A PROOF TECHNIQUE FOR COMMUNICATING SEQUENTIAL PROCESSES (WITH AN EXAMPLE)

Gary Marc Levin *

TR79-401

Computer Science Department
Cornell University
Ithaca, N.Y. 14853

*This research supported in part by the National Science Foundation Under Grant MCS-7622360.

A Proof Technique for Communicating Sequential Processes (with an example)

Gary Marc Levin

Cornell University

ABSTRACT

We present proof rules for an extension of the Communicating Sequential Processes proposed by Hoare. The send and receive statements are treated symmetrically, simplifying the rules and allowing send to appear in guards. An example is given to explain the use of the technique. This is an outline of a substantial part of a Ph.D. thesis that is expected to be completed in June 1980.

This research supported in part by the National Science Foundation under Grant MCS-7622360.

A Proof Technique for Communicating Sequential Processes (with an example)

Gary Marc Levin

Cornell University

1. Introduction

When considering systems of processes, two models come to mind. The first is *centralized* and all processes share (have access to) all variables. Alternatively, we may consider *distributed* processes, in which the variables of each process are private, i.e., not accessible to other processes. Instead, message passing is used to allow interaction between processes.

The language Communicating Sequential Processes (CSP), defined by Hoare in [HO], is intended to describe distributed systems of processes. It is derived from Dijkstra's simple programming language described in [DI]. The set of simple statements (assignment and skip) has been augmented by two communication statements, send and receive. Send and receive serve to pass information and to synchronize processes. The alternative and repetitive statements have been extended to allow send and receive to appear in guards. A new form of statement composition, parallel composition, allows defining and naming of processes, where the variables of parallel processes are disjoint. Both processes and alternatives can be parameterized to introduce abbreviations.

The approach presented in this paper separates the *sequential proof* of processes, which may need assumptions about the effects of communication, and the *satisfaction proof* that these assumptions are valid. This view requires that the system be taken as a whole; proofs of processes in isolation are not handled.

A brief description of CSP is given in section 2. We give in the next section a program that illustrates the use of the language. Section 4 presents proof rules for correctness (in the absence of deadlock), while section 5 applies these rules to the program of section 3. Similarly, section

This research supported in part by the National Science Foundation under Grant MCS-7622360.

6 presents requirements for showing a set of processes to be deadlock free, while section 7 proves the program of section 3 to be deadlock free.

2. The language CSP

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

In sequential languages, there is the tacit assumption that a program continues executing at some finite speed until it reaches the end or an error. This allows the conclusion that a program that could terminate, will in fact terminate. In CSP, where there can be more than one process, such an assumption is insufficient. We must state both when a process can proceed and when it must proceed.

We say that a process is ready, blocked, or terminated. A _ready_ process can execute a statement. A _blocked_ process is waiting for communication so that it can continue. A blocked process is said to be _prepared to communicate_ (send or receive) with one or more processes. A process that has executed its last statement is _terminated_. Execution of a send is always simultaneous with execution of a receive; the processes are said to have _synchronized_ at the send (receive).

So long as some process is ready, there must be progress in a finite time. This is a minimal assumption, allowing any scheduling scheme to be used in implementation.

Because we treat send and receive in similar ways, it will frequently be convenient to speak of a _communication statement_; that is, either a send or a receive. This is denoted as A.T(x). This notation never appears in the text of a program; it merely reduces repetition in definitions. A communication statement has 3 parts: A, a process name; T, a template (any identifier or null); and x, a vector of expressions or variables. Where the template is null and the vector is of length 1, we drop the parenthesis. A send has the character "!" in place of the period; receive, a "?".

The simple statements

Assignment

$x_1,\ldots,x_n := e_1,\ldots,e_n$

When execution reaches this statement, the process is ready. The values $v_1,\ldots,v_n$ of expressions $e_1,\ldots,e_n$ are computed; the references $r_1,\ldots,r_n$ represented by variables $x_1,\ldots,x_n$ are determined; and then $v_1,\ldots,v_n$ are assigned to $r_1,\ldots,r_n$ in left-to-right order.

Skip

skip

When execution reaches this statement, the process is ready. Execution of this statement has no effect.

Send

A!T(e)

Assume that this statement appears in process B. When execution reaches this statement, process B is prepared to send to A a message with template T. Execution in B is blocked except when process A is prepared to receive from B a message with template T, e.g., B?T(x). When A is prepared to receive, process B is ready. On execution the values of the expressions e are assigned to the corresponding variables in the receive in left-to-right order.

Receive

B?T(x)

Assume that this statement appears in process A. When execution reaches this statement, process A is prepared to receive from B a message with template T. Execution in A is blocked except when process B is prepared to send to A a message with template T, e.g., A!T(e). When B is prepared to send, process A is ready. On execution the variables x are assigned the values of the corresponding expressions in the send in left-to-right order.

[illegible] composition

Sequence

$S_1; S_2$

Execute S_1, then execute S_2.

Alternative

if $b_1; c_1 \rightarrow S_1 \ \square \ \cdots \ \square \ b_n; c_n \rightarrow S_n$ **fi**

where c_i = skip

If b_i is false, the guard is aborted. If b_i is true, the guard is ready.

where c_i = A.T(x)

If b_i is false or when A is terminated, the guard is aborted. If b_i is true, this process is prepared to communicate with process A. When A is prepared to communicate, the guard is ready; at other times it is blocked.

One and only one ready guard can be selected. When it is selected, the corresponding statement is executed. If all guards abort, the alternative statement aborts. The statement is ready whenever some guard is ready. If the i^{th} guard is selected, c_i and only c_i must be executed, followed by the execution of S_i.

The insistence on non-null expressions for the b_i and non-null commands for the c_i is for purposes of explanation only. In programs, the boolean true may be omitted from the guard, as may the command skip. In these cases, the separating semicolon is omitted as well.

Repetitive

do $b_1; c_1 \rightarrow S_1 \ \square \ \cdots \ \square \ b_n; c_n \rightarrow S_n$ **od**

The interpretation of the guards is the same as for the alternative statement. When a guard "$b_i; c_i$" is selected, the corresponding statement S_i is executed. This is repeated until all guards abort. The process is ready when any guard is ready.

Parallel

[A:: S_1 || B:: S_2 || ... || C:: S_n]

Process S_1 is named A, process S_2 is named B, etc. Execution of $S_1,\ldots,S_n$ proceed in parallel. The scope conventions for the names are similar to those used in Algol 60. For example, any statement of S_1, including statements in nested parallel statements, may communicate with B. The matching communication statement in B must reference A, because no names internal to S1 are known in B. The reason for these conventions is that from the outside it should not be possible to know the composition of a process.

Parameterization

When many similar processes are desired we allow the abbreviation

[(i: 1..n) A(i):: S(i)]

to represent

[A(1):: S(1) || ... || A(n):: S(n)].

Similar conventions apply to guarded command lists in the alternative and repetitive statements.

3. A example: Finding the minimum of a set.

We want to receive in process P from process A the minimum of the set $\{a_i \mid i \in 1..N\}$. Define process A to be the parallel execution of N processes Min(i), i∈1..N.

```
A:: [(i:1..N) Min(i) ]

Min(i):: integer my_min, their_min; boolean sent;
    my_min,sent:= a_i,false;

    do  (j:1..N ∧ i≠j) ¬sent; Min(j)!my_min
          → sent:= true

     □  (j:1..N ∧ i≠j) ¬sent; Min(j)?their_min
          → my_min:= min(my_min,their_min)
    od;

    if   sent → skip
     □  ¬sent → P!my_min
    fi
```

Each process must send the minimum value it has received. (We consider Min(i) to have received a_i.) Having done so, it terminates without further communication. Initially Min(i) is "responsible" for a_i. When Min(i) sends to Min(j), Min(j) becomes responsible for the elements for which Min(i) had been responsible.

To prevent deadlock, which would occur if no process were ready to receive, each process can also receive a similar message from any other process. Min(i) must continue to receive until it has either sent its minimum to some other Min(j) or until there are no other Min(j) with which to communicate. In the second case, the minimum value that Min(i) has received is the minimum of $\{a_i \mid i \in 1..N\}$. Therefore, it sends this value to process P.

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.

4. Proof rules for weak correctness

A proof of <u>weak correctness</u> (that is, correctness in the absence of deadlock) of a set of communicating processes consists of three parts: a <u>sequential</u> proof, a <u>satisfaction</u> proof, and <u>non-interference</u> proof. The sequential proof for each process is in the style of a Hoare-logic proof (rules given below). While creating the sequential proof, assumptions are made about the effect of communication statements. A satisfaction proof shows that these assumptions are valid. Although the original language requires that variables be accessible in at most one process, auxiliary variables are allowed to appear in more than one process. This is reasonable because auxiliary variables are not needed to correctly execute a program and so do not violate the distributed model. With the addition of auxiliary variables, it becomes necessary to show that execution of other processes cannot interfere with the validity of assertions. The notion of synchronously altered variables is introduced to simplify the proof of non-interference.

The axioms and rules of inference

{P} S {Q} means that if S is started in a state satisfying P, S will terminate in a state satisfying Q. The notation P^x_e represents the predicate P with all free occurrences of x replaced by e. In many cases x and e will be vectors; then elements of x are replaced by corresponding elements of e.

Axioms

Assignment

$\{P^x_e\}$ x:= e {P}

Skip

{P} skip {P}

Receive and Send

{P} A.T(x) {Q}

This rule seems to violate the law of the excluded miracle [DI], and so it does. We explain this seeming miracle by a satisfaction proof.

Inference rules

Sequence

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

Consequence

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

Alternative

$$\frac{\begin{array}{l}\text{for all } i,\ \{P \wedge b_i\}\ c_i;\ S_i\ \{Q\} \\ P \Rightarrow (\exists i\colon b_i \wedge \neg \mathrm{term}(c_i))\end{array}}{\{P\}\ \mathbf{if}\ \ b_1;\ c_1 \rightarrow S_1\ \square\ \cdots\ \square\ b_n;\ c_n \rightarrow S_n\ \mathbf{fi}\ \{Q\}}$$

where

term(skip) = false

term(A.T(x)) = post(A)

The proofs of the hypotheses will include instances of the axiom for communication statements, necessitating a satisfaction proof.

Note that process A cannot terminate in a state in which term(A.T(x)) is false.

Repetitive

$$\frac{\begin{array}{l}\text{for all } i,\ \{P \wedge b_i\}\ T{:=}\ t;\ c_i;\ S_i\ \{P \wedge t<T\} \\ (P \wedge (\exists i\colon b_i \wedge \neg \mathrm{term}(c_i))) \Rightarrow t>0\end{array}}{\begin{array}{l}\{P\}\ \mathbf{do}\ \ b_1;\ c_1 \rightarrow S_1\ \square\ \cdots\ \square\ b_n;\ c_n \rightarrow S_n\ \mathbf{od} \\ \qquad \{P \wedge (\forall i\colon \neg b_i \vee \mathrm{term}(c_i))\}\end{array}}$$

where t is an integer function and T is a fresh variable.

Parallel

$$\frac{\text{for all } i,\ \{P_i\}\ S_i\ \{Q_i\}}{\{(\forall i\colon P_i)\}\ [\ A_1{::}\ S_1\ \|\ \cdots\ \|\ A_n{::}\ S_n\ ]\ \{(\forall i\colon Q_i)\}}$$

Satisfaction proof

Consider any communication statement S with its pre- and post-conditions. We allow any post-condition because the change in state is effected by communication, and not just by execution of S. Hence the requirement that all statements that could communicate with S be shown to justify the post-condition.

A _possible communication pair_ consists of a receive in B of the form A?T(x) and a send in A of the form B!T(e). The pair must match with respect to process names and templates. That is, the send must be a statement contained in the process named by the receive, and _vice versa_.

In the proof, the pair appears as: {P} A?T(x) {Q} and {R} B!T(e) {S}. Should these two statements communicate, the effect would be equivalent to the assignment x:= e. Hence, before the communication can be executed, we need $(Q \wedge S)^x_e$ and we know $(P \wedge R)$. Therefore, the assumptions Q and S are satisfied if:

$$(P \wedge R) \Rightarrow (Q \wedge S)^x_e.$$

With the initial assumption that the variables of different processes are distinct, P can have no bearing on S (which would be the same as S^x_e). But the sending process should be allowed to deduce facts from P. To model this, we allow _auxiliary variables_ to appear in more than one process, giving relations between the states of different processes.

Note that if $\neg(P \wedge R)$, then it is impossible for these statements to synchronize.

Auxiliary variables

The addition of auxiliary variables to processes and their proofs allows assertions in distinct processes to refer to non-disjoint state spaces. This is important, particularly in satisfaction proofs, in order to relate the program variables of one process with the program variables of another. Auxiliary variables are defined by Owicki and Gries [OW] for use in proofs in the centralized model. Our definition is adapted to include communication.

An auxiliary variable may not affect the flow of control of the process, nor may it affect the value of any non-auxiliary variable. If these conditions are met, then the auxiliary variables are not necessary to the computation and may be omitted from the program, but not from the proof. These conditions are ensured if the following set of syntactic restrictions are met.

Auxiliary variables may only appear:

1) in assertions;

2) in expressions being assigned to auxiliary variables;

3) as parameters in a receive;

4) in expressions as parameters of a send corresponding to auxiliary variables in any receive that forms a possible communication pair.

<u>Proof of non-interference</u>

Until the introduction of auxiliary variables, there was no need to prove non-interference. Now, however, it is possible for execution of one process to affect the assertions (that is, the pre-conditions of statements and post-conditions of processes) of another.

For each assertion P in process C it must be shown that P is invariant over any parallel execution. This is similar to the non-interference property of Owicki and Gries [OW]. In addition to showing for every statement S parallel to P that

$$\{P \wedge pre(S)\}\ S\ \{P\}$$

we must show for any possible communication pair, S: A!T(e) and R: B?T(x):, parallel to P that

$$(P \wedge pre(S) \wedge pre(R)) \Rightarrow P^x_e$$

S is <u>parallel to</u> P if S is contained in a process of a parallel statement and A is contained in a different process of the same parallel statement. A possible communication pair S and R is <u>parallel to</u> P if both S and R are parallel to P. Note that neither S nor R may appear in process C.

The non-interference proof for statements shows that if P is true before execution of S then P will be true after execution. The proof for possible communication pairs recognizes that communication between S and R is equivalent to the assignment x:= e.

Although at first this seems to require a large number of proofs, a judicious choice of auxiliary variables can make these proofs trivial. A basis upon which the choice of variables may be made is the concept of synchronously altered variables.

A variable v is _synchronously altered in process A_ if the only occurrences of v, aside from appearances in expressions, are in any or all of the following cases.

1) The left part of an assignment in process A.
2) A receive in process A.
3) A receive in process B, from process 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 the assertions contain only variables that are synchronously altered in A.

5. A proof of weak correctness for Min

With these proof rules a formalization of the _ad hoc_ description of Min can be given. Given the formal description, it is fairly straightforward to see that Min is weakly correct.

To remove concerns about repetitions in the set $\{a_i\}$, we deal with sets over the range 1 to N, corresponding to the elements $a_1, a_2, \ldots, a_N$. Each process Min(i) has a local variable set_i defined as the set of indices for the set of values of which my_min is the minimum. (The set of values for which Min(i) is "responsible".) Once Min(i) has sent its minimum, set_i will have the value $\emptyset$. The function MIN refers to this relation.

$$MIN(x,S) \Leftrightarrow (S=\emptyset \vee x=\min_{i \in S} a_i)$$

A fundamental property of the processes is that at any time exactly one process is "responsible" for each a_i. The predicate UNION, which embodies this idea, is universally true, i.e., initially, finally, and between any pair of statements in all processes. UNION will not be repeated at each assertion. It should, however, be checked.

UNION: $$\bigcup_{i \in 1..N} set_i = 1..N \wedge (\forall i,j: i \neq j: set_i \cap set_j = \emptyset)$$

Assume initially $(\forall i: set_i = \{i\})$.

```
[ P:: {true} A?(m,answered) {MIN(m,1..N)} | A::[ (i:1..N) Min(i) ]]

{ (∀i: set_i={i}) }
Min(i)::
    { set_i={i} }
    my_min,sent:= a_i,false;

    { (set_i=∅)⇔sent ∧ MIN(my_min,set_i) }
    do  (j:1..N ∧ i≠j) ¬sent; Min(j)!(my_min,set_i∪set_j,∅)
         → { set_i=∅ ∧ MIN(my_min,set_i) }
            sent:= true

     □  (j:1..N ∧ i≠j) ¬sent; Min(j)?(their_min,set_i,set_j)
         → { (set_i=∅)⇔sent ∧ MIN(min(my_min,their_min),set_i) }
            my_min:= min(my_min,their_min)
    od;

    { (set_i=∅)⇔sent ∧ MIN(my_min,set_i)
       ∧ (∀j: j≠i: sent ∨ set_j=∅ ∨ (∀k: k≠j:set_k=∅)) }

    { (set_i=∅)⇔sent ∧ MIN(my_min,set_i)
       ∧ (sent ∨ (∀j: j≠i: set_j=∅))
       ∧ (sent ∨ set_i=1..N) }
    if   sent → skip
     □  ¬sent → P!(my_min,i)
    fi
    { set_i=∅ ∨ (∀j: j≠i: set_j=∅) }
```

The variant function t is $(\blacksquare i: set_i \neq \emptyset)$. It is non-negative, non-increasing, and decreases with each message sent. In this proof, the pre-condition of the loop is also the loop invariant. While most of the steps of the proof are applications of the assignment rule or problems postponed to the satisfaction proof, the application of the rule of consequence after the loop is difficult to follow; therefore we expand upon this point in the proof.

$$(set_i=\emptyset)\Leftrightarrow sent \wedge MIN(my\_min,set_i)$$
$$\wedge\ (\forall j: j\neq i: sent \vee set_j=\emptyset \vee (\forall k: k\neq j: set_k=\emptyset))$$

Factor out the term sent:

$$(set_i=\emptyset)\Leftrightarrow sent \wedge MIN(my\_min,set_i)$$
$$\wedge\ (sent \vee (\forall j: j\neq i: set_j=\emptyset \vee (\forall k: k\neq j: set_k=\emptyset)))$$

Suppose sent is false. Then $set_i \neq \emptyset$. Therefore, for any j not equal to i, there is some k not equal to j (i.e. i) that has $set_k \neq \emptyset$. Hence the term $(\forall k: k \neq j: set_k = \emptyset)$ must be false, and the above is equivalent to

$$(set_i=\emptyset) \Leftrightarrow sent \wedge MIN(my\_min, set_i)$$
$$\wedge\ (sent \vee (\forall j: j \neq i: set_j = \emptyset))$$

Now, recalling that UNION is universally true, we conclude from $(\forall j: j \neq i: set_j = \emptyset)$ that $set_i = 1..N$, and hence:

$$(set_i=\emptyset) \Leftrightarrow sent \wedge MIN(my\_min, set_i)$$
$$\wedge\ (sent \vee (\forall j: j \neq i: set_j = \emptyset))$$
$$\wedge\ (sent \vee set_i = 1..N)$$

In the proof of non-interference, it is this assertion that must be checked and not the post-condition of the loop. In the annotation of the program, the post-condition of the loop is unnecessary, but was included to make the proof easier to read.

Satisfaction proof

Examination of the program reveals that there are two classes of possible communication pairs, Min(i) sends to Min(j), for $i \neq j$; and Min(i) sends to P. To facilitate reading, the communication statements have been copied, along with their pre- and post-conditions. In the assertions and statements of Min(j), necessary substitutions have been made. Additionally, the variables local to Min(j) have been primed to avoid name conflicts.

For all i,j where $i \neq j$.

Min(i):

$\{\ \neg sent\ \wedge (set_i=\emptyset) \Leftrightarrow sent\ \wedge MIN(my\_min, set_i)\ \wedge UNION\ \}$

$Min(j)!(my\_min, set_i \cup set_j, \emptyset)$

$\{\ set_i=\emptyset \wedge MIN(my\_min, set_i) \wedge UNION\ \}$

Min(j):

$\{\ \neg sent' \wedge (set_j=\emptyset) \Leftrightarrow sent' \wedge MIN(my\_min', set_j) \wedge UNION\ \}$

$Min(i)?(their\_min', set_j, set_i)$

$\{\ (set_j=\emptyset) \Leftrightarrow sent' \wedge MIN(min(my\_min', their\_min'), set_j) \wedge UNION\ \}$

$$\neg sent \wedge (set_i=\emptyset)\Leftrightarrow sent \wedge MIN(my\_min,set_i) \wedge$$
$$\neg sent' \wedge (set_j=\emptyset)\Leftrightarrow sent' \wedge MIN(my\_min',set_j) \wedge UNION$$

$$\Rightarrow [(set_j=\emptyset)\Leftrightarrow sent' \wedge MIN(min(my\_min',their\_min'),set_j) \wedge set_i=\emptyset \wedge MIN(my\_min,set_i) \wedge UNION]^{their\_min',\ set_j,\ set_i}_{my\_min,\ set_i \cup set_j,\ \emptyset}$$

Min(i):

$\{ \neg sent \wedge (set_i=\emptyset)\Leftrightarrow sent \wedge MIN(my\_min,set_i) \wedge (sent \vee (\forall j\colon j\neq i\colon set_j=\emptyset)) \wedge (sent \vee set_i=1..N) \wedge UNION \}$

P!(my_min,i)

$\{ (set_i=\emptyset \vee (\forall j\colon j\neq i\colon set_j=\emptyset)) \wedge UNION \}$

P:

{ true }

A?(m,answered)

$\{ MIN(m,1..N) \}$

$$\neg sent \wedge (set_i=\emptyset)\Leftrightarrow sent \wedge MIN(my\_min,set_i)$$
$$\wedge (sent \vee (\forall j\colon j\neq i\colon set_j=\emptyset)) \wedge (sent \vee set_i=1..N) \wedge UNION$$

$$\Rightarrow [(set_i=\emptyset \vee (\forall j\colon j\neq i\colon set_j=\emptyset)) \wedge MIN(m,1..N) \wedge UNION]^{m,\ answered}_{my\_min,\ i}$$

Proof of non-interference

A proof of non-interference, if approached mechanically, is an awesome task. Every assertion in every process must be compared against every statement in every other process and against every possible communication pair. Fortunately, through good structuring of the program and careful selection of the assertions and auxiliary variables, it is possible to reduce the amount of work needed.

Most terms in most assertions in this proof refer only to variables that are synchronously altered with respect to the process in which the assertion appears. Any terms not covered in this way will be handled separately.

The auxiliary variables in the proof are: $set_1,\ldots,set_N$ and answered. No assertions are made about answered, which is included for the strong correctness proof. The variable set_i is altered in two types of communications: in the receiving guards of process Min(i), and in the guards of Min(j) that receive from Min(i). This makes set_i synchronously altered in

Min(i).

Given this information the reader may verify that the only term appearing in the proof of Min(i) that can change value is

$$P: (\forall j: j \neq i: set_j = \emptyset)$$

There are no assignments to set_j. Hence only possible communication pairs that affect set_j and occur in Min(j) and Min(k), where i, j, and k are all distinct, need to be checked. The possible communication pairs are of the form Min(j)?(their_min,set_k,set_j) in Min(k) and Min(k)!(my_min,$set_j \cup set_k$,$\emptyset$) in Min(j).

The pre-condition of the send in Min(j) is

$$Q: \neg sent \wedge (set_j = \emptyset) \Leftrightarrow sent \wedge MIN(my\_min, set_j) \wedge UNION$$

But P $\wedge$ Q is false, implying that there is no interference.

6. Requirements for strong correctness

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

A waiting process is said to be blocked. In and of itself, blocking is not bad. If, however, all processes are blocked or terminated then no progress can be made and the blocking will not end. This condition is known as deadly embrace or deadlock.

Blocking can only occur at communication statements, communicating guards with true booleans, and at the end of processes in parallel statements. If it can be shown for every set of points at which S could synchronize that either some process is ready or the conjunction of the assertions at these points is false, then there can be no deadlock.

A *configuration* of S is a set of statements in S (including possibly the end of a process) at which S could synchronize. We denote the end of process P by the term "end_P". The set of configurations for a statement S is C(S).

S	C(S)
skip	$\emptyset$
x:= e	$\emptyset$
X.T(x)	{ {S} }
S_1; S_2	$C(S_1) \cup C(S_2)$
if b_1; $c_1 \rightarrow S_i$ $\Box \cdots \Box$ b_n; $c_n \rightarrow S_n$ **fi**	$\cup_i C(S_i)$ $\cup$(if $\exists i$: $c_i \neq$skip then { {S} } else $\emptyset$)
do b_1; $c_1 \rightarrow S_1$ $\Box \cdots \Box$ b_n; $c_n \rightarrow S_n$ **od**	$\cup_i C(S_i)$ $\cup$(if $\exists i$: $c_i \neq$skip then { {S} } else $\emptyset$)
[X_1:: S_1 $\Vert \cdots \Vert$ X_n:: S_n]	$\{ \cup\{x_1, x_2, \ldots\} \mid x_i \in \{ \{end\_X_i\} \} \cup C(S_i) \}$ $- \{ \{end\_X_i \mid i \in 1,2,\ldots\} \}$

It is assumed that instances of statements are distinguishable and that a configuration can be indexed by process name.

For each configuration Γ in C(S), the following must be shown to prove absence of deadlock in S.

(DL) $(\forall s: s \in \Gamma: pre(s)) \Rightarrow progress(\Gamma)$

$$\text{where } progress(\Gamma) = (\exists s \in \Gamma: s = \text{"}\mathbf{do}\ b_i;\ c_i \rightarrow S_i\ \mathbf{od}\text{"}: (\forall i: \neg end\_A_i \in \Gamma: \neg b_i)) \vee (\exists s, r \in \Gamma: pair(s,r): pre(s) \wedge pre(r))$$

1. In the first term, if c_i=skip, it is assumed that $\neg end\_A_i \in \Gamma$.

2. In the second term, s or r may appear in a guard. We define pair(s,r) as s and r are a possible communication pair.

3. The pre-condition of "end_P" is defined to be post(P).

The hypothesis of (DL) is the set of states in which it is possible for S to synchronize at Γ. If this set is not empty, then it is required that some process be ready. Progress(Γ) says that either there is an s that is a repetitive statement that can terminate or a communication pair that can be executed.

7. Proof of freedom from deadlock

The assertions used for showing weak correctness are not sufficient to prove strong correctness. We assume initially that answered=0 and leave to the reader the proof that

$$\text{answered}=0 \vee (\forall j: j\neq\text{answered}: set_j=\emptyset)$$

is universally true.

Here again is the program, now with labels for the points that are included in the configurations of the program. The program has been annotated with additional assertions; they should be checked by the reader.

```
[  P:: P1 {answered=0} A?(m,answered) P2 {answered≠0}
|| A::[ (i:1..N) Min(i) A1 {answered≠0} ]]

Min(i)::
    my_min,sent:= a_i,false;
    M_i1 {answered≠i}
    do  (j:1..N ∧ i≠j) ¬sent; Min(j)!(my_min,set_i∪set_j,∅)
          → sent:= true

     □  (j:1..N ∧ i≠j) ¬sent; Min(j)?(their_min,set_i,set_j)
          → my_min:= min(my_min,their_min)
    od;

    if   sent → skip
     □  ¬sent → M_i2 {answered≠i} P!(my_min,i)
    fi
    M_i3 {(set_i=∅ ∨ answered=i) ∧ (∃j: set_j≠∅)}
```

We partition the set C([P||A]) and show that the system is deadlock free. Each section head lists conditions on the configuration Γ. Subheadings are additional constraints.

1. $(\exists i,j: i\neq j: M_i1, M_j1 \in \Gamma)$.

   Either $sent_i$ (or $sent_j$), in which case Min(i) (Min(j)) can progress; or $\neg sent_i \wedge \neg sent_j$, in which case Min(i) can send to Min(j). These cases correspond, respectively, to the first and second terms of progress(Γ).

2. $(\exists i: M_i1\in\Gamma: \forall j: i\neq j: \neg M_j1\in\Gamma))$.

If ($\forall j$: $j \neq i$: $M_j 3 \in \Gamma$), then the first term of progress is satisfied. If $M_j 2 \in \Gamma$, then ($set_i = \emptyset$)$\Leftrightarrow sent_i$ (from the loop invariant in Min(i)) and $set_i = \emptyset$ (implied by the pre-condition of $M_j 2$) are conjuncts in the antecedent of DL. Together they imply $sent_i$, which in turn implies the first term of progress(Γ).

3\. ($\forall i$: $\neg M_i 1 \in \Gamma$).

3.1 ($\exists j$: $M_j 2 \in \Gamma$).

3.1.1 $P1 \in \Gamma$.

$M_i 2$ sends to P1, ensuring progress. Note that pair($M_j 2$,P1) holds.

3.1.2 $P2 \in \Gamma$.

The pre-conditions answered$\neq$j and $set_j \neq \emptyset$, and the universal answered=0 $\vee$ ($\forall j$: $j \neq$answered: $set_j = \emptyset$), imply answered=0. The assertion at P2, answered$\neq$0, makes the antecedent of DL false.

3.2 $A1 \in \Gamma$.

3.2.1 $P1 \in \Gamma$.

The conjunction of the pre-conditions is false.

If there is no $M_i 1$ in Γ, then either some $M_i 2$ is in Γ or all the Min(i) are completed and $A1 \in \Gamma$. Similarly, {A1, P2} is not a configuration.

This completes the proof that this program is free from deadlock.

8. Conclusions

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 the result of communication is the same as the result of 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 (Chandy and Misra [CH], Hoare [HO2]), 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; these can be used to record as much or as little history as is needed.

The rule for the repetitive statement allows termination to occur because other processes are terminated. While this allows some interesting programs to be written, it might be better to require that the loop always terminate because of false guards. This makes the termination conditions explicit and simplifies both the proof rule and the implementation.

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 finessed by requiring all processes to terminate. How, then, do non-terminating processes fit into this system?

While preparing this paper, we heard of similar research being done by Apt, Francez, and de Roever [AP]. Their system has the same axioms for send and receive and a cooperation proof corresponding to our satisfaction proof. On the other hand, their cooperation proof derives from the forward assignment rule and they use a global invariant to relate auxiliary variables rather than allowing shared references to auxiliary variables. This global invariant is then used to eliminate those possible communication pairs that cannot synchronize.

9. Bibliography

[AP] Apt, K., Francez, N., and W. de Roever, A Proof System for Communicating Sequential Processes. TR RUU-CS-79-8, Department of Computer Science, University of Utrecht, 1979.

[CH] Chandy, K. and J. Misra, An Axiomatic Proof Technique for Networks of Communicating Processes. TR-98, Department of Computer Science, University of Texas at Austin, 1979.

[DI] Dijkstra, E. W., A Discipline of Programming. Prentice-Hall, New Jersey, 1976.

[HO] Hoare, C. A. R., Communicating Sequential Processes. CACM 21 (August 1978), 666-677.

[HO2] Hoare, C. A. R., Towards a Theory of Communicating Sequential Processes. Santa Cruz, August 1979.

[OW] Owicki, S. and D. Gries, An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica 6 (1976) 319-340.