_V)}.$$

With the unspecified predicates R and Q this is not necessarily valid. We now simplify $\text{Sat}_{\text{synch}}(r', s_V^i)$ so that it is always possible to derive a satisfaction proof

for the CSP simulation of a program that uses virtual circuits from a proof using our proof rules.

First, delete I_V from the antecedent and assert it as an axiom.

Virtual Circuit Network Axiom. $(\forall V\text{: } V \text{ a virtual circuit: } \rho_V \leq \sigma_V)$

This can be done because I_V is true before and after every communication in the CSP simulation. Second, introduce a fresh variable MTEXT to stand for $\text{hd}(\sigma_V - \rho_V)$. After some logical manipulations, we have the satisfaction formula for virtual circuits:

$$\text{Sat}_{\text{vc}}(r)\text{: } (\text{pre}(r) \wedge (\sigma_V - \rho_V) \neq \Phi \wedge \text{MTEXT} = \text{hd}(\sigma_V - \rho_V)) \Rightarrow Q^{m,\rho_V}_{\text{MTEXT},\rho_V \oplus \text{MTEXT}}$$

As before, since MTEXT is a free variable in $\text{Sat}_{\text{vc}}(r)$, but not a program or auxiliary variable, it is implicitly universally quantified over the formula. A satisfaction proof involves

Virtual Circuit Satisfaction Proof. For every **receive** r, prove $\text{Sat}_{\text{vc}}(r)$ valid.

The final step of the proof of the CSP simulation is establishing noninterference. Process N_V contains communications commands and skip statements only. A skip statement interferes with no assertion and the communications commands match those used in the simulation of **send** and **receive** statements in P_1, $P_2, \ldots$. Consequently, noninterference of statements in N_V with the proofs of other processes will be established if noninterference of statements in $P_1, P_2, \ldots$ is established.

For every statement S' not in N_V, where S' is an assignment, the simulation of a **send** S, or the simulation of a **receive** S and every assertion I parallel to S' $\text{NI}_{\text{synch}}(S', I)$ must be proved. Define

$$\text{NI}_{\text{vc}}(S, I)\text{: } \{I \wedge \text{pre}(S)\}\ S\ \{I\}.$$

First, note that no assignment, simulation of a **receive**, or simulation of a **send** will interfere with the only assertion in the proof of N_V, I_V. No assignment statement in any process mentions variables named in I_V. Communication commands that mention variables named in I_V can only be executed along with communication commands in N_V, which we know preserve I_V.

For S' an assignment, $\text{NI}_{\text{synch}}(S', I)$ follows from a proof of $\text{NI}_{\text{vc}}(S', I)$.

For S' a simulation of a **send** S, $\text{NI}_{\text{synch}}(S', I)$ follows trivially from the Output Command Axiom. However, the corresponding satisfaction formula $\text{NI\_Sat}_{\text{synch}}(r^i_V, S', I)$

$$(I \wedge I_V \wedge \text{pre}(S')) \Rightarrow I^{\sigma_V}_{\sigma_V + \text{expr}}$$

must be valid. Given the Virtual Circuit Network Axiom, this is equivalent to proving

$$\{I \wedge \text{pre}(S)\}\quad S\text{: } \textbf{send } \text{expr } \textbf{on } V\quad \{I\}.$$

which is just $\text{NI}_{\text{vc}}(S, I)$! Thus, in order to establish $\text{NI}_{\text{synch}}(S', I)$, it is sufficient to establish $\text{NI}_{\text{vc}}(S, I)$ for S' a simulation of **send** S.

For S' a simulation of a **receive** S, $\text{NI}_{\text{synch}}(S', I)$ follows trivially from the Input Command Axiom. However, the corresponding satisfaction formula

$\text{NI\_Sat}_{\text{synch}}(S', s_V^i, I)$ must be shown valid:

$$(I \wedge \text{pre}(S') \wedge (\sigma_V - \rho_V) \neq \Phi \wedge I_V) \Rightarrow I^{m,\rho_V}_{\text{hd}(\sigma_V - \rho_V),\rho_V + \text{hd}(\sigma_V - \rho_V)}.$$

As before, this is simplified by introducing a new variable MTEXT and deleting I_V from the antecedent (due to the Virtual Circuit Network Axiom). This results in a satisfaction formula for showing noninterference for a **receive** S:

$$\text{NI\_Sat}_{\text{vc}}(S, I): (I \wedge \text{pre}(S) \wedge (\sigma_V - \rho_V) \neq \Phi \wedge \text{MTEXT} = \text{hd}(\sigma_V - \rho_V)) \Rightarrow I^{m,\rho_V}_{\text{MTEXT},\rho_V + \text{MTEXT}}.$$

Showing the validity of $\text{NI\_Sat}_{\text{vc}}(S, I)$ is sufficient to ensure the validity of $\text{NI\_Sat}_{\text{synch}}(S', s_V^i, I)$, which is required in proving the CSP simulation of virtual circuits. As might be expected, $\text{NI\_Sat}_{\text{vc}}(S, I)$ is the same as $\text{Sat}_{\text{vc}}(S)$ when $I \wedge \text{pre}(S)$ is used for $\text{pre}(S)$ and I is used for Q. Thus, to prove noninterference:

Virtual Circuits Noninterference Proof

For every **send** and assignment S and every assertion I parallel to S, prove $\text{NI}_{\text{vc}}(S, I)$.

For every **receive** S and every assertion I parallel to S, prove $\text{NI\_Sat}_{\text{vc}}(S, I)$ valid.

At this point, it is instructive to compare our proof rules for unreliable datagrams with those for virtual circuits. The Network Axioms and axioms for **send** and **receive** in the two proof systems are almost identical. The significant difference is the interpretation of σ and ρ—in one case they are multisets, in the other case they are sequences. The satisfaction formulas $\text{Sat}_{\text{asynch}}$ and Sat_{vc} are also similar. In fact, it is possible to make the two satisfaction formulas look identical by choosing "true" for β in $\text{Sat}_{\text{asynch}}$ and defining an operation $\in$ on sequences as[11]

$$\text{MTEXT} \in (\sigma_V - \rho_V) \equiv (\sigma_V - \rho_V) \neq \Phi \textbf{ cand } \text{MTEXT} = \text{hd}(\sigma_V - \rho_V).$$

Finally, the noninterference proof obligations in the two proof systems differ only in the use of different satisfaction formulas when defining NI_Sat.

One apparent difference between the proof systems is that σ and ρ are auxiliary variables in one and program variables in the other. Although σ and ρ are program variables in the virtual circuit proof system, they are not directly accessible to the programmer; they are manipulated solely by executing **send** and **receive**. Thus, as in the unreliable datagram proof system, σ and ρ can only appear in proofs, just as if they were auxiliary variables.

The similarity of the two proof systems is no accident. The unreliable datagram proof system could have been derived from the proof obligations for a CSP simulation of unreliable datagrams in much the same way as the virtual circuit proof system was. One way to implement this simulation is to employ a buffer process N_D for each process D. N_D will look very much like N_V; the major difference is the use of a choose function instead of hd. Consequently, a proof system derived from this simulation will be similar to the proof system derived in this section.

[11] A **cand** B is defined as "**if** A **then** B **else** false".

The similarity of the proof rules for the two systems should not be misunderstood. After receiving a message from a virtual circuit, it is possible to conclude that all messages sent prior to that message along that circuit have been received. This is not the case for unreliable datagrams. Thus, in the following program, which uses virtual circuits, it is possible to prove that the final value of m will be 2.

A :: **send** 1 **on** V;
  send 2 **on** V

D :: **receive** m **on** V;
  $\{m = 1\}$
  receive m **on** V
  $\{m = 2\}$

With unreliable datagrams, messages may not be received in the order sent, and then, the most one can prove is that the final value of m will be 1 or 2.

A :: **send** 1 **to** D;
  send 2 **to** D

D :: **receive** m **when** true;
  $\{m = 1 \vee m = 2\}$
  receive m **when** true;
  $\{m = 1 \vee m = 2\}$

The difference in the semantics of the two types of message passing is embedded in the different interpretations of σ and ρ—in particular, $\in$ versus hd for selecting eligible values for MTEXT.

4.2 Example: A Merge Process

The following example is taken from [18].

> A process merge has two input channels in1 and in2 and an output channel out. Merge expects to receive strictly increasing positive integers along in1 and along in2; it outputs on channel out a merged list of these values. A value that appears in both in1 and in2 is not duplicated in the output; the input streams are assumed to be terminated with the distinguished value ∞, which is not output.

A program to implement merge is

merge :: **receive** $v1$ **on** in1;
  receive $v2$ **on** in2;
  do $v1 < v2 \rightarrow$ **send** $v1$ **on** out;
    receive $v1$ **on** in1
  $\square\, v1 > v2 \rightarrow$ **send** $v2$ **on** out;
    receive $v2$ **on** in2
  $\square\, v1 = v2 \neq \infty \rightarrow$ **send** $v1$ **on** out;
    receive $v1$ **on** in1;
    receive $v2$ **on** in2
  od

We would like to prove that merge establishes

$$R\text{: incr}(\sigma_{\text{out}}) \wedge (\forall x\text{: } x \in (\rho_{\text{in1}} \cup \rho_{\text{in2}})\text{: } x = \infty \vee x \in \sigma_{\text{out}}) \wedge \sigma_{\text{in1}} = \rho_{\text{in1}} \wedge \sigma_{\text{in2}} = \rho_{\text{in2}}$$

where incr(σ_{out}) states that the values of sequence σ_{out} are increasing; the second conjunct states that all but the final values received are eventually sent; and the last two conjuncts state that all values sent to merge are processed by it.

We choose as an invariant for the loop:

$$\begin{aligned}\text{Inv: } & v1 = \text{last}(\rho_{in1}) \wedge v2 = \text{last}(\rho_{in2}) \\ & \wedge (\sigma_{out} = \Phi \vee (v1 \geq \text{last}(\sigma_{out}) \wedge v2 \geq \text{last}(\sigma_{out})) \\ & \wedge \text{incr}(\sigma_{out}) \wedge (\forall x: x \in (\rho_{in1} \cup \rho_{in2}): x \in \sigma_{out} \vee x = v1 \vee x = v2).\end{aligned}$$

The loop terminates when $v1 = v2 = \infty$, and so from Inv we have that $\infty = \text{last}(\rho_{in1})$ and $\infty = \text{last}(\rho_{in2})$. No value is sent on in1 or in2 after ∞. Further, from the Virtual Circuit Network Axiom we have $\rho_{in1} \leq \sigma_{in1}$ and $\rho_{in2} \leq \sigma_{in2}$. We conclude that upon termination, $\rho_{in1} = \sigma_{in1}$ and $\rho_{in2} = \sigma_{in2}$. Thus, if Inv is a loop invariant, then when the loop terminates R will be true.

To show that Inv is a loop invariant, a sequential proof is constructed.

```
merge :: {ρin1 = ρin2 = σout = Φ}
        r1: receive v1 on in1;
        {v1 = last(ρin1) ∧ ρin2 = σout = Φ}
        r2: receive v2 on in2;
        {v1 = last(ρin1) ∧ v2 = last(ρin2) ∧ σout = Φ}
        {Inv}
        do v1 < v2 → {Inv ∧ v1 < v2 ∧ σout = s0}
                     send v1 on out;
                     {Inv ∧ v1 < v2 ∧ σout = s0 + v1}
                     r3: receive v1 on in1 {Inv}
         □ v1 > v2 → {Inv ∧ v1 > v2 ∧ σout = s0}
                     send v2 on out;
                     {Inv ∧ v1 > v2 ∧ σout = s0 + v2}
                     r4: receive v2 on in2 {Inv}
         □ v1 = v2 ≠ ∞ → {Inv ∧ v1 = v2 ∧ σout = s0}
                         send v1 on out;
                         {Inv ∧ v1 = v2 ∧ σout = s0 + v1}
                         r5: receive v1 on in1;
                         {Inv ∧ v1 = v2 ∧ σout = s0 + v1}
                         r6: receive v2 on in2 {Inv}
        od
```

The satisfaction formula for each **receive** is now constructed and shown valid. For $r1$, we have

$$\begin{aligned}&(\rho_{in1} = \rho_{in2} = \sigma_{out} = \Phi \wedge (\sigma_{in1} - \rho_{in1}) \neq \Phi \wedge \text{MTEXT} = \text{hd}(\sigma_{in1} - \rho_{in1})) \\ &\quad \Rightarrow (v1 = \text{last}(\rho_{in1}) \wedge \rho_{in2} = \sigma_{out} = \Phi)^{v1, \rho_{in1}}_{\text{MTEXT}, \rho_{in1} + \text{MTEXT}}\end{aligned}$$

which, after performing the designated substitutions is clearly valid. $\text{Sat}_{vc}(r2)$ is similar; we leave the proof of its validity to the interested reader.

The satisfaction formulas for $r3$–$r6$ are almost identical, so only the details of $r3$ are worked out below.

$$\begin{aligned}&(\text{Inv} \wedge v1 < v2 \wedge \sigma_{out} = s0 + v1 \wedge (\sigma_{in1} - \rho_{in1}) \neq \Phi \wedge \text{MTEXT} = \text{hd}(\sigma_{in1} - \rho_{in1})) \\ &\quad \Rightarrow \text{Inv}^{v1, \rho_{in1}}_{\text{MTEXT}, \rho_{in1} + \text{MTEXT}} \\ = &(\text{Inv} \wedge v1 < v2 \wedge \sigma_{out} = s0 + v1 \wedge (\sigma_{in1} - \rho_{in1}) \neq \Phi \wedge \text{MTEXT} = \text{hd}(\sigma_{in1} - \rho_{in1})) \\ &\quad \Rightarrow (\text{MTEXT} = \text{last}(\rho_{in1} + \text{MTEXT}) \wedge v2 = \text{last}(\rho_{in2}) \\ &\quad \wedge (\sigma_{out} = \Phi \vee (\text{MTEXT} \geq \text{last}(\sigma_{out}) \wedge v2 \geq \text{last}(\sigma_{out})) \\ &\quad \wedge \text{incr}(\sigma_{out}) \\ &\quad \wedge (\forall x: x \in (\rho_{in1} + \text{MTEXT} \cup \rho_{in2}): x \in \sigma_{out} \vee x = \text{MTEXT} \vee x = v2)).\end{aligned}$$

The first conjunct MTEXT = last(ρ_{in1} + MTEXT) of the consequent is equivalent to true; the second ($v2$ = last(ρ_{in2})) and fourth (incr(σ_{out})) conjuncts are obviously implied by Inv in the antecedent.

In the third conjunct, the first disjunct is obviously false due to the presence of $\sigma_{out} = s0 + v1$ in the antecedent. However, the second disjunct is not. To see that MTEXT ≥ last(σ_{out}), recall that incr(σ_{in1}) was given in the problem definition. Given that MTEXT = hd($\sigma_{in1} - \rho_{in1}$), from the antecedent, we conclude MTEXT ≥ last(ρ_{in1}). From Inv in the antecedent we have $v1$ = last(ρ_{in1}), so MTEXT ≥ $v1$. Again, from Inv in the antecedent we have $v1$ ≥ last(σ_{out}). Therefore, by transitivity MTEXT ≥ last(σ_{out}). The final part of this conjunct is directly implied by the antecedent.

To see that the fifth conjunct in the consequent is implied by the antecedent, substitute "$s0 + v1$" (from the antecedent) for σ_{out}.

Thus, satisfaction is established. Noninterference follows trivially.

As pointed out in [18], it is possible to interconnect a number of these processes and construct a larger merge network. To prove properties of this larger network using our proof system, one merely needs to equate the names of connected channels.

5. RENDEZVOUS AND REMOTE PROCEDURES

Rendezvous is the basic mechanism for synchronization and communication in Ada[12] [15].[13] It provides a disciplined way for processes to communicate and synchronize and can be easily implemented using message passing.

A *rendezvous* results from execution of a **call** statement by one process and a matching **accept** statement by another. Execution of

```
call server.proc(in_args # out_args)
```

by process client causes it to be delayed until a matching **accept**—an **accept** labeled proc in process server—is executed.

Execution of an **accept** statement

```
proc: accept(in_parms # out_parms)
        body
        end
```

in process server is as follows. Process server is delayed until some process executes a matching **call**; the in_parms are assigned the values of the corresponding in_args; body is executed; and then the out_args are assigned the values of the corresponding out_parms. Thus, execution is equivalent to

```
in_parms := in_args;
body;
out_args := out_parms
```

A *remote procedure* is like a procedure in a sequential programming language, except the procedure body is not necessarily executed by the caller; in particular,

[12] Ada is a registered trademark of the U.S. Department of Defense.

[13] In the following, a simplified version of the rendezvous mechanism in Ada is treated. For example, we do not consider conditional and timed entry calls, task failure and termination, or entry queues.

it may be executed by another process that is synchronized with the caller so that the usual semantics of the procedure call is achieved. Parameter transmission is by value-result since the procedure body may be executed on a processor different from the one on which the caller executes. Mesa contains remote procedures [20].

It is simple to implement remote procedures using **call** and **accept** statements. For a process client to invoke remote procedure server.proc, it executes

call server.proc(in_args # out_args).

Remote procedure server.proc is implemented by a process

```
server :: do true → proc: accept(in_parms # out_parms)
                          body
                          end
          od
```

Therefore, proof rules for **call** and **accept** can be used for reasoning about programs written in terms of remote procedures.

5.1 Proof Rules for Rendezvous

Proof rules for **call** and **accept** can be derived from the proof obligations for an operationally equivalent CSP program.[14] The **call** statement

c: **call** server.proc(in_args # out_args)

is translated into:[15]

```
c': s_call: server!proc(in_args);
    r_call: server?proc(out_args).
```

The **accept** statement

```
proc: accept(in_parms # out_parms)
        body
        end
```

in process server is translated into

```
proc': if □_i r^i_acpt: client_i?proc(in_parms)
            → body; s^i_acpt: client_i!proc(out_parms)
       fi
```

assuming $client_1$, $client_2$, ... are the processes that call server.proc.

Provided procedure names are distinct, the communications commands in c' will match only commands in the process (server) corresponding to the simulation of an **accept** labeled proc, and communications commands in proc′ will match only those corresponding to the simulation of a **call** naming server.proc.

We tentatively choose the following as the axiom for **call**.

---

[14] Basically, the same translation as proposed by [8] is used here.

[15] This translation works only if the meaning of the out_args does not change as a result of executing proc. For example, if $A[i]$ is one of the out_args and i is a shared variable that is changed by proc, then the translation will be wrong.

Call Axiom. $\{P\}$ **call** server.proc(in_args # out_args) $\{R\}$.

On the basis of our CSP simulation of **call**, in order to prove

$$\{P\} \textbf{ call } \text{server.proc(in\_args \# out\_args)} \{R\}$$

it is sufficient to prove

$$\{P\}\ s_{\text{call}};\ r_{\text{call}}\ \{R\} \tag{5.1}$$

and establish satisfaction and noninterference.

The sequential proof of (5.1) follows from

$$\{P\}\ \text{server!proc(in\_args)}\ \{P'\} \tag{5.2}$$

$$\{P'\}\ \text{server?proc(out\_args)}\ \{R\}, \tag{5.3}$$

both of which are derived from the axioms for communications commands. Note that (5.2) and (5.3), hence (5.1), are provable for any choice of P'. This will be helpful later when deriving the obligations for satisfaction.

In order to prove

```
    {Q}
proc: accept(in_parms # out_parms)
        {T}
        body
        {U}
        end
    {W}
```

it suffices to prove

$$\{Q\}\ r^i_{\text{acpt}}\ \{T\}, \tag{5.4}$$

$$\{T\}\ \text{body}\ \{U\}, \tag{5.5}$$

$$\{U\}\ s^i_{\text{acpt}}\ \{W\}, \tag{5.6}$$

for all i and establish satisfaction and noninterference. Since (5.4) and (5.6) follow directly from the axioms for communications statements, this suggests the following inference rule for **accept**.

Accept Rule

$$\frac{\{T\}\ \text{body}\ \{U\}}{\{Q\}\ \textbf{accept}(\text{in\_parms \# out\_parms})\ \text{body}\ \textbf{end}\ \{W\}}.$$

To establish satisfaction for our CSP simulation, $\text{Sat}_{\text{synch}}(r^i_{\text{acpt}}, s_{\text{call}})$

$$(Q \wedge P) \Rightarrow (T \wedge P')^{\text{in\_parms}}_{\text{in\_args}} \tag{5.7}$$

and $\text{Sat}_{\text{synch}}(r_{\text{call}}, s^i_{\text{acpt}})$

$$(P' \wedge U) \Rightarrow (R \wedge W)^{\text{out\_args}}_{\text{out\_parms}} \tag{5.8}$$

are shown to be valid for each **call** c in P_i and matching **accept** proc. These formulas form the basis for the satisfaction obligation for **call** and **accept**.

The appearance of P' in (5.7) and (5.8) is a problem insofar as P' does not appear in the original proof—it is used to characterize an intermediate state of our CSP simulation of **call**. Clearly, our proof rules should be in terms of assertions that are derivable from the original proof. To mitigate this problem, we stipulate that P' be chosen so that

$$P \Rightarrow P'^{\text{in\_parms}}_{\text{in\_args}}. \tag{5.9}$$

(Recall, anything can be used for P'.) This ensures that P' can be derived from assertions in the original proof. Choosing "true" for P' will always satisfy (5.9). However, usually P' will be a predicate describing some aspect of the caller's state that can be asserted both before and after the call. This allows a single proof of the body of an **accept** to be used for all **call** statements.

We adopt the convention that both the precondition P of a **call** and the additional predicate P' used in satisfaction appear before that **call** in a sequential proof outline. A vertical bar (|) separates the two predicates, as in the following:

```
  ...
  {P | P'}
c: call ...
  {R}
  ...
```

This is easily formalized by extending the syntax of the well-formed formulas of programming logic.

Formulas (5.7)–(5.9) are combined to form the satisfaction obligations for rendezvous primitives. Owing to (5.9), (5.7) becomes

$$(Q \wedge P) \Rightarrow T^{\text{in\_parms}}_{\text{in\_args}}.$$

Combining with (5.8) results in the following satisfaction formula for **call** c and **accept** proc.

$\text{Sat}_{\text{rndvs}}(c, \text{proc})$:

$$P \Rightarrow P'^{\text{in\_parms}}_{\text{in\_args}} \wedge (Q \wedge P) \Rightarrow T^{\text{in\_parms}}_{\text{in\_args}} \wedge (P' \wedge U) \Rightarrow (R \wedge W)^{\text{out\_args}}_{\text{out\_parms}}.$$

Thus, in order to establish satisfaction:

Rendezvous Satisfaction Proof. For every **call** c and **accept** proc that match, prove $\text{Sat}_{\text{rndvs}}(c, \text{proc})$ valid.

The final step of the proof is establishing noninterference. Define

$$\text{NI}_{\text{rndvs}}(S, I): \{I \wedge \text{pre}(S)\}\ S\ \{I\}.$$

As before, showing noninterference involves proving $\text{NI}_{\text{synch}}(S, I)$ for every assignment, simulation of a **call**, and simulation of an **accept** and every assertion I parallel to S.[16] The proof of $\text{NI}_{\text{synch}}(S, I)$ for S an assignment statement and any assertion I follows from the proof of $\text{NI}_{\text{rndvs}}(S, I)$. The proof of $\text{NI}_{\text{synch}}(S, I)$

[16] If "$\{P \mid P'\}$" appears in the proof of a process, then both P and P' are considered assertions that are parallel to statements in other processes.

for S a **call** or an **accept** and any assertion I follows immediately from the axioms for the communications commands that simulate S. However, satisfaction must be established for these theorems. Thus, for each matching **call** c and **accept** proc $\text{NI\_Sat}_{\text{synch}}(r^i_{\text{acpt}}, s_{\text{call}}, I)$, which is

$$(I \wedge Q \wedge P) \Rightarrow I^{\text{in\_parms}}_{\text{in\_args}}$$

and $\text{NI\_Sat}_{\text{synch}}(r_{\text{call}}, s^i_{\text{acpt}}, I)$, which is

$$(I \wedge P' \wedge U) \Rightarrow I^{\text{out\_args}}_{\text{out\_parms}}$$

must be shown valid. Combining these, we get

$$\text{NI\_Sat}_{\text{rndvs}}(c, \text{proc}, I)\text{: } (I \wedge Q \wedge P) \Rightarrow I^{\text{in\_parms}}_{\text{in\_args}} \wedge (I \wedge P' \wedge U) \Rightarrow I^{\text{out\_args}}_{\text{in\_args}}.$$

Thus, from the CSP noninterference proof obligations, it is sufficient to do the following in order to establish noninterference.

Rendezvous Noninterference Proof

For every assignment statement S and assertion I parallel to S, prove $\text{NI}_{\text{rndvs}}(S, I)$.

For every **call** S_{call} and matching **accept** S_{acpt} and every assertion I parallel to both S_{call} and S_{acpt}, prove $\text{NI\_Sat}_{\text{rndvs}}(S_{\text{call}}, S_{\text{acpt}}, I)$ valid.

5.2 Example: Producer/Consumer

Consider the following producer/consumer system. A producer process prod transfers the contents of an array $A[1:N]$ to a single-slot buffer.

```
prod :: var A: array 1 .. N of portion;
        i: integer;
     i := 1;
     do i≠N+1 → p: call buffer.dep(A[i]#);
                   i := i+1
     od
```

The consumer process cons obtains these values from the buffer and uses them to fill its array $B[1:N]$.

```
cons :: var B: array 1 .. N of portion;
        j: integer;
     j := 1;
     do j≠N+1 → c: call buffer.ret(#B[j]);
                   j := j+1
     od
```

The single-slot buffer process is defined as follows.

```
buffer :: var buff: portion;
          n: integer;
       n := 0;
       do n≠N+1 → dep: accept(val #)
                       buff := val
                       end;
                  ret: accept(#res)
                       res := buff
                       end;
                  n := n+1
       od
```

Thus, when prod and cons have both finished executing,

$$(\forall k\colon 1 \le k < N+1\colon A[k] = B[k]).$$

Sequential proof outlines for the processes follow. In these proofs, some auxiliary variables are used. Variable *ni* records the number of elements that have been deposited in buff; *nj*, the number of elements that have been retrieved from buff. Boolean variable in_dep is true whenever a process is executing in remote procedure dep; and Boolean variable in_ret is true whenever a process is executing in remote procedure ret. The values of in_dep and in_ret are changed by including them as both value parameters and result arguments of **calls** to their respective procedures. The auxiliary variables are initialized so that

$$ni = nj = 0 \wedge \neg\text{in\_dep} \wedge \neg\text{in\_ret}$$

```
prod :: var A: array 1 .. N of portion;
          i: integer;
  i := 1;
  do i≠N+1 → {i=ni+1 ∧ ¬in_dep | true}
                p: call buffer.dep(A[i], true#in_dep);
                {ni=i ∧ ¬in_dep}
                i := i+1
  od

cons :: var B: array 1 .. N of portion;
          j: integer;
       j := 1;
       do j≠N+1 → {j=nj+1 ∧ (∀k: 1≤k<j: A[k]=B[k]) ∧ ¬in_ret
                     | (∀k: 1≤k<j: A[k]=B[k])}
                   c: call buffer.ret(true#B[j], in_ret);
                   {j=nj ∧ (∀k: 1≤k≤j: A[k]=B[k]) ∧ ¬in_ret}
                   j := j + 1
       od

buffer :: var buffer: portion;
            n: integer;
          n := 0;
          do n≠N+1 →  {n = ni = nj ∧ ¬in_dep ∧ ¬in_ret}
                      dep: accept(val, in_dep#false)
                           {i=ni+1 ∧ val=A[i] ∧ n=ni=nj ∧ in_dep
                                ∧ ¬in_ret}
                           buff := val; ni := ni+1
                           {i=ni ∧ buff=A[i] ∧ n+1=ni=nj+1
                                ∧ in_dep ∧ ¬in_ret}
                           end;
                      {buff=A[ni] ∧ n+1=ni=nj+1 ∧ ¬in_dep ∧ ¬in_ret}
                      ret: accept(in_ret#res, false)
                           {j=nj+1 ∧ buff=A[ni] ∧ n+1=ni=nj+1
                                ∧ ¬in_dep ∧ in_ret}
                           res := buff; nj := nj+1
                           {j=nj ∧ res=buff=A[ni] ∧ n+1=ni=nj
                                ∧ ¬in_dep ∧ in_ret}
                           end;
                      {n+1=ni=nj ∧ ¬in_dep ∧ ¬in_ret}
                      n := n+1
          od
```

To establish satisfaction, $\text{Sat}_{\text{rndvs}}(p, \text{dep})$ and $\text{Sat}_{\text{rndvs}}(c, \text{ret})$ are constructed and shown to be valid. First, $\text{Sat}_{\text{rndvs}}(p, \text{dep})$

$$(i = ni + 1 \wedge \neg\text{in\_dep} \Rightarrow \text{true}^{\text{val, in\_dep}}_{A[i], \text{true}}$$

$$(n = ni = nj \wedge \neg\text{in\_dep} \wedge \neg\text{in\_ret} \wedge i = ni + 1 \wedge \neg\text{in\_dep})$$

$$\Rightarrow (i = ni + 1 \wedge \text{val} = A[i] \wedge n = ni = nj \wedge \text{in\_dep} \wedge \neg\text{in\_ret})^{\text{val, in\_dep}}_{A[i], \text{true}}$$

$$\wedge (\text{true} \wedge i = ni \wedge \text{buff} = A[i] \wedge n + 1 = ni = nj + 1 \wedge \text{in\_dep} \wedge \neg\text{in\_ret})$$

$$\Rightarrow (ni = i \wedge \neg\text{in\_dep} \wedge \text{buff} = A[ni] \wedge n + 1 = ni = nj + 1 \wedge \neg\text{in\_dep}$$

$$\wedge \neg\text{in\_ret})^{\text{in\_dep}}_{\text{false}}.$$

Clearly, this formula is valid. The proof that $\text{Sat}_{\text{rndvs}}(c, \text{ret})$ is also valid is left to the reader.

The final step of the proof is to establish noninterference. First, consider statements in prod. No statement in prod interferes with any assertion in cons because prod changes no variable appearing in an assertion in cons. To see that statements in prod do not interfere with assertions in buffer, note that i is the only variable changed by prod that appears in assertions in buffer. However, $\text{pre}(i := i + 1) \Rightarrow \neg\text{in\_dep}$, and in_dep appears as a conjunct in every assertion I in buffer. Thus, $\text{pre}(\text{NI}_{\text{rndvs}}(S, I)) \Rightarrow \text{in\_dep} \wedge \neg\text{in\_dep}$ is false, and so $\text{NI}_{\text{rndvs}}(S, I)$ is a theorem. For the **call** labeled p, $\text{NI\_Sat}_{\text{rndvs}}(p, \text{dep}, I)$ must also be proved of every assertion I in cons. This follows trivially because val and in_dep do not appear in any assertion in cons.

Similar reasoning shows that no statement in cons interferes with assertions in prod and buffer.

Lastly, we must show that no statement in buffer interferes with assertions in prod or cons. The only variables of concern here are *ni* and *nj*: they are changed in buffer and appear in assertions in the other processes. However, note that pre("$ni := ni + 1$") $\Rightarrow$ in_dep. Since $\neg$in_dep is a conjunct of every assertion in prod that mentions *ni*, interference is not possible. Similarly, pre("$nj := nj + 1$") $\Rightarrow$ in_ret and $\neg$in_ret is a conjunct of every assertion in cons that mentions *nj*. Therefore, interference is not possible here, either.

6. RELATED WORK

Axioms for reasoning about reliable virtual circuits were first proposed in connection with Gypsy [11]. There, **send** and **receive** are characterized in terms of their effects on shared, auxiliary objects called *buffer histories*. The proof rules for **send** and **receive** are derived from the assignment axiom by translating these statements into semantically equivalent assignments to buffer histories. Because in Gypsy program variables manipulated by one process may not appear in assertions in the proof of another, there is no need to perform a noninterference proof. Unfortunately, this also restricts the class of programs that can be proved correct to those in which (only) values are transferred; programs in which predicates are transferred by message passing cannot be proved. Nevertheless, Gypsy has been successfully used to verify several large concurrent systems.

Apt et al. define a proof system for reasoning about CSP programs [1]. In it, each process is proved in isolation with its communications commands deleted,

and then a *cooperation proof* is performed to show that these proofs can be combined. The axioms for communications commands allow anything to be asserted as the postcondition of an input or output command, and the cooperation proof establishes that assumptions made in the postcondition of a communications command will, in fact, be true whenever execution of that command completes. Performing a cooperation proof involves constructing a global invariant that characterizes pairs of syntactically matching communications commands that can actually exchange a message during some computation (called semantic matching). The global invariant is in terms of program and auxiliary variables, but auxiliary variables may not be shared (in contrast to [17]). Consequently, there is no need to construct a noninterference proof.

Proof systems based on the idea of a cooperation test have been defined for a variety of other programming notations, as well. A proof system for Distributed Processes [3] is presented in [10]. In [8], a proof system for a subset of Ada containing rendezvous is defined and proved sound and complete; in [9] and [2] larger subsets of the concurrency features of Ada are axiomatized.

Proof systems based on cooperation ([1, 2, 8, 9, and 10]) differ from proof systems based on satisfaction ([17] and the work in this paper) in two important regards. First, advocates of the cooperation approach do not allow shared auxiliary variables, arguing that they are inconsistent with the philosophy of a distributed system, where there is no shared storage. By prohibiting shared variables of any kind, noninterference need not be checked. Proponents of the satisfaction approach argue that shared auxiliary variables are convenient and that the noninterference obligation that arises from the use of shared variables is manageable if these shared variables are used in a disciplined manner. Moreover, there appear to be some language features that are most conveniently axiomatized by using shared variables. For example, Gerth and de Roever [9] suggest that Ada entry queues be modeled by shared variables, and thus to handle this feature using their proof system, some form of noninterference test would have to be added.

Secondly, in the cooperation approach a global invariant is furnished by the program prover as part of the cooperation proof. In order to construct the invariant, auxiliary variables may be added to each process. Assignments to these variables are then grouped with communications commands to form bracketed sections—the global invariant need not hold in bracketed sections. In contrast, in the satisfaction approach a mechanical procedure—the satisfaction formula—is given for constructing the global invariant. The programmer need not be concerned with defining bracketed sections. (However, to prove a satisfaction formula valid, the sequential proof might have to be strengthened by adding auxiliary variables.)

In fact, there is really a simple principle behind both the satisfaction and cooperation approaches: proving the invariance of an assertion [5, 16]. The approaches differ only in how the invariant is constructed and how it is partitioned. The differences are, nevertheless, significant—they affect how one thinks about a program when constructing its proof.

Another approach to designing a proof system for distributed programs is based on the use of traces [14, 18, 19, 22]. In this approach, processes and networks of processes are described in terms of their input/output histories, called *traces*. A

program proof involves two steps. First, the trace of each process is characterized by using a programming logic in which input and output commands are modeled as assignments to traces. Then, proofs of processes are combined by using inference rules that relate these traces. Once the input/output history of a process or network is determined, reasoning can proceed in terms of assertions on traces, instead of assertions on program variables. This approach is particularly interesting because it allows one to reason about the result of combining a collection of components—processes or networks—without concern for their implementations.

7. CONCLUSION

Proof rules for asynchronous message-passing with unreliable datagrams, asynchronous message-passing with reliable virtual circuits, remote procedures, and rendezvous have been presented. The proof systems obtained are based on extending and applying the notion of a satisfaction proof. We have shown two approaches to obtaining the satisfaction obligations for a set of message-passing primitives: the obligations can be derived directly, or they can be derived from a CSP program that simulates the operation of the primitives.

The message-passing primitives we have considered are intended to be representative of the major approaches to synchronization and communication in distributed programs. As such, we have described and analyzed general constructs with the hope that existing mechanisms could then be modeled as instances of these representatives. For example, the syntax of **send** for unreliable datagrams requires that a destination process be explicitly named. However, the approach can be generalized to handle the case where a communications channel is named as a destination. It is also possible to handle the case where the destination in a **send** is computed at runtime using the technique described in [7]. Similarly, the inclusion of the client name in a remote procedure and of a server name in a remote procedure call was merely to allow mechanical construction of satisfaction formulas when these constructs are used. It is a simple matter to generalize our proof rules to the case where the client and server are not explicitly named (as in Ada) [8].

ACKNOWLEDGMENTS

Discussions with Alan Demers, David Gries, Leslie Lamport, and Gary Levin have been invaluable in formulating the ideas in this paper. We are also indebted to Bowen Alpern, Greg Andrews, Ozalp Babaoglu, Rob Gerth, Rob McCurley, W. P. de Roever, and an anonymous referee for helpful comments on an earlier draft of this paper. Gil Neiger assisted with the example in Section 5.2.

REFERENCES

1. APT, K., FRANCEZ, N., AND DE ROEVER, W. A proof system for communicating sequential processes. *ACM Trans. Prog. Lang. Syst. 2*, 3 (July 1980), 359–385.
2. BARRINGER, H., AND MEARNS, I. Axioms and proof rules for Ada tasks. *IEE Proc. 129, Part E*, 2 (Mar. 1982), 38–48.
3. BRINCH HANSEN, P. Distributed processes: A concurrent programming concept. *Commun. ACM 21*, 11 (Nov. 1978), 934–941.

4. CLINT, M. Program proving: Coroutines. *Acta Inf. 2*, 1 (1973), 50–63.
5. COUSOT, P., AND COUSOT, R. Semantic analysis of communicating sequential processes. In *Proceedings of ICALP80.* Lecture Notes in Computer Science, vol. 85. Springer-Verlag, New York, 1980, pp. 119–133.
6. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.
7. FRANCEZ, N. Extended naming conventions for communicating processes. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 25–27). ACM, New York, 1982, pp. 40–45.
8. GERTH, R. A sound and complete Hoare axiomatization of the Ada-rendezvous. In *Proceedings of ICALP82,* Lecture Notes in Computer Science, vol. 140. Springer-Verlag, New York, 1982, pp. 252–264.
9. GERTH, R., AND DE ROEVER, W.P. A proof system for concurrent Ada programs. Tech. Rep. RUU-CS-83-2, Vakgroep Informatica, Rijksuniversiteit Utrecht, The Netherlands, Jan. 1983.
10. GERTH, R., DE ROEVER, W.P., AND RONCKEN, M. Procedures and concurrency: A study in proof. In *Proceedings of ISOP'82,* Lecture Notes in Computer Science, vol. 137. Springer-Verlag, New York, 1982, pp. 132–163.
11. GOOD, D., COHEN, R., AND KEETON-WILLIAMS, J. Principles of proving concurrent programs in Gypsy. In *Proceedings of the 6th Annual Symposium on Principles of Programming Languages* (San Antonio, Tex., Jan. 29–31). ACM, New York, 1979, pp. 42–52.
12. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12,* 10 (Oct. 1969), 576–580.
13. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21,* 8 (Aug. 1978), 666–677.
14. HOARE, C.A.R. A calculus of total correctness for communicating processes. *Sci. Comput. Prog. 1,* (1981), 49–72.
15. ICHBIAH, J.D., KRIEG-BRUECKNER, B., WICHMANN, B.A., LEDGARD, H.F., HELIARD, J.-C., ABRIAL, J.-R., BARNES, J.G.P., AND ROUBINE, O. Preliminary ADA Reference Manual. *SIGPLAN Not. 14,* 6 part A (June 1979).
16. LAMPORT, L. AND SCHNEIDER, F.B. The "Hoare Logic" of CSP, and all that. *ACM Trans. Prog. Lang. Syst. 6,* (Apr. 1984), 281–296.
17. LEVIN, G., AND GRIES, D. Proof techniques for communicating sequential processes. *Acta. Inf. 15* (1981), 281–302.
18. MISRA, J., AND CHANDY, K.M. Proofs of networks of processes. *IEEE Trans. Softw. Eng. SE-7,* 4 (July 1981), 417–426.
19. MISRA, J., CHANDY, K.M., AND SMITH, T. Proving safety and liveness of communicating processes with examples. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug. 18–20). ACM, New York, 1982, pp. 201–208.
20. MITCHELL, J.G., MAYBURY, W., AND SWEET, R. Mesa language manual (Version 5.0). Tech. Rep. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif., 1979.
21. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs I. *Acta Inf. 6* (1976), 319–340.
22. SOUNDARARAJAN, N. Axiomatic semantics of communicating sequential processes. Tech. Rep., Dept. of Computer and Information Science, Ohio State Univ., Columbus, 1981.
23. TANENBAUM, A. *Computer Networks.* Prentice Hall, Englewood Cliffs, N.J., 1981.

Received June 1982; revised June and December 1983; accepted February 1984