Acta Informatica 15, 281 – 302 (1981)

Acta Informatica
© Springer-Verlag 1981

A Proof Technique for Communicating Sequential Processes

Gary Marc Levin[1] and David Gries[2]

[1] University of Arizona, Dept. of Computer Science, Tucson, Arizona 85721, USA
[2] Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, New York 14853, USA

Summary. Proof rules are presented for an extension of Hoare's Communicating Sequential Processes. The rules deal with total correctness; all programs terminate in the absence of deadlock. The commands *send* and *receive* are treated symmetrically, simplifying the rules and allowing *send* to appear in guards. Also given are sufficient conditions for showing that a program is deadlock-free. An extended example illustrates the use of the technique.

1. Introduction

Two common models for parallel computation are: *centralized*, where all processes share (have access to) all variables; and *distributed*, where the variables of each process are private, i.e. not accessible to other processes, and message passing is used to provide interaction. The centralized model has been well studied. Here we consider the distributed model, which corresponds to a system of processors, each with its own memory, and a communication network through which messages may be sent between processors.

To properly study algorithms, one needs a notation for their description. In this paper we use Hoare's CSP (Communicating Sequential Processes) [10], which is intended for this purpose. It is derived from Dijkstra's simple programming notation [6].

The set of simple commands (assignment and *skip*) has been augmented by two communication commands, *send* and *receive*, which serve to pass information between processes. Message passing is synchronous and typed – the sender waits until the message has been received and a process specifies the type of message to be received.

In addition to sequential composition ($S_1; S_2$), there is parallel composition, which syntactically specifies the commands (processes) to be executed concurrently and associates a name with each process.

The composite commands alternation and repetition have been extended to allow a *send* or *receive* to appear in each guard. The selection of a guard that includes a communication command is based on the readiness of another process to communicate.

The approach presented in this paper separates the *sequential proofs* of processes, which may need assumptions about the effects of communication, from the *satisfaction proofs*, which show these assumptions to be valid. Although in this paper we only prove the correctness of systems of processes, isolated processes can also be handled. Satisfaction proofs place constraints on other processes, much as assumptions about the parameters of a procedure place constraints on how the procedure may be invoked.

Section 2 contains an operational description of CSP, Sect. 3 proof rules for correctness in the absence of deadlock, and Sect. 4 sufficient conditions for proving the absence of deadlock. Section 5 illustrates the proof method with a program and its proof of correctness.

This paper represents part of the Ph.D. thesis of the first author, and publication of this material was delayed until the thesis was completed. Similar, independent work by Krzysztof Apt, Nissim Francez and Willem de Roever, done roughly at the same time, has already appeared in the literature [1]. References [3] and [5] also deal with proving communicating sequential processes correct. The differences in our approach and [1] are discussed in Sect. 6, and the reader who has read [1] may want to turn to Sect. 6 before reading this paper.

2. The Notation CSP

In this section we present an informal, operational semantics from which the reader can get an intuitive grasp of the notation.

In dealing with concurrency we will have to deal with problems of synchronization, and in doing so we will use the terms ready, blocked, and terminated. Execution of a *ready* process can continue, execution of a *blocked* process is being delayed for a communication, and execution of a *terminated* process is finished. In the description below, the terms ready and blocked are explained more precisely.

2.1. Simple Commands

skip:	*skip*
assignment:	$x_1, \dots, x_n := e_1, \dots, e_n$ or $\bar{x} := \bar{e}$
send:	$A!\,T(\bar{x})$
receive:	$A?\,T(\bar{x})$

The first two simple commands, from Dijkstra's sequential notation, place no restriction on execution; a process whose next command is *skip* or assignment is ready and can be executed. Execution of *skip* does nothing. To execute an assignment: determine the references represented by $x_1, \ldots, x_n$ and the values of $e_1, \ldots, e_n$, then store the values in the locations of the corresponding references, in left-to-right order. The notation $\bar{x}$ represents the vector $x_1, \ldots, x_n$.

The communication commands *send* and *receive* have three parts: a process name A, a template (which for simplicity is restricted to an identifier) T and a parameter list (a list of variables or expressions) $\bar{x}$. A *send* command $A!T(\bar{e})$ and a *receive* command $B?T(\bar{x})$ form a *matching pair* if and only if $A!T(\bar{e})$ appears in process B, $B?T(\bar{x})$ appears in process A, they both have the template T, and $\bar{x}:=\bar{e}$ is a syntactically legal multiple assignment. The template is a means of distinguishing different kinds of messages that are passed between a pair of processes. If there is no need for this distinction, templates can be omitted. If the parameter list has length one and there is no template, the parentheses are omitted. If there is no information to be passed (only the type of message is important), the parameter list and parentheses are omitted.

For the rest of the paper, in order to simplify the discussion, we will use the notation $A.T(\bar{x})$ to refer to a *send* $A!T(\bar{x})$ or a *receive* $A?T(\bar{x})$ whenever it is immaterial which it is. The notation never appears in program text; it is used only to reduce repetition in definitions. The notation $pair(r, s)$ stands for "r and s are a matching pair".

A matching pair is executed as follows. Suppose execution of process B is at a command $A.T(\bar{x})$. If process A is *not* at a matching command $B.T(\bar{y})$, then B is *blocked* and *prepared to communicate.* On the other hand, if A is at a matching command $B.T(\bar{y})$, then both processes are *ready* and execution may proceed. Execution consists of executing the multiple assignment statement $\bar{x}:=\bar{y}$ (or ($\bar{y}:=\bar{x}$, depending on which command is the *send*). When such execution occurs, the processes are said to have *synchronized* at the matching pair.

2.2. Composite Commands

sequence:	$S_1; \ldots; S_n$
parallel:	$[A_1::S_1 \,\|\, \ldots \,\|\, A_n::S_n]$
alternation:	**if** $b_1; c_1 \rightarrow S_1 \,\Box\, \ldots \,\Box\, b_n; c_n \rightarrow S_n$ **fi**
repetition:	**do** $b_1; c_1 \rightarrow S_1 \,\Box\, \ldots \,\Box\, b_n; c_n \rightarrow S_n$ **od**

Composition provides the means of forming commands from simpler ones. Sequence is the most familiar form of composition. (In fact, it is frequently called composition; we have renamed it to avoid confusion.) To execute $S_1; \ldots; S_n$, execute S_1, then $S_2, \ldots$, and finally S_n.

Execution of a parallel command consists of executing the component commands $S_1, \ldots, S_n$ concurrently; the command is completely executed when all component commands are completely executed. Each command S_i is called a process. For purposes of communication, the processes must be named; here, S_i is named A_i.

Alternation and repetition are formed from sets of *guarded commands.* A guarded command b; $c \rightarrow S$ consists of a *guard* b; c and a *command* S. In the guard, b is a boolean expression and c is either *skip* or $A.T(\bar{x})$.

We now explain when a guard is failed, ready or blocked. If b is *false*, the guard is *failed*. If b is *true* and $c = skip$, the guard is *ready*. If b is *true* and c is $A.T(\bar{x})$, the guard is *prepared to communicate* with process A; it is *ready* when A is prepared to communicate and *blocked* at other times.

Execution of an alternative command selects a guarded command with a ready guard and executes the sequence "$c; S$". If c is *skip*, execution is independent of other processes; if c is a communication command, then a matching communication command must be executed simultaneously. When some guards are blocked and none are ready, the process is blocked and must wait. If all guards are failed, the process aborts.

Execution of the repetitive command is the same except that, whereas execution of alternation selects one guarded command and is completed, for repetition the selection is repeated until all guards are failed, at which time execution of the repetition is completed.

The boolean *true* or the command *skip* may be omitted from guards, in which case the separating semicolon is omitted as well.

2.3. *Scheduling*

In sequential notations, there is the tacit assumption that a program is executed at some finite speed until it reaches the end or an error. This allows the conclusion that a program that can terminate will in fact do so. In CSP, where there can be more than one process, choices must sometimes be made as to which processes are to proceed. This is a question of scheduling. Our assumptions below are based on our interest in correctness in the absence of deadlock. Other interests may require other assumptions.

A system is *deadlocked* if some process is not terminated and no process is ready. Our scheduling assumption is: if any process is ready, progress must be made in a finite, bounded amount of time. This says that if the system is not deadlocked, something will happen.

We make no assumptions about fairness; a ready process can be delayed arbitrarily long before it is selected to make progress. A proper study of fairness would have required the notion of time, and we have preferred to concentrate on the simpler aspects of the problem instead. Fairness is often required in order to avoid the possibility of individual starvation – a dynamic form of deadlock in which one process is forever blocked while others progress. We have finessed this problem by requiring all processes to terminate, if not deadlocked, so that no process can continue indefinitely. Hence, eventually, all processes are either blocked or terminated, so that the only form of starvation is the simple, static deadlock.

2.4. *Parameterization*

When many similar processes are desired we allow the abbreviation

$$[\|_{i \in N} A_i :: S_i]$$

to represent

$$[A_{n_1}::S_{n_1} \| \dots \| A_{n_k}::S_{n_k}]$$

where $N=\{n_1, \dots, n_k\}$. In the case of repetition and alternation,

$$\Box_{i \in N} b_i;\ c_i \rightarrow S_i$$

is an abbreviation for

$$b_{n_1}; c_{n_1} \rightarrow S_{n_1} \Box \dots \Box b_{n_k}; c_{n_k} \rightarrow S_{n_k}.$$

2.5. An Example – Bounded Buffer

At this point, a simple, well-known example will allow readers to check their understanding. Because CSP assumes synchronous message passing, a common complaint about it is that processes are too tightly coupled. But it is simple to insert a buffer process between processes, as we now show.

Assume that two processes I and O wish to communicate through a buffer B. Process I sends message e to O using "$B!e$" and O receives a message in x using "$B?x$". Process B is shown in Fig. 1 for a buffer of size $N>0$.

```
Declare $b(0{:}N-1)$, $out$, $count$, and $done$.
    1) Messages are, in order, in $b(i \bmod N)$, $out \leqq i < out + count$.
    2) $0 \leqq count \leqq N$.
    3) "$\oplus$" is used to denote addition modulo $N$.

$out, count, done := 0, 0, false;$
do $\neg done \wedge count < N$; $I?b(out \oplus count) \rightarrow count := count + 1$
$\Box$            $count > 0$;  $O!b(out)$       $\rightarrow out, count := out \oplus 1,\ count - 1$
$\Box$ $\neg done$;            $I?quit$         $\rightarrow done := true$
od;
$O!quit$
```

Fig. 1. A bounded buffer

Process B can receive two kinds of messages: an integer, which is a parameter without a template; and *quit*, which is a template without a parameter. B can receive if the buffer is not full ($count < N$) and process I has not indicated that no more messages are to be buffered ($\neg$done) and can send if the buffer is not empty ($count > 0$). If the buffer is neither full nor empty, B can either send or receive; what it does next depends on the speeds of I and O and on the scheduler. B can also receive *quit* from I; having received *quit*, it can no longer receive from I. When the repetitive command terminates, B sends *quit* to O and terminates.

2.6. Changes from Hoare's CSP

There are two main differences between our notation and Hoare's [10]: the inclusion of *send* in guards and the removal of distributed termination.

Hoare allowed only *receive* in guards, basing his decision on efficiency concerns. At least in some environments [15], there is no additional overhead in providing conditional output and others have suggested the inclusion of output guards [2, 16].

The program in Sect. 2.5 is an example of one advantage of this addition. Without output guards, O would have to send a request for input to B. This would entail extra messages, place an unnecessary burden on the programmer, and change the form of process O. As is, O looks the same regardless of whether it is receiving input synchronously or from a buffer.

Distributed termination provides the means for automatic termination of a loop in one process because another process has terminated. We have not included it because it seems to complicate the semantics. Furthermore, algorithms that use it can be difficult to modify. In particular, if the modification requires that the previously terminating process continue, there is no way to simply terminate the loop; new signals and, possibly, restructuring of the algorithm will be needed. Instead, we assume that all termination of loops is done explicitly.[1]

An earlier version of this paper [13] included distributed termination, and the complications in an example (given later in this paper) caused us to omit it from consideration.

3. Proof Rules for Weak Correctness

A proof of *weak correctness* (total correctness in the absence of deadlock) of a set of communicating processes consists of three parts: a *sequential* proof, a *satisfaction* proof, and a *non-interference* proof.

The sequential proof for each process is in the style of a Hoare-logic proof [9].

While creating the sequential proof, assumptions are made about the effect of communication commands. A satisfaction proof shows that these assumptions are valid.

Although CSP requires that each variable be local to a process, *auxiliary variables*, which are allowed to appear in more than one process, are usually needed in a proof of correctness. This does not violate the distributed model because auxiliary variables are needed only for the proof and not for the execution. However, it now becomes necessary to show that execution of other processes cannot interfere with the validity of assertions, as described in [14]. The notions of *synchronously altered variables* and *universal assertions* are introduced to simplify proof of non-interference; in fact, with certain restrictions, non-interference becomes a syntactic property of the program and its sequential and satisfaction proofs.

3.1. Axioms and Rules of Inference

The notation $\{P\}\,S\,\{Q\}$ means that execution of S begun in a state satisfying P is guaranteed to terminate in a state satisfying Q, provided deadlock does not

[1] The message *quit* is used in the bounded buffer example for this purpose

occur. The notation $P^{\bar{x}}_{\bar{e}}$ denotes conventional textual substitution: the predicate P with all free occurrences of the x_i simultaneously replaced by the e_i, where $\bar{x}$ is a list of distinct identifiers.

Axioms

skip:	$\{P\}\ skip\ \{P\}$
assignment:	$\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$
communication:	$\{P\}\ A.T(\bar{x})\ \{Q\}$

The axioms for *skip* and assignment are conventional. The notation $P^{\bar{x}}_{\bar{e}}$ is usually used only when $\bar{x}$ is a list of distinct identifiers, but in [8] it is extended to allow the x_i to be array elements and records so that the assignment axiom given above holds in general. For the purposes of this article one may use the former view and restrict multiple assignment without loss of understanding.

The communication axiom is explained as follows. Remember that $\{P\}\,S\,\{Q\}$ means total correctness in the absence of deadlock and that sequential proofs only prove facts about processes running in isolation. With only one process running, communication commands deadlock; thus, any predicate Q may be assumed to be true upon termination of a communication command because termination never occurs!

The communication axiom does violate the Law of the Excluded Miracle [6], allowing proofs like

$$\{true\}\ A?\,T(\bar{x})\ \{false\} \tag{3.1.1}$$

which implies that, after execution, *false* is *true*. Such proofs are what the Law of the Excluded Miracle was designed to avoid. A satisfaction proof will plug the hole later on.

Inference rules

sequence
$$\frac{\{P\}\,S_1\,\{Q\},\ \{Q\}\,S_2\,\{R\}}{\{P\}\,S_1;\,S_2\,\{R\}}$$

consequence
$$\frac{P \Rightarrow P',\ \{P'\}\,S\,\{Q'\},\ Q' \Rightarrow Q}{\{P\}\,S\,\{Q\}}$$

alternation
$$\frac{\begin{array}{l} P \Rightarrow (\exists i\colon b_i) \\ (\forall i\colon \{P \wedge b_i\}\,c_i;\,S_i\,\{Q\} \end{array}}{\{P\}\ \mathbf{if}\ \Box_{i=1:n}\, b_i;\, c_i \rightarrow S_i\ \mathbf{fi}\ \{Q\}}$$

repetition
$$\frac{\begin{array}{l} (P \wedge (\exists i\colon b_i)) \Rightarrow t > 0 \\ (\forall i\colon \{P \wedge b_i\}\ T := t;\, c_i;\, S_i\,\{P \wedge t < T\}) \end{array}}{\{P\}\ \mathbf{do}\ \Box_{i=1:n}\, b_i;\, c_i \rightarrow S_i\ \mathbf{od}\ \{P \wedge (\forall i\colon \neg b_i)\}}$$
where t is an integer function and T a fresh variable.

parallel
$$\frac{(\forall i\colon \{P_i\}\,S_i\,\{Q_i\})\ \textit{satisfied and interference-free}}{\{(\forall i\colon P_i)\}\ [\,\|_{i=1:n} A_i\colon\colon S_i]\ \{(\forall i\colon Q_i)\}}$$

The inference rules for sequence and consequence are common and will not be discussed here. The rules for alternation and repetition are also familiar, except that the hypothesis corresponding to a guarded command now includes execution of a command *in the guard*.[2] The careful reader will note that the hypotheses for

$$\mathbf{if}\, b_1;\, c_1 \rightarrow S_1 \,\square \ldots \square\, b_n;\, c_n \rightarrow S_n\, \mathbf{fi} \tag{3.1.2}$$

are the same as the hypotheses for

$$\mathbf{if}\, b_1 \rightarrow c_1;\, S_1 \,\square \ldots \square\, b_n \rightarrow c_n;\, S_n\, \mathbf{fi}. \tag{3.1.3}$$

How then do (3.1.2) and (3.1.3) differ? Actually, in the absence of deadlock there is no difference. In the case of (3.1.3), if a correct alternative is chosen, command c will be executed and all is well (i.e. as it would be if this alternative were chosen in (3.1.2)); at worst, c cannot be executed and the system deadlocks. Obviously the difference is that (3.1.3) is more prone to deadlock. This difference is analyzed more formally in [12].

It is clear that the rule for the parallel command implies that each component is executed, as long as the free variables of P_i and Q_i are limited to the local variables of A_i. The necessary introduction of shared auxiliary variables brings the need for non-interference proofs, so that the conventional model of execution and the inference rule are still consistent. For a similar rule, see [14].

Technically, a sequential proof consists of a list of statements (either Hoare triples or statements of the predicate calculus), each of which is either an instance of an axiom or the conclusion of an inference rule with all the hypotheses of that rule preceding it in the list. This form is awkward, and instead one usually gives an *annotated program*. In an annotated program, assertions are placed before and after the program (corresponding to the input-output specifications) and between commands of the program. An assertion must imply the precondition of the following command and, in turn, be implied by the postcondition of the preceding command. This convention allows us to include only one of two assertions where the rule of consequence is applied.

The functions *pre* and *post* are applied to commands and have as value, respectively, the assertion preceding and succeeding the command in the annotated program. At times we will only include the invariant of a loop once; the reader should consider the invariant to be the postcondition of each alternative. Similarly, the postcondition of an alternative command is the postcondition of each of the alternatives and is not repeated.

The precondition of a command that appears in a guard is the conjunction of the precondition of the alternation (or the invariant of the repetition) and the boolean part of the guard; the postcondition of the guard is the precondition of the command that is guarded.

[2] The proofs of the hypotheses may include instances of the axiom for communication commands, necessitating a satisfaction proof

As a brief example consider Fig. 2. Figure 2a is the annotated program, Fig. 2 is the list of Hoare triples and implications that corresponds to the annotated program.

$$\begin{array}{ll}
\{x=a \wedge y=b\} & \{x=a \wedge y=b\}\ t:=x\ \{t=a \wedge y=b\} \\
t:=x; & \\
\{t=a \wedge y=b\} & \{t=a \wedge y=b\}\ x:=y\ \{t=a \wedge x=b\} \\
x:=y; & \\
\{t=a \wedge x=b\} & \{t=a \wedge x=b\}\ y:=t\ \{y=a \wedge x=b\} \\
y:=t & \\
\{x=b \wedge y=a\} & (y=a \wedge x=b) \Rightarrow (x=b \wedge y=a) \\
 & \\
 & \{x=a \wedge y=b\}\ t:=x;\ x:=y;\ y:=t\ \{x=b \wedge y=a\} \\
\text{(a)} & \text{(b)}
\end{array}$$

Fig. 2. An annotated program and corresponding proof

3.2. Satisfaction Proof

Consider any communication command S and its pre- and postconditions from a sequential proof. The communication axiom allows any postcondition because in isolation deadlock is inevitable. Now we are concerned with combining processes. When processes are executed concurrently, deadlock is not inevitable and we must show that the assertions are still satisfied.

Suppose, then, that a matching communication pair appears in processes A and B, as follows:

$$[B :: \ldots \{P\}\, A?T(\bar{x})\, \{Q\} \ldots \| A :: \ldots \{R\}\, B!T(\bar{e})\, \{S\} \ldots]. \tag{3.2.1}$$

Should these two commands communicate, the effect would be equivalent to $\bar{x}:=\bar{e}$. Hence, $(Q \wedge S)$ is true after communication if and only if $(Q \wedge S)^{\bar{x}}_{\bar{e}}$ is true before. Before communication, both preconditions are true and we may assert $(P \wedge R)$. Therefore, postconditions Q and S are satisfied if and only if

$$(P \wedge R) \Rightarrow (Q \wedge S)^{\bar{x}}_{\bar{e}}. \tag{3.2.2}$$

The *Rule of Satisfaction* is that every matching pair of the form (3.2.1) must satisfy (3.2.2).

This fills the gap left by our violation of the Law of the Excluded Miracle. Consider, for example, the use of the communication axiom that caused concern, (3.1.1). A matching *send* would have the form

$$\{R\}\, B!T(\bar{e})\, \{S\} \tag{3.2.3}$$

and we would be obliged to prove

$$(\mathit{true} \wedge R) \Rightarrow (\mathit{false} \wedge S)^{\bar{x}}_{\bar{e}} \qquad (\mathit{or}\ \neg R).$$

If we can prove $\neg R$, then $B!T(\bar{e})$ can never be prepared to communicate, and the matching pair will never be executed. (In general, if $\neg(P \wedge R)$ can be proved, then (3.2.2) is trivially satisfied, regardless of postconditions. This covers the cases where the logic of the program prevents the pair from communicating.)

3.3. Auxiliary Variables

The addition of auxiliary variables to the proof system allows assertions in distinct processes to refer to non-disjoint state spaces. Auxiliary variables are needed to relate program variables of one process to program variables of another. Auxiliary variables are defined by Owicki [14] for use in proofs in the centralized model; our definition is adapted to the distributed model.

An auxiliary variable may affect neither the flow of control nor the value of any non-auxiliary variable. Hence, auxiliary variables are not necessary to the computation and may be omitted from the program – but not the proof. These conditions are ensured if the following syntactic restrictions are met.

Auxiliary variables may appear only

1) in assertions;
2) in expressions being assigned to auxiliary variables;
3) as parameters in a *receive*; and
4) in expressions as parameters of a *send* corresponding to auxiliary variables in any *receive* that forms a matching pair.

When a program is augmented with auxiliary variables, one can add assignments and extend the parameters of communication commands. Adding parameters to communication commands must not change the set of matching pairs, because a change could affect flow of control, which is not allowed.[3]

Auxiliary variables can help describe global relations; local variables, only approximations to them. Commonly, preconditions for communication commands will assert that a local variable equals a global auxiliary variable. When communication occurs, each process involved will have a local variable equal to the global auxiliary variable, and hence all three variables will be equal.

Figure 3 contains the bounded buffer example of Sect. 2, augmented with auxiliary variables IN and OUT. They contain the sequences of messages sent from process I and received by O, respectively. Process I now sends e with "$B!(e, IN \circ e)$"[4] and O receives x with "$B?(x, OUT)$".

```
$out, count, done := 0, 0, false$;
do $\neg done \wedge count < N$; $I?(b(out \oplus count), IN)$      $\rightarrow count := count + 1$
□              $count > 0$; $O!(b(out), OUT \circ b(out))$   $\rightarrow out, count := out \oplus 1, count - 1$
□                          $I?quit$                     $\rightarrow done := true$
od;
$O!quit$
```

Fig. 3. A bounded buffer (with auxiliary variables)

[3] The set of matching pairs is determined from the program without auxiliary variables

[4] Operator "$\circ$" denotes catenation of an element to a sequence

Besides the definitions given in Fig. 1, the invariant of the loop of Fig. 3 contains the following predicate to relate the input stream, the output stream, and the buffered messages:

$$IN = OUT \circ b(out) \circ \ldots \circ b(out \oplus (count - 1)).$$

3.4. *Proof of Non-Interference*

Without auxiliary variables there is no need to prove non-interference. With disjoint state spaces, execution of one process cannot affect the state of, nor the validity of assertions about, another process (except when communication occurs, and the satisfaction proof takes care of this case). However, with auxiliary variables it is possible for execution of one process to affect assertions about another.

For each assertion P in process C it must be shown that P is invariant over any parallel execution. This is the non-interference property of Owicki [14].

Let us introduce some terminology. Command S is *parallel to* assertion P if S is contained in a process of a parallel command and P is contained in a different process of the same parallel command. A matching communication pair S and R is *parallel to* P if both S and R are parallel to P. Note that neither S nor R may appear in the same process as P.

Every command S parallel to P must satisfy

$$\{P \wedge pre(S)\}\, S\, \{P\}. \tag{3.4.1}$$

Similarly, every matching communication pair, $S{:}A!T(\bar{e})$ and $R{:}B?T(\bar{x})$, that is parallel to P must satisfy

$$(P \wedge pre(S) \wedge pre(R)) \Rightarrow P^{\bar{x}}_{\bar{e}}. \tag{3.4.2}$$

A proof of non-interference, if approached mechanically, is an awesome task. Every assertion in every process must be compared against every command in every other process and against every matching communication pair, so it takes time proportional to the product of the lengths of the processes. Fortunately, through judicious structuring of the program and careful selection of the assertions and auxiliary variables, it is possible to reduce the amount of work needed. The following notions of synchronously altered variables and universal assertions are important in designing good proofs.

3.5. *Synchronously Altered Variables*

Variable v is *synchronously altered in process A* if the only occurrences of v, outside of expressions, are in

1) the left part of assignments in A,
2) *receives* in A, and
3) *receives* in a process B, from A.

The value of an expression that contains only variables synchronously altered in A cannot change except when A progresses. Hence, there is no interference with assertions in the proof of A, provided that these assertions contain only variables synchronously altered in A.

More formally, a non-interference proof consists of proving instances of (3.4.1) and (3.4.2). But the definition of "parallel to" allows us to conclude that none of these instances will contain commands of the type described above. All other types of commands trivially satisfy (3.4.1) and (3.4.2) and there is no interference.

In many cases, synchronously altered variables arise naturally. In the bounded buffer example, IN and OUT are both synchronously altered in B and, respectively, in I and O. Hence, assertions about IN may be made in proofs about both I and B and no interference proof is needed.

Unfortunately, it is sometimes difficult to express global relations in terms of synchronously altered variables. The assertion that OUT is a prefix of IN is fine in B, but is subject to interference in I and O. The following notion of universal assertions is also convenient in limiting non-interference proofs.

3.6. *Universal Assertions*

Some assertions can be shown, in the sequential and satisfaction proofs, to be true initially, finally, and between every pair of commands in all processes. Such assertions are said to be *universal.*

Universal assertions are not subject to interference. Why? Consider the two cases of a non-interference proof. A universal assertion must hold after execution of a command or communication, given that the precondition and the universal assertion hold before. But the proof of universality of the assertion shows that it holds afterwards. Hence the proof must exist and need not be shown explicitly.

4. Requirements for Strong Correctness

4.1. *An Illustration of Proving Freedom from Deadlock*

The cooperative nature of CSP introduces a problem that does not exist in sequential notations. It is possible for a process to reach a point at which progress must wait for synchronization with another process.

A process waits when it is blocked. In and of itself, blocking is not bad. If, however, all processes are blocked or terminated and at least one process is blocked, then no progress can be made and blocking will not end. This situation is known as *deadlock.*

The basic idea for proving freedom from deadlock (see e.g. [14]), tailored to CSP, is as follows. Condider a possible deadlocked configuration: the processes that are ready to communicate are at commands $S_1, \dots, S_n$ and all other processes, $P_1, \ldots, P_m$, are terminated. Then the state of execution is one of the states represented by the predicate

$$P = pre(S_1) \wedge \ldots \wedge pre(S_n) \wedge post(P_1) \wedge \ldots \wedge post(P_m).$$

If it can be shown that either P is false – i.e. the state is impossible to reach during execution – or progress can occur in this state, then deadlock is not possible in this configuration. If this can be shown for each such configuration, then the program is free from deadlock.

An example will clarify the idea. Consider the program

$$\begin{array}{l} \{step=0\} \\ [A :: \{step=0\}\ B!(0, step+1)\ \{step=1\}\ B?(x, step)\ \{step=2\} \\ \| B :: \{step=0\}\ A?(z, step)\ \{step=1\}\ A!(3, step+1)\ \{step=2\}] \end{array}$$

Auxiliary variable step indicates at which commands execution of the program is. Using $\emptyset$ to indicate that a terminated process is waiting at no command, we enumerate the commands at which processes A and B may be waiting, along with the corresponding predicate *pre* or *post*:

$$\begin{array}{ll} A0 = [step=0: \{B!(0, step+1)\}] & B0 = [step=0: \{A?(z, step)\}] \\ A1 = [step=1: \{B?(x, step)\}] & B1 = [step=1: \{A!(3, step+1)\}] \\ A2 = [step=2: \emptyset] & B2 = [step=2: \emptyset] \end{array}$$

A possible deadlock configuration is determined by joining one of the Ai with one of the Bj (except for $A2$ and $B2$, which together describe the termination state of the whole program). These configurations are:

$$\begin{array}{l} join\ A0\ and\ B0: [step=0 \wedge step=0: \{B!(0, step+1), A?(z, step)\}] \\ join\ A0\ and\ B1: [step=0 \wedge step=1: \{B!(0, step+1), A!(3, step+1)\}] \\ join\ A0\ and\ B2: [step=0 \wedge step=2: \{B!(0, step+1)\}] \\ join\ A1\ and\ B0: [step=1 \wedge step=0: \{B?(x, step), A?(z, step)\}] \\ join\ A1\ and\ B1: [step=1 \wedge step=1: \{B?(x, step), A!(3, step+1)\}] \\ join\ A1\ and\ B2: [step=1 \wedge step=2: \{B?(x, step)\}] \\ join\ A2\ and\ B0: [step=2 \wedge step=0: \{A?(z, step)\}] \\ join\ A2\ and\ B1: [step=2 \wedge step=1: \{A!(3, step+1)\}] \end{array}$$

Any configuration with a predicate of the form $step=i \wedge step=j$ for $i \neq j$ can never be reached, for the predicate is equivalent to *false*. Each of the other configurations contains a matching communication pair, so that communication can occur. Hence, the program is free from the possibility of deadlock.

In proving freedom from deadlock, it is sufficient to argue as follows. Consider the three states at which A may be waiting or terminated, as given by $A0$, $A1$ and $A2$. For $A0$, show that $pre(A0)$ is enough to conclude that B is at the matching communication $B0$; for $A1$, show that $pre(A1)$ is enough to conclude that B is at the matching communication $B1$; for $A2$ show that B is also terminated.

4.2. A Sufficient Condition for Freedom from Deadlock

We have just given the general concept for proving the absence of deadlock. All that remains is to formally define the configurations of a program, based on the structure of the program, and what it means for a configuration to be deadlock-free.

A *configuration* K of command S corresponds to a possible waiting state of S: a state in which each process currently in execution may be either blocked or terminated and at least one process is not terminated. Formally, a configuration K consists of a pair

$$[\mathit{condition}\ (K): \mathit{commands}(K)]$$

The condition must hold if the processes of S are waiting in this configuration. Each element of the set $\mathit{commands}(K)$ is a guarded command $b \rightarrow c$, which indicates that c may be executed in order to make progress provided b is true.

We now define the *set of configurations* $C(S)$ *for a program* S *and its proof*, based on the structure of S. Note carefully that the set of configurations depends on the proof of the program and not just the program.

Simple commands cannot block, and hence there are no configurations to indicate possible waiting states:

$$C(\text{“}\mathit{skip}\text{”}) = C(\text{“}\bar{x} := \bar{e}\text{”}) = \emptyset$$

When a process is waiting at a command $A.T(\bar{x})$, $\mathit{pre}(\text{“}A.T(\bar{x})\text{”})$ is true. Under any condition, execution can continue as soon as the matching communication command is ready:

$$C(\text{“}A.T(\bar{x})\text{”}) = \{[\mathit{pre}(\text{“}A.T(\bar{x})\text{”}): \{\mathit{true} \rightarrow \text{“}A.T(\bar{x})\text{”}\}]\}$$

It is assumed that we can tell to which process a variable or command belongs; this is necessary when determining matching pairs later on.

A waiting state of S_1; S_2 is either a waiting state of S_1 or a waiting state of S_2:

$$C(\text{“}S_1;\ S_2\text{”}) = C(S_1) \cup C(S_2)$$

Execution of an alternative command **if** $\square_{i=1:n} b_i;\ c_i \rightarrow S_i$ **fi** can be waiting for a communication (or *skip*) c_i — if the corresponding guard b_i is *true*. It can also be waiting in one of the subcommands S_i:

$$C(\text{“}\mathbf{if}\ \square_{i=1:n} b_i; c_i \rightarrow S_i\, \mathbf{fi}\text{”})$$
$$= \{[\mathit{pre}(\text{“}\mathbf{if} \ldots \mathbf{fi}\text{”}): \cup_{i=1:n}\{b_i \rightarrow c_i\}]\} \cup (\cup_{i=1:n} C(S_i))$$

As with the alternative command, execution of a loop can be blocked wating for a communication at a guard. The condition in this case is the invariant of the loop. An extra guarded command $\neg BB \rightarrow \mathit{skip}$ is included in the guarded commands of the configuration because progress can be made when all the guards are false (the loop terminates).[5] Execution can also be blocked in one of the subcommands S_i.

[5] $BB = (\exists i: b_i)$

To handle distributed termination in the proof system, some changes would be necessary.

$$\begin{aligned} &C(\text{“}\mathbf{do}\ \square_{i=1:n} b_i;\ c_i \rightarrow S_i\ \mathbf{od}\text{”}) \\ &\quad = \{[inv(\text{“}\mathbf{do}\ldots\mathbf{od}\text{”}):\ \{\neg BB \rightarrow skip\} \cup (\cup_{i=1:n}\{b_i \rightarrow c_i\})]\} \\ &\qquad \cup(\cup_{i=1:n} C(S_i)) \end{aligned}$$

Finally, we have to define $C(S)$ for a parallel command $S = [A_i\colon\colon S_i \| \ldots \| A_n\colon\colon S_n]$. The set of configurations for the parallel command S is, in an informal sense, the set of all combinations of configurations of its subprocesses — with the combination of all the terminating configurations removed. (Refer back to the example of section 4.1). For example, suppose process $S1$ has two waiting configurations $[Q1\colon\ \{b1 \rightarrow c1\}]$ and $[Q2\colon\ \{b2 \rightarrow c2\}]$ and a terminating configuration $[T1\colon\ \emptyset]$. Suppose process $S2$ has 1 waiting configuration $[R\colon\ \{d \rightarrow e\}]$ and a terminating configuration $[T2\colon\ \emptyset]$. Then the parallel command $[S1 \| S2]$ has the waiting configurations

$$\begin{aligned} &[Q1 \wedge R\colon\ \{b1 \rightarrow c1, d \rightarrow e\}], \\ &[Q1 \wedge T2\colon \{b1 \rightarrow c1\}], \\ &[Q2 \wedge R\colon\ \{b2 \rightarrow c2, d \rightarrow e\}], \\ &[Q2 \wedge T\colon\ \{b2 \rightarrow c2\}], \quad \text{and} \\ &[T1 \wedge R\colon\ \{d \rightarrow e\}]. \end{aligned}$$

The terminating configuration for a process S_i, written $[post(S_i)\colon\ \emptyset]$, defines the state in which S_i has terminated and is waiting for the other processes to terminate. It is not in $C(S_i)$ and will have to be taken care of specially.

To define the configurations for the parallel command, we first define the *join*(SC), the “join” of a set of configurations SC:

$$join(SC) = [\wedge_{K \in SC}\ condition(K)\colon\ \cup_{K \in SC}\ commands(K)].$$

For $S = [A_1\colon\colon S_1 \| \ldots \| A_n\colon\colon S_n]$ *we then have*

$$\begin{aligned} C(S) = &\{join(\{a_i, i \in 1\colon n\}) \mid a_i \in (C(S_i) \cup \{[post(S_i)\colon \emptyset]\})\} \\ &- \{[(\wedge_{i=1:n} post(S_i))\colon \emptyset]\} \end{aligned}$$

We now have defined the set of configurations for any command (or program). For a program S to be deadlock free, for each configuration K of $C(S)$ it must be shown that K is either impossible to reach or that progress can be made in it. That is, either the condition of K is *false*, or K contains a guarded command $b \rightarrow skip$ for which b is *true*, or K contains two guarded commands $b1 \rightarrow r$ and $b2 \rightarrow s$ for which $b1$ and $b2$ are *true* and r and s are a matching communication pair. This is formalized in the following predicate DLF:

$$\begin{aligned} DLF(K) = &\neg\ condition\ (K) \vee \\ &(\exists (b \rightarrow skip) \in commands(K)\colon\ b) \vee \\ &(\exists (b1 \rightarrow r,\ b2 \rightarrow s) \in commands(K)\colon\ b1 \wedge b2 \wedge pair(r, s)) \end{aligned}$$

5. An Example – Finding the Minimum of a Set

5.1. *The Program*

Process B should receive from process A the minimum of the set $\{a_i | i \in 1:N\}$. Define process A to be the parallel execution of N processes *Least*(i), $i \in 1:N$, as shown in Fig. 4.

$$A :: [\|_{i=1:N} Least(i)]$$

```
Least (i) :: integer my_min, their_min, my_size, their_size;
    my_min, my_size := a_i, 1;
    do □_{j=1:N ∧ i≠j} 0 < my_size < N; Least (j)! (my_min, my_size)
          → my_size := 0
    □  □_{j=1:N ∧ i≠j} 0 < my_size < N; Least(j)? (their_min, their_size)
          → my_min, my_size := min(my_min, their_min), my_size + their_size
od;
if  my_size = 0 → skip
□   my_size = N → B! my_min
fi
```

Fig. 4. Least

Initially, each process *Least*(i) is responsible for the value a_i. As execution progresses, *Least*(i) is responsible for the minimum of a set of values, the number of values in this set being *my_size*. *Least*(i) tries to relieve itself of this responsibility by sending the minimum of the set, together with the size of the set, to another process *Least*(j) (say). If successful, the set of values for which *Least*(i) is responsible becomes empty, *Least*(i) terminates, and *Least*(j) becomes responsible for the minimum of the union of its own set and *Least*(i)'s. Execution continues until $N-1$ processes have become not responsible and one process is responsible for the minimum of the set of all values. This process then gives complete responsibility to B by sending B the minimum value.

To prevent deadlock, which would occur if all executing processes refused to receive, each process must agree to take on additional responsibility. *Least*(i) must continue to send or receive until it has become responsible for either the null set or a set of size N. In the second case, the minimum value that *Least*(i) has received is the minimum of all the a_i, and it sends this value to process B.

This explanation gives the flavor of the processes, but does not really provide sufficient information for answering questions about termination, deadlock or even partial correctness.

5.2. *Sequential Proof*

First we give a formalization of the *ad hoc* description of *Least*. Given the formal description, it is fairly straightforward to see that *Least* is weakly correct. The annotated program is given in Fig. 5.

```
$\{(\forall i\colon 1 \leqq i \leqq N\colon set_i = \{i\}) \wedge set_0 = \emptyset\}$
$[B\colon\colon \{M(0,0,set_0)\}$
    $\mathbf{if}\ \Box_{i=1:N} Least(i)?\,(m, set_0, set_i) \rightarrow skip\ \mathbf{fi}$
    $\{M(m, N, set_0)\}$
$\| A\colon\colon [\|_{i=1:N} \{set_i = \{i\}\}\ Least(i)\ \{M(0,0,set_i)\}]]$

$\{(\forall i\colon set_i = \{i\})\}$
$Least(i)\colon\colon$
    $\{set_i = \{i\}\}$
    $my\_min, my\_size := a_i, 1;$

    $\{M(my\_min, my\_size, set_i)\}$
    $\mathbf{do}\ \Box_{j=1:N \wedge i \neq j} 0 < my\_size < N;\ Least(j)!\ (my\_min, my\_size, set_i \cup set_j, \emptyset)$
        $\rightarrow \{M(my\_min, 0, set_i)\}$
          $my\_size := 0$
    $\Box\quad \Box_{j=1:N \wedge i \neq j} 0 < my\_size < N;\ Least(j)?\,(their\_min,\ their\_size,\ set_i, set_j)$
        $\rightarrow \{M(min(my\_min, their\_min),\ my\_size + their\_size,\ set_i)\}$
          $my\_min, my\_size := min(my\_min, their\_min),\ my\_size + their\_size$
    $\mathbf{od};$
    $\{M(my\_min, my\_size, set_i) \wedge (my\_size = 0 \vee my\_size = N)\}$
    $\mathbf{if}\ \ my\_size = 0 \rightarrow skip\{M(my\_min, 0, set_i)\}$
    $\Box\ \ my\_size = N \rightarrow \{M(my\_min, N, set_i)\}$
                          $B!(my\_min, set_i, \emptyset)$
                          $\{M(my\_min, 0, set_i)\}$
    $\mathbf{fi}$
    $\{M(my\_min, 0, set_i)\}$
```

Fig. 5. Least (annotated)

To remove concerns about repetitions in the set $\{a_i\}$, we deal with sets over the range $1{:}N$, corresponding to the elements $a_1, a_2, \ldots, a_N$. For each process *Least*(i) define auxiliary variable set_i as the indices of the values for which it is responsible. Therefore, my_min is the minimum and my_size the size of set_i. When *Least*(i) no longer has responsibility, $set_i = \emptyset$ and $my\_size = 0$. Furthermore, let auxiliary variable set_0 contain the set for which process B is responsible.

$M(my\_min,\ my\_size,\ set_i)$ is an invariant of the loop of each process, where M is defined by

$$M(mn, sz, S) \Leftrightarrow (sz = |S| \wedge (S = \emptyset \vee mn = \min_{j \in S} a_j)) \tag{5.2.1}$$

Predicate UNION, given in (5.2.2), expresses the fundamental property that exactly one process is responsible for each a_i – i.e. $set_0, set_1, \ldots, set_N$ form a partition of the integers $1{:}N$. It is universally true: initially, finally, and between any pair of commands in all processes. UNION will not be repeated at each assertion.

$$UNION: (\cup_{i=0:N} set_i) = 1:N \wedge (\forall i,j: 0 \leqq i < j \leqq N: set_i \cap set_j = \emptyset) \qquad (5.2.2)$$

The variant function t used to prove termination is $(\mathbf{N}i: set_i \neq \emptyset)$.[6] It is non-negative, non-increasing, and decreases with each message sent.

5.3. *Satisfaction Proof*

To prove satisfaction, we must show that (3.2.2) holds for each matching pair of the form (3.2.1). Examination of the program reveals two classes of matching pairs:

Lj: $Least(i)!\,(my\_min, my\_size, set_i \cup set_j, \emptyset)$	occurring in $Least(j)$,
Li: $Least(j)?\,(their\_min, their\_size, set_i, set_j)$	occurring in $Least(i)$

and

$B!(my\_min, set_i, \emptyset)$ occurring in $Least(i)$,
$Least(i)?(m, set_0, set_i)$ occurring in B.

Considering the first pair, Lj and Li, and priming local variables of $Least(j)$ to distinguish them from those of $Least(i)$, we must show that

$$(pre(Lj) \wedge pre(Li)) \Rightarrow (post(Lj) \wedge post(Li))^{their\_min,\ their\_size,\ set_i,\ set_j}_{my\_min',\ my\_size',\ set_i \cup set_j,\ \emptyset}$$

where

$$\begin{aligned}
pre(Lj) &= 0 < my\_size' < N \wedge M(my\_min', my\_size', set_j) \wedge UNION\\
pre(Li) &= 0 < my\_size < N \wedge M(my\_min, my\_size, set_i) \wedge UNION\\
post(Lj) &= M(my\_min', 0, set_j) \wedge UNION\\
post(Li) &= M(min(my\_min, their\_min), my\_size + their\_size, set_i) \wedge UNION
\end{aligned}$$

This straightforward exercise is left to the reader.

For the second matching pair, looking at the annotated proof in Fig. 5 we see that satisfaction holds if

$$\begin{aligned}
&(M(0,0,set_0) \wedge M(my\_min, N, set_i) \wedge UNION)\\
&\Rightarrow\ (M(m, N, set_0) \wedge M(my\_min, 0, set_i) \wedge UNION)^{m,\ set_0,\ set_i}_{my\_min,\ set_i,\ \emptyset}
\end{aligned}$$

which obviously holds.

5.4. *Non-Interference Proof*

The auxiliary variables in the proof are: $set_0, \ldots, set_N$. Variable set_i (for $i \in 1:N$) is altered in two types of communications: in the receiving guards of process

[6] An expression of the form $(\mathbf{N}i: P(i))$ denotes the number of values i for which $P(i)$ is true

Least(i) and in the guards of *Least*(j) that receive from *Least*(i). Thus set_i is synchronously altered in *Least*(i). Variable set_0 is only altered in a *receive* in B and so is synchronously altered in B. The only auxiliary variable referred to in assertions in *Least*(i) (except for the universal assertion *UNION*) is set_i; hence, assertions in this proof refer only to synchronously altered variables and there is no interference.

5.5. *Proof of Freedom from Deadlock*

We must show that in all "waiting states" either the state cannot be encountered during execution or at least one process may make progress. We classify possible waiting states into as few cases as possible, in order to keep case analysis to a minimum.

Case 1. process Least(i) is prepared to communicate with another process Least(j) in the main loop of Least(i). In this case, we have $0<my\_size<N$. From *UNION* and the fact that auxiliary variable set_0 for process B has size 0 or N, we note that local variable *my_size* of some other process *Least*(j) also satisfies $0<my\_size<N$. The annotated proof assures us that process *Least*(j) can *only* be in the same position: prepared to communicate with another process in the main loop. Since both processes can send and receive, both processes are ready and progress can be made.

Case 2. process Least(i) is prepared to communicate by sending a message to process B. In this case, $my\_size=N$. Because of the universal assertion *UNION*, we have $set_0=\emptyset$. The assertions in B's annotated proof assure us that B must be waiting to receive the message from *Least*(i) and, since these two communication commands match, both processes are ready and progress can occur.

Case 3. process B is prepared to receive from Least(i). This means that $set_0=\emptyset$. Therefore, by assertion *UNION*, set_i is nonempty for at least one process *Least*(i). Hence, *Least*(i) has not terminated. If it is waiting, it is waiting in the main loop to communicate with some *Least*(j) or it is waiting to send to B, cases that have already been shown to be deadlock free.

We have investigated each waiting state of *Least*(i) and each waiting state of B and shown that, indeed, these states are free from deadlock. This ends the proof of absence from deadlock. The fact that only three cases are needed with this program lends some credence to the practicality of the method.

6. Discussion

We have shown how to extend a proof method for sequential programs to encompass communication. The satisfaction proof formalizes the intuitive argument that says that communication is distributed assignment.

The system presented views communication as a means to an end; that is, processes are sequential programs with communication providing external information. In other proof systems the sequence of messages produced is the purpose of the process; sequential programs provide a means of controlling the communications.

Proof systems that are based on the history of communication introduce variables that record each *send* and *receive*. Rather than include this in our proof rules we allow auxiliary variables, which can be used to record as much or as little history as is needed.

Unlike the repetitive command described in [11], our repetitive command does not allow termination to occur because other processes are terminated. Instead, termination only occurs when all boolean guards are false. This makes the termination conditions explicit and simplifies both the proof rule and the implementation. The sufficient condition for absence of deadlock would also be more complicated if distributed termination were included. See Levin [13] for a rule that handles distributed termination.

Further research needs to be done regarding deadlock and starvation. The suggested approach to deadlock requires too much in the way of case analysis, a common source of error. The problem of starvation is ignored; instead, all processes are required to terminate. The problem of dealing with non-terminating processes is an area for future research.

While preparing this paper, we learned of similar research being done by Apt, Francez, and de Roever [1]. Their system treats partial instead of total correctness, although the change to the latter is slight. [1] deals with distributed termination; this paper does not. [1] contains the same axioms for *send* and *receive*, although our motivation for these axioms may be more appealing.

A property of *cooperation* in [1] corresponds to our property of satisfaction. Cooperation is different in that it is derived from the forward assignment rule and that a global invariant is used to relate auxiliary variables, rather than allowing shared references to auxiliary variables. This global invariant is then used to eliminate matching communication pairs that cannot synchronize. The idea to extend matching communication pairs to allow assignments to auxiliary variables during communication, as a means for reducing the work involved in proving non-interference, was not present in their work. Instead they allow sections of code to be considered atomic; this is the means for changing local variables synchronously in the processes.

Initially, we had separate axioms for *send* and *receive*. The *receive* axiom had the form it does now; there was no relation between the pre- and postconditions. This was in recognition of the command's ability to change the value of its parameter in a way not determined by the command itself. The *send* axiom was the same as that for *skip*; after all, sending a message should not affect the state of the sender.

After a time, we found that the *send* axiom was not strong enough, and, realizing that in terms of weak correctness one should be able to assert anything after a *send* executing in isolation, we changed it. We also liked the simplicity of one axiom for communication. We observed that in Hoare's formal model [11] there is no real difference between *send* and *receive*. In his model there is only synchronization, where *receive* is an abbreviation for a (possibly infinite) set of alternatives, the choice of which is determined by synchronization and determines the value received. This recognizes that synchronization provides an information flow into a sending process. This observation strengthened our opinion that the *send* axiom we now have is the correct one.

A paper by P. Cousot and S. Cousot [5] also deals with a proof system for communicating sequential processes. It is, however, even more formal than this paper, and we have difficulty understanding it.

Mention should be made of reference [4] by K. Chandy and J. Misra, which presents a new approach to proving CSP programs correct. The approach is noteworthy in at least two respects; it does not require the programmer to find auxiliary variables the way ours and [1] does, and it is designed to handle processes in isolation. Externally, a process is described in terms of input/output sequences. These sequences contain the information that would be recorded in auxiliary variables in our system. Using these external specifications, systems of processes can be combined. In combining processes, some communications become hidden, invisible to processes outside the system; the combination of processes yields internal specifications that may be simplified to yield external specifications.

Acknowledgements. It is a pleasure to acknowledge the assistance of Fred Schneider and Carl Hauser in performing this research and in reading earlier drafts of this paper. We also would like to thank the referees for their careful, yet rapid, reading of this paper and their helpful suggestions. Finally, we must thank Krzysztof Apt, Nissim Francez, and Willem de Roever for their cooperation. Despite the similarity of this work to their own, they encouraged us in our intentions to submit this article.
This research was supported in part by the National Science Foundation under grant MCS-7622360.

References

1. Apt, K., Francez, N., de Roever, W.: A proof system for communicating sequential processes. TOPLAS **2**, 359–385 (1980)
2. Bernstein, A.: Output guards and non-determinism in "Communicating Sequential Processes". TOPLAS **2**, 234–238 (1980)
3. Chandy, K., Misra, J.: An axiomatic proof technique for networks of communicating processes. TR-98, Dept. of Computer Science, Univ. of Texas at Austin, 1979
4. Chandy, K., Misra, J.: Proofs of networks of processes. Technical Report, Dept. of Computer Science, Univ. of Texas at Austin, 1980
5. Cousot, P., Cousot, R.: Semantic analysis of communicating sequential processes. In: Automata, languages and programming (J.W. de Bakker, J. van Leeuwen, eds.). Lecture Notes in Computer Sciences, Vol. 85, pp. 119–133. Berlin-Heidelberg-New York: Springer 1980
6. Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, N.J.: Prentice-Hall 1976
7. Francez, N.: Distributed termination. TOPLAS **2**, 42–55 (1980)
8. Gries, D., Levin, G.M.: Multiple assignment and procedure call proof rules. TOPLAS **2**, 564–579 (1980)
9. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM 12, 576–580, 583 (1969)
10. Hoare, C.A.R.: Communicating sequential processes. CACM **21**, 666–677 (1978)

11. Hoare, C.A.R.: Towards a theory of communicating sequential processes. Programming Methodology Conference at Santa Cruz, August 1979
12. Levin, G.M.: Proof rules for communicating sequential processes. Ph.D. thesis, Dept. of Computer Science, Cornell University, August 1980
13. Levin, G.M.: A proof technique for communicating sequential processes (with an example). TR 79-401, Computer Science Dept., Cornell Univ., 1979
14. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informat. **6**, 319-340 (1976)
15. Schneider, F.B.: Synchronization in a distributed environment. TR 79-391, Dept. of Computer Science, Cornell Univ. TOPLAS (1981, in press)
16. Silberschatz, A.: Communication and synchronization in distributed systems. IEEE Trans. Software Engrg. SE-**5**, **6**, 542-546 (1979)

Received July 4, 1980/January 30, 1981