A PROOF SYSTEM FOR A SUBSET OF THE CONCURRENCY SECTION OF ADA

(Preliminary Report)

Rob Gerth

RUU-CS-81-17

November 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

A PROOF SYSTEM FOR A SUBSET OF THE CONCURRENCY SECTION OF ADA
(Preliminary Report)

Rob Gerth

Technical Report RUU-CS-81-17
November 1981

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

A PROOF SYSTEM FOR A SUBSET OF THE CONCURRENCY SECTION OF ADA

(Preliminary Report)

Rob Gerth

Department of Computer Science, University of Utrecht

P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Keywords: ADA concurrency, rendezvous, cooperation test, partial correctness, proofsystem, soundness and completeness.

CR Categories: 5.24, 4.22.

1. Introduction.

This paper concerns the concurrency section of the programming language ADA [ARM81]. We study the basic ADA-synchronization primitive, the *rendezvous* and construct a proof system (for partial correctness properties) for a subset, ADA-CS, of the concurrency section.

As CSP [H78] is one of the languages which influenced the design of ADA, it should come as no surprise that the resulting proof system is similar to proof systems for CSP; especially to the one in [AFdeR80].

Within the latter system, the (CSP-)synchronization is captured by a *general invariant* and a *cooperation test*. The main result of this paper, is the formulation of the cooperation test for an ADA-rendezvous. Such a rendezvous is more complicated than a CSP-synchronization activity, which results in the transmission of a value between the two synchronized processes. An ADA-rendezvous is more properly characterized as something akin to a procedure-call and hence is related to process-communication as in Distributed Processes [BH78].

In this preliminary report, we first describe the subset, ADA-CS. Next, in section 3, a proof system for ADA-CS is constructed that closely follows the example of the CSP-proof system in [AFdeR80]. In the same section some of the semantic distinctions between ADA-CS and ADA are discussed. In section 4 the proof system is applied to prove a producer-consumer program in ADA-CS correct. An appendix contains the complete list of axioms and proof rules, of the proof system.

The full paper will contain the soundness and completeness proofs of the proof system: we show that each ADA-CS-program can be translated into an equivalent CSP-program and that our new cooperation test can be derived by applying the CSP-proof rules to the translation. (The CSP-proofsystem has been proven sound and complete in [A81a]).
For the exposition of the proofsystem, the reader is assumed to be familiar with the CSP-system in [AFdeR80].

2. The subset, ADA-CS.

The syntax of the subset of ADA is described by the following augmented BNF-grammar. The conventions used are similar to those of [ARM81]: *italicized* prefixes in the nonterminals are irrelevant, "[...]" denotes an optional part,"{...}" denotes repetition, zero or more times. Notice that we have taken some liberties with the ADA-syntax, which is somewhat verbose.

program ::= begin task {task} end

task ::= task *task*_id decl begin stats end

decl ::= {entry_decl}{var_decl}

entry_decl ::= entry *entry*_id (formal_part)

var_decl ::= var_id_list:int | var_id_list:bool

var_id_list ::= *var*_id{,*var*_id}

formal_part ::= [var_id_list][# var_id_list]

stats ::= stat{;stat}

```
stat ::= null | ass_st | if_st | while_st | call_st | acc_st | sel_st

ass_st ::= var_id := expr

if_st ::= if bool_expr then stats else stats end if

while_st ::= while bool_expr do stats end while

call_st ::= call task_id.entry_id (actual_part)

actual_part ::= {expr}[# var_id_list]

acc_st ::= accept entry_id (formal part) do stats end accept

sel_st ::= select sel_br {| sel_br} end select

sel_br ::= bool_expr : acc_st[;stats]

expr ::= "expression"

bool_expr ::= "boolean expression"

id ::= "identifier"
```

Thus an ADA-CS program consists of a fixed set of tasks. These tasks are all activated simultaneously. The declarations within each of the tasks are then elaborated, after which the statements constituting the task-body are executed. When execution reaches the end of the task-body, the task terminates. There are no global variables. Each task can have entry-declarations. Such an entry may be called by other tasks. The actions to be performed when such an entry is called, are specified by corresponding accept-statements.
Entry-call and accept-statement are the primary means of communication between tasks and synchronization of tasks: the *rendezvous*.
Within the body of a task, each of its entries can be named by the corresponding identifier. Outside the declaring task, the entry-name must also specify the task in which the entry is declared (see the syntax).

Apart from the synchronization, an entry-call executes as an ordinary procedure-call. An entry-declaration may specify a formal part. Only parameters of type int are allowed. The first set of parameters, closed off by the '#'-sign, is of mode in, the second set is of mode in out.

Hence, in the actual part of a corresponding call, the first set of actual parameters may be expressions, the second set must be variables.

An accept-statement specifies the actions to be performed at a call of the named entry. The formal part given in the accept-statement must match that given in the corresponding entry-declaration. A task may only contain accept-statements for one of its own entries, but it may contain more than one accept-statement for the same entry.

Execution of an accept-statement is synchronized with the execution of a corresponding entry-call. Consequently, a task executing an accept or entry-call-statement, will be suspended until another process reaches a corresponding entry-call or accept-statement. When this is the case, the statements of the accept-body are executed by the called task, while the calling task remains suspended. This action is called a *rendezvous*. Thereafter, the two tasks can continue their execution in parallel.

The select-statement allows a task to wait for synchronization with one of a set of alternatives. First, all boolean expressions are evaluated to determine which branches of the select-statement are open. If all are closed, the statement is equivalent with the null-statement (i.e., it does nothing). Otherwise, the task (if necessary) waits until a rendezvous corresponding with one of the open branches is possible. (Notice that each branch starts with an accept-statement.) If more than one rendezvous is possible, one is selected arbitrarily.

Notice that we have defined the ADA subset to be just large enough to study the rendezvous. If-, while- and select-statements are included only for the purpose of proving an example program correct.

3. The proof system.

A characteristic tendency in proof systems for concurrent languages is the reduction to sequential reasoning. Of course there is a price to pay for this: in [OG76] an *interference freedom test* has to be introduced and in [AFdeR80] a *cooperation test.*

In the CSP proof system, this reduction is brought about by the introduction of the two axioms for local reasoning about processes in isolation:

(1) $\{p\}\ P_i!a\ \{q\}$ and (2) $\{p\}\ P_j?x\ \{q\}$.

The actual test whether these pre and post-assertions are compatible with the communication action, is deferred to a second global stage: that of the cooperation test. To express this test, a *general invariant*, GI, is introduced to tie the local reasonings, within each of the processes, globally together. (In particular GI is used to distinguish among all potential communication-actions, i.e., the *syntactically* matching ones, the ones which may actually occur, i.e., the *semantically* matching ones.) Also, auxiliary variables have to be introduced to express the necessary assertions. As the variables appearing in GI have to be updated to model synchronization, GI cannot hold throughout the whole program. Hence the introduction of *bracketed sections* (each associated with a unique communication action) inside of which GI need not hold.

An ADA-rendezvous may be viewed as a double CSP-communication, as the following translation shows (1):

ADA (2):	CSP:
T_1: <u>call</u> entry(e # x)	$T_2!(e,x);\ T_2?x$
T_2: <u>accept</u> entry(u # v) <u>do</u> S <u>end</u> <u>accept</u>	$T_1?(u,v);\ S;\ T_1!v$

---

(1) For this translation, we have assumed a specific parameter-passing mechanism, which is not implied by the ADA-standard [ARM81]. This assumption will be discussed at the end of this section, together with the other assumptions to be made in sequel.
(2) If no confusion can arise, references to the task containin the entry are omitted.

Correspondingly in the proof system, the following axioms are adopted (1):

A3. *call* $\{p\}$ <u>call</u> entry(e # x) $\{q\}$

A4. *accept* $\{p\}$ <u>accept</u> entry(u # v) <u>do</u> S <u>end</u> <u>accept</u> $\{q\}$

A *general invariant*, GI, is introduced and also the concept of *bracketed sections*:

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

(1) S_1; <u>call</u> entry(e # x); S_2

(2) <u>accept</u> entry(u # v) <u>do</u> S_1

(3) S_1 <u>end</u> <u>accept</u>

where S_1 and S_2 do not contain any call or accept-statements and may be empty.

Notice that we cannot replace (2) and (3) together by a single clause like

(2') <u>accept</u> entry(u # v) <u>do</u> S_1; S; S_2 <u>end</u> <u>accept</u>.

The idea is that GI must hold when a communication-action is executed; that is, upon entry to and exit from the associated bracketed section. But as the "body", S, of an accept-statement may contain other call or accept-statements, the validity of GI would not have been assured at these points were clause (2') to be used in the above definition, since that would have implied that GI would not be required to hold in S.

---

(1) The numbering of the axioms and the proof rules refers to the complete list in the appendix.

Although a rendezvous is modelled by two CSP-like communications, it makes sense not to separate them entirely, because, disregarding the synchronization involved, a rendezvous is no more than an ordinary procedure call, which we want to treat as a meaningful whole. With this in mind, a cooperation test can be formulated:

Definition. Let $<S_1>$ and $<S_2>$ be a *communication pair*:

$<S_1> \equiv <S_1';$ <u>call</u> $\text{entry}_1(e \# x);\ S_1''>$

$<S_2> \equiv$ <u><accept</u> $\text{entry}_2(u \# v)$ <u>do</u> $S_2';> S <;S_2''$ <u>end</u> <u>accept></u>

($<S_1>$ and $<S_2>$ contained in different tasks).

We say that $<S_1>$ and $<S_2>$ *match* if 'entry_1' and 'entry_2' are the same name.

Definition. Consider (a proof of) an ADA-CS program <u>begin</u> task $T_1;\ldots;$ <u>task</u> T_n <u>end</u>. The proofs of $\{p_i\}\ T_i\ \{q_i\}$ $(1 \leq i \leq n)$ *cooperate* if

(1) the assertions used in the proof of $\{p_i\}\ T_i\ \{q_i\}$ have no free variables subject to change in T_j $(j \neq i)$,

(2) $\{\text{pre}(<S_1>) \wedge \text{pre}(<S_2>)\}\ <S_1> \parallel <S_2>\ \text{post}(<S_1>) \wedge \text{post}(<S_2>)$ holds for all matching communications pairs $<S_1>$ and $<S_2>$, within the program (1).

As is the case in the CSP-proof system, we need an additional rule to establish cooperation. This rule basically expresses the value-transfer at execution of a rendezvous. In CSP this is simple, as the execution of two matching io-commands, $P_i!a$ and $P_j?x$, is equivalent with executing the assignment $x := a$. In ADA however, a rendezvous results in a procedure call. In order to render the semantics of this procedure call simple, a few restrictions are placed on the actual

---

(1) Notice that, because of the one-side-naming in ADA we have more matching pairs to check than in CSP.

parameters of a call. (After all, we are not interested in the purely sequential aspects of ADA.) Hence:

For any entry-call $\underline{\text{call}}\ \text{entry}(e_1, e_2, \ldots, e_m \# x_1, x_2, \ldots, x_n)$

to an entry declared as $\underline{\text{entry}}\ \text{entry}(u_1, u_2, \ldots, u_m \# v_1, v_2, \ldots, v_n)$

the following three assumptions are made about the actual parameters, $\vec{e}$ and $\vec{x}$:

(1) the x_i are pairwise disjoint

(2) $\text{free}(\vec{e}) \cap \vec{x} = \emptyset$

(3) $(s \in \text{free}(\vec{e}) \cup \vec{x} \wedge s \notin \vec{u} \cup \vec{v}) \rightarrow s \notin \text{free}(\text{"bodies of entry"})$ (1)

Under these restrictions, the parameter-transfer at the execution of an entry call may be simulated by a substitution:

$$S[\vec{e} \,/\, \vec{u},\ \vec{x} \,/\, \vec{v}] \equiv \underline{\text{begin}}\ \underline{\text{new}}\ \vec{u}, \vec{v};\ \vec{u} := \vec{e};\ \vec{v} := \vec{x};\ S;\ \vec{x} := \vec{v}\ \underline{\text{end}}$$

The rule which is immediately suggested by this result (cfr. [A81b, 6.1.1]) however, is too simple, as we have also the bracketed sections to consider. Hence the following rule:

R10. *Formation*

$$\frac{\begin{array}{l}\{p_1 \wedge p_2 \wedge GI\}\ S_1';\ S_2'[\cdot]\ \{\bar{p}_1 \wedge \bar{p}_2[\cdot] \wedge GI\} \\ \{\bar{p}_2\}\ S\ \{\bar{q}_2\} \\ \{p_1 \wedge \bar{q}_2[\cdot] \wedge GI\}\ S_2''[\cdot];\ S_1''\ \{q_1 \wedge q_2 \wedge GI\}\end{array}}{\{p_1 \wedge p_2\} <S_1'; \underline{\text{call}}\ a(\vec{e} \# \vec{x}); S_1''> \| <\underline{\text{accept}}\ a(\vec{u} \# \vec{v})\ \underline{\text{do}}\ S_2'; > S <; S_2''\ \underline{\text{end}}\ \underline{\text{accept}}> \{q_1 \wedge q_2\}}$$

where (1) the call is contained in the task T_i and the accept in T_j $(i \neq j)$

(2) $[\cdot] = [\vec{e} \,/\, \vec{u},\ \vec{x} \,/\, \vec{v}]$

---

(1) With "bodies of entry" is meant, the bodies of the accept-statements for this particular entry.

(3) $\mathrm{free}(p_1, q_1) \subseteq \mathrm{free}(T_i)$

(4) $\mathrm{free}(\bar{p}_1) \subseteq \mathrm{free}(T_i) \setminus (\vec{x} \cup \mathrm{free}(\vec{e}))$

(5) $\mathrm{free}(p_2, \bar{p}_2, q_2, \bar{q}_2) \subseteq \mathrm{free}(T_j)$

In this rule, the first and third premiss (above the line) express the invariance checks of GI over the bracketed sections. Notice that the second premiss does not refer to the actual parameters. This means that a proof of the body, S, of the select-statement need only be given once and that this canonical proof suffices for the cooperation test for all matching communication pairs containing this select-statement. In the first premiss, we must show that the input is "legal" by deriving $\bar{p}_2[\cdot]$. If the input is legal, $\bar{q}_2[\cdot]$ specifies the output of the body S. The intermediate assertion, $\bar{p}_1$, is used to retain information about the variables of T_i which do not appear in the actual parameter list of the call. Such information cannot be placed in p_2, because $\bar{p}_2$ is an assertion belonging to T_j (and hence may not be subject to change in T_i).

After having formulated the new cooperation test, the (usual) parallel composition rule can be stated:

R9. *parallel composition*

$$\frac{\text{proofs of } \{p_i\}\ T_i\ \{q_i\},\ i=1\ldots n \text{ cooperate}}{\{p_1 \wedge \ldots \wedge p_n \wedge GI\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1 \ldots\ \underline{\text{task}}\ T_n\ \underline{\text{end}}\ \{q_1 \wedge \ldots \wedge q_n \wedge GI\}},$$

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

The last rule we will discuss, is the rule for the select-statement. This is in fact quite simple, once one remembers that possible waiting, in case no immediate rendezvous is possible, is not a partial correctness notion. Hence

R3. *select*

$$\frac{\{p \wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p \wedge b_n\}\ S_n\ \{q\}}{\{p\}\ \underline{\text{select}}\ b_1\ :\ S_1\ \mid\ \ldots\ \mid\ b_n\ :\ S_n\ \underline{\text{end}}\ \underline{\text{select}}\ \{q\}}$$

The other rules and axioms are the usual ones, for assignment, if, while-statements etc. A complete list of all axioms and proof rules can be found in the appendix.

The proof system, sketched above, is sound and (relatively) complete w.r.t partial correctness properties. This is intuitivily clear, given the close relationship with the CSP-proof system in [AFdeR80]. In fact we will prove this result (in the full paper) by an actual translation into CSP. It is interesting to see the rather smooth way in which procedures and concurrency have been combined here. The relevant rule, the formation rule, is a straightforward combination of the CSP-cooperation test and a simple procedure-call rule.

We conclude this section by discussing some of the differences between the ADA-CS semantics and the ADA semantics of the corresponding constructs.

Parameter passing.

We have explicitely assumed, that parameter-passing is done by 'call-by-value-result'. Moreover, additional constraints were imposed upon the actual parameters of a call. None of this is implied by the ADA-standard. However, the ADA-standard explicitly defines a program as being erroneous ([ARM81, 6.2])

(a) if it relies on a specific parameter-mechanism (i.e., call-by-reference or call-by-value-result in our subset) and

(b) if it "uses" aliasing.

Restrictions (1) and (3) on the actual parameters (p. 8), are precisely the conditions under which the two parameter-mechanism give the same results: (1) disallows aliasing between two parameters and (3) might be interpreted as disallowing aliasing between a parameter and a global variable (notice that within a proof system, we work with variable *names* not with locations). Otherwise, none of the restrictions limit the concurrency features of the language; only the formation rule is simplified.

Entry queues.

Contrary to the ADA-standard ([ARM81, 9.5]), which specifically states that entry-calls are processed strictly in order of arrival, we do not impose any order on the acceptance of entry-calls. These queues, are a somewhat difficult concept, because of racing conditions involved. For instance, the fact that there are no entry calls for a particular entry (i.e., the corresponding queue is empty) at a certain time, may just be a reflection of the difference in execution speed of the processors executing the tasks and hence is not a logical *necessity* of the state of the computation. A possible solution is to disregard this real time aspect completely and concentrate on those events which are a logical consequence of the computation (for instance, there is no rendezvous possible because all entry call statements are within the then-part of the if-statement whose condition is false). However, this is still a subject of further research. In any case, incorporating entry-queues will not alter the cooperation test.

Tasking and select errors.

In ADA, execution of a task is aborted in the following two situations:
(1) in case an entry of a terminated task is called,
(2) in case all boolean guards of a select-statement evaluate to false.
In ADA-CS, the task deadlocks in the first situation and ignores the select-statement in the second case. These decisions are not essential and are made to simplify the proof system.

4. Example.

In this section, we prove correct (a slightly adapted version of) the buffer-example in [ARM81, 9.12]. A producer-task generates values that are read in by a consumer-task. A buffer-task is used to smooth out the variations between the speed of the two other tasks. The buffer-task terminates when both the consumer and the producer have signalled that they are ready (terms = 2).
The text is as follows (we take the usual liberties with the data-types in our subset and the operations on them):

```
begin
   task producer
     vec1:array(1..n) of int;
       i:int
   begin
       i:=1;
       while i≠n+1 do
          call buffer.put(vec1(i));
          i:=i+1
       end while
       call buffer.term()
   end

   task consumer
     vec2:array(1..n) of int;
       j:=int
   begin
       j:=1
       while j≠n+1 do
          call buffer.get(#vec2(j));
          j:=j+1
       end while
       call buffer.term()
   end

   task buffer
     entry put(x)
     entry get(#x)
     entry term()
     pool:array(0..99) of int
     in,out,count,terms:int
   begin
     in:=0; out:=0; count:=0; terms:=0;
     while terms≠ 2 do
        select
          count<100:
            accept put(x) do
              pool(in mod 100):=x
            end accept;
            in:=in+1;
            count:=count+1
          |count>0:
            accept get(#x) do
              x:=pool(out mod 100)
            end accept;
            out:=out+1;
            count:=count-1
          |true:
            accept term() do null end accept;
            terms:=terms+1
        end select
     end while
   end
end
```

As a shorthand, we denote 'producer', 'consumer' and 'buffer' by 'T_1', 'T_2' and 'T_3', respectively.
We will prove that

$$\{n \geq 0\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1\ \underline{\text{task}}\ T_2\ \underline{\text{task}}\ T_3\ \underline{\text{end}}\ \{\forall i=1..n\ \text{vec1}(i)=\text{vec2}(i)\}$$

As is usual in proofs for concurrent programs, we will make use of a *proof-outline*, in which the program is given with the assertions interleaved at the appropriate places. The outline corresponding to the task T_i will be denoted by T_i'.

In the proof, we use the following auxiliary variables:

h_1 - recording in the producer, the sequence of values that has been sent off,

h_2 - recording in the consumer, the sequence of values that has been read in,

$\bar{h}_1$ - corresponds with h_1. records in the buffer, the values received,

$\bar{h}_2$ - corresponds with h_2. records in the buffer, the values removed.

The variables h_1, h_2, $\bar{h}_1$ and $\bar{h}_2$ are tuples. In the proof-outline, 'a^b' is used to denote the concatenation of the tuples 'a' and 'b' (or of the tuple 'a' and the element 'b'). In the assertions, arrays (or array-slices) will occasionally be used as tuples.

In the outline of the buffer task, the following invariant, I, for the while-loop is used:

$$\begin{aligned} I \equiv\ & \text{count}=\text{in}-\text{out} \wedge 0 \leq \text{count} \leq 100 \wedge \\ & \text{out} \leq \text{in} \supset h_1 = h_2\text{\^{}pool}(\text{out}\ \underline{\text{mod}}\ 100\ ..\ (\text{in}-1)\ \underline{\text{mod}}\ 100) \wedge \\ & \text{out} > \text{in} \supset h_1 = h_2\text{\^{}pool}(\text{out}\ \underline{\text{mod}}\ 100\ ..\ 99)\text{\^{}pool}(0\ ..(\text{in}-1)\ \underline{\text{mod}}\ 100) \end{aligned}$$

Finally

$GI \equiv h_1 = \bar{h}_1 \wedge h_2 = \bar{h}_2$

The proof-outline is as follows (note that the program text is already bracketed):

```
              begin
                task producer'
                  vec1:array(1..n) of int;
                    i:int; $h_1$:tuple of int
      {$h_1$=Λ} begin
                    i:=1;
{$h_1$=vec1(1..i-1)} while i≠n+1 do
    <$h_1$:=$h_1$^vec1(i); call buffer.put(vec1(i))>; {$h_1$=vec1(1..i)}
                       i:=i+1  {$h_1$=vec(1..i-1)}
                    end while {$h_1$=vec(1..n)}
                    <call buffer.term()>
                end {$h_1$=vec(1..n)}
                task consumer'
                  vec2:array(1..n) of int;
                    j:=int; $h_2$:tuple of int
      {$h_2$=Λ} begin
                    j:=1
{$h_2$=vec2(1..j-1)} while j≠n+1 do
                       <call buffer.get(#vec2(j)); $h_2$:=$h_2$^vec2(j)>{$h_2$=vec2(1..j)}
                       j:=j+1  {$h_2$=vec2(1..j-1)}
                    end while {$h_2$=vec2(1..n)}
                    <call buffer.term()>
                end {$h_2$=vec2(1..n)}
                task buffer'
                 entry put(x)
                 entry get(#x)
                 entry term()
                 pool:array(0..99) of int;
                 in, out, count, terms:int; $\bar{h}_1$, $\bar{h}_2$:tuple of int;
 {$\bar{h}_1$=Λ∧$\bar{h}_2$=Λ} begin
                 in:=0; out:=0; count:=0; terms:=0;
```

```
{I} while terms≠2 do
      select
        count<100: {I∧count<100}
         <accept put(x) do $\bar{h}_1:=\bar{h}_1$^x>
            {count=out-in∧0≤count<100∧
            (out≤in ⊃ $\bar{h}_1=\bar{h}_2$^pool(out mod 100..(in-1) mod 100)^x)∧
            (out>in ⊃ $\bar{h}_1=\bar{h}_2$^pool(out mod 100..99)^
                                            pool(0..(in-1) mod 100)^x)}

            pool(in mod 100):=x
            {count=out-in∧0≤count<100∧
            (out≤in ⊃ $\bar{h}_1=\bar{h}_2$^pool(out mod 100..in mod 100))∧
            (out in ⊃ $\bar{h}_1=\bar{h}_2$^pool(out mod 100..99)^
                                            pool(0..in mod 100))}
         <end accept>;
         in:=in+1;
         count:=count+1 {I}
        |count>0: {I∧count>0}
         <accept get(#x) do> {I∧count>0}
            x:=pool(out mod 100)
<$\bar{h}_2:=\bar{h}_2$^x; end accept>;
          {count=out-in∧0≤count<100∧
          (out≤in ⊃ $\bar{h}_1=\bar{h}_2$^pool((out+1) mod 100..(in-1) mod 100))∧
          (out>in ⊃ $\bar{h}_1=\bar{h}_2$^pool((out+1) mod 100..99)^
                                              pool(0..(in-1) mod 100))}
          out:=out+1;
          count:=count-1 {I}
        |true: {I}
         <accept term() do>null<end accept>;
         terms:=terms+1
     end select {I}
  end while {I∧terms=2}
 end
end {I}
```

Before the parallel-execution-rule can be applied, the proofs must be checked for cooperation.
We only prove cooperation for the 'put'-entry. The test for the 'get'-entry is analogous to the first one, while the test for the 'term'-entry is trivial.

There is only one matching communication pair to consider. Hence, by applying the formation-rule, the following formulas should be proved:

(1) $\{h_1=\text{vec1}(1..i-1)\wedge I\wedge \text{count}<100\wedge GI\}$

$h_1:=h_1\hat{}\,\text{vec1}(i);\ \bar{h}_1:=\bar{h}_1\hat{}\,\text{vec1}(i)$

$\{h_1=\text{vec1}(1..i)\wedge \text{count}=\text{out}-\text{in}\wedge 0\leq \text{count}<100\wedge$

$\text{out}\leq \text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..(\text{in}-1)\ \underline{\text{mod}}\ 100)\hat{}\,\text{vec1}(i)\wedge$

$\text{out}>\text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..99)\hat{}\,\text{pool}(0..(\text{in}-1)\ \underline{\text{mod}}\ 100)\hat{}\,\text{vec1}(i)\wedge GI\}$

(2) $\{\text{count}=\text{out}-\text{in}\wedge 0\leq \text{count}<100\wedge$

$(\text{out}\leq \text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..(\text{in}-1)\ \underline{\text{mod}}\ 100)\hat{}\,x)\wedge$

$(\text{out}>\text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..99)\hat{}\,\text{pool}(0..(\text{in}-1)\ \underline{\text{mod}}\ 100)\hat{}\,x)\}$

$\text{pool}(\text{in}\ \underline{\text{mod}}\ 100):=x$

$\{\text{count}=\text{out}-\text{in}\wedge 0\leq \text{count}<100\wedge(\text{out}\leq \text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..\text{in}\ \underline{\text{mod}}\ 100)\wedge$

$(\text{out}>\text{in}\supset \bar{h}_1=\bar{h}_2\hat{}\,\text{pool}(\text{out}\ \underline{\text{mod}}\ 100..99)\hat{}\,\text{pool}(0..\text{in}\ \underline{\text{mod}}\ 100))\}$

(3) $\{h_1=\text{vec1}(1..i)\wedge...\wedge GI\}$ $\underline{\text{null}}$ $\{h_1=\text{vec1}(1..i)\wedge...\wedge GI\}$,

where '...' denotes the post-assertion of clause (2).

Clearly, clause (3) is proven trivially and clause (2) follows by application of the assignment axiom. Clause (1) also follows by applying this axiom (twice); GI is invariant, because the same element (vec1(i)) is appended to both 'h_1', and '$\bar{h}_2$'.

Now we may apply the formation-rule and establish cooperation.

After having dealt with the other communication pairs in a similar way, the parallel execution rule may be applied to yield:

$$\{h_1 = \Lambda \wedge h_2 = \Lambda \wedge \bar{h}_1 = \Lambda \wedge \bar{h}_2 = \Lambda \wedge n \geqq 0 \wedge GI\}$$
$$\underline{\text{begin}}\ \underline{\text{task}}\ T_1\ \underline{\text{task}}\ T_2\ \underline{\text{task}}\ T_3\ \underline{\text{end}}$$
$$\{h_1 = vec1 \wedge h_2 = vec2 \wedge I \wedge GI\}$$

Remembering that vec1 and vec2 have the same length, we can apply the consequence2-rule (R12), to get the required post-assertion "vec1 = vec2".

With the auxiliary variable rule, all auxiliary variables are removed (note that the post-assertion does not contain h_i, $\bar{h}_i$ anymore).

Finally using the substitution2-rule (R11), the auxiliary variables in the pre-assertion are removed, thus completing the proof.

Acknowledgements.

I would like to thank Adrie van Bloois, Marly Roncken and Job Zwiers for some fruitful discussions and especially Willem P. de Roever for his stimulating guidance during the past year.

5. References.

[A81a] Apt, K.: Formal Justification of a proofsystem for Communicating Sequential Processes, to appear.

[A81b] Apt, K.: Ten Years of Hoare's Logic: A survey - Part 1, TOPLAS 3-4, p. 431-484, 1981.

[AFdeR80] Apt, K., N. Francez, W.P. de Roever: A Proof System for Communicating Sequential Processes. TOPLAS 2-3, p. 359-385, 1980.

[ARM81] The programming language ADA. Reference Manual. LNCS 106, Springer Verlag, New York 1981.

[BH78] Brinch Hansen, P.: Distributed Processes: A Concurrent Programming Concept. CACM 21-11, p. 934-941, 1978.

[H78] Hoare, C.A.R.: Communicating Sequential Processes. CACM 21-8, p. 666-677, 1978.

[OG76] Owicki, S., D. Gries: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf.6, p. 319-340, 1976.

Appendix.

For completeness sake, we provide a list of all axioms and rules of the proof system presented in this paper.

Axioms:

A1.	*assignment*	$\{p[e/x]\}x := e\ \{p\}$
A2.	*null*	$\{p\}\ \underline{\text{null}}\ \{p\}$
A3.	*call*	$\{p\}\ \underline{\text{call}}\ \text{entry}(\vec{e}\ \#\ \vec{x})\ \{q\}$
A4.	*accept*	$\{p\}\ \underline{\text{accept}}\ \text{entry}(\vec{u}\ \#\ \vec{v})\ \underline{\text{do}}\ S\ \underline{\text{end}}\ \underline{\text{accept}}\ \{q\}$

Rules:

R1.	*if*	$\dfrac{\{p \wedge b\}\ S_1\ \{q\}, \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{end}}\ \underline{\text{if}}\ \{q\}}$
R2.	*while*	$\dfrac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{end}}\ \underline{\text{while}}\ \{p \wedge \neg b\}}$
R3.	*select*	$\dfrac{\{p \wedge b_1\}\ S_1\ \{q\}, \ldots, \{p \wedge b_n\}\ S_n\ \{q\}}{\{p\}\ \underline{\text{select}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}\ \underline{\text{select}}\ \{q\}}$
R4.	*composition*	$\dfrac{\{p\}\ S_1\ \{q\}, \{q\}\ S_2\ \{r\}}{\{p\}\ S_1;\ S_2\ \{r\}}$
R5.	*consequence*	$\dfrac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$
R6.	*conjunction*	$\dfrac{\{p\}\ S\ \{q\}, \{p\}\ S\ \{r\}}{\{p\}\ S\ \{q \wedge r\}}$
R7.	*substitution*	$\dfrac{\{p\}\ S\ \{q\}}{\{p[t/z]\}\ S\ \{q\}}$ provided $z \notin \text{free}(S,q)$

R8. *auxvar*

$$\frac{\{p\}\ S'\ \{q\}}{\{p\}\ S\ \{q\}}$$ provided $\mathrm{free}(q) \cap AV = \emptyset$ and S is obtained from S' by deleting all assignments to variables of AV.

Here, AV is the set of (auxiliary) variables such that $x \in AV$ implies that x appears in S' only in assignments of the form $y := t$ with $y \in AV$.

R9. *parcom*

$$\frac{\text{proofs of } \{p_i\}\ T_i\ \{q_i\},\ i = 1..n \text{ cooperate}}{\{p_1 \wedge .. \wedge p_n \wedge GI\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1 \underline{..\text{task}}\ T_n\ \underline{\text{end}}\ \{q_1 \wedge .. \wedge q_n \wedge GI\}}$$

R10. *formation*

$$\frac{\begin{array}{c}\{p_1 \wedge p_2 \wedge GI\}\ S_1';\ S_2'[\cdot]\{\bar{p}_1 \wedge \bar{p}_2[\cdot] \wedge GI\} \\ \{\bar{p}_2\}\ S\ \{\bar{q}_2\} \\ \{p_1 \wedge \bar{q}_2[\cdot] \wedge GI\}\ S_2''[\cdot];\ S_1''\ \{q_1 \wedge q_2 \wedge GI\}\end{array}}{\{p_1 \wedge p_2\} <S_1> \| <S_2>\ \{q_1 \wedge q_2\}}$$

where (1) $<S_1> \equiv <S_1';\ \underline{\text{call}}\ a(\vec{e}\#\vec{x});S_1''>$ in T_i

(2) $<S_2> \equiv <\underline{\text{accept}}\ a(\vec{u}\#\vec{v})\ \underline{\text{do}}\ S_2';>S<;\ S_2''\ \underline{\text{end}}\ \underline{\text{accept}}>$

in $T_j\ (j \neq i)$

(3) $[\cdot] = [\vec{e}/\vec{u},\ \vec{x}/\vec{v}]$

(4) $\mathrm{free}(p_1, q_1) \subseteq \mathrm{free}(T_i)$

(5) $\mathrm{free}(p_1) \subseteq \mathrm{free}(T_i) \setminus (\vec{x} \cup \mathrm{free}(\vec{e}))$

(6) $\mathrm{free}(p_2, \bar{p}_2, q_2, \bar{q}_2) \subseteq \mathrm{free}(T_j)$

R11. *substitution2*

$$\frac{\{p\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1 \ldots\ \underline{\text{task}}\ T_n\ \underline{\text{end}}\ \{q\}}{\{p[t/z]\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1 \ldots\ \underline{\text{task}}\ T_n\ \underline{\text{end}}\ \{q\}}$$

provided $z \notin \mathrm{free}(T_1, \ldots, T_n, q)$

R12. *consequence2*

$$\frac{p \rightarrow p_1, \{p_1\}\underline{\text{begin}}\ \underline{\text{task}}\ T_1 \ldots\ \underline{\text{task}}\ T_n\ \underline{\text{end}}\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T_1 \ldots\ \underline{\text{task}}\ T_n\ \underline{\text{end}}\ \{q\}}$$