A PROOF SYSTEM FOR CONCURRENT ADA PROGRAMS

Rob Gerth

Willem P de Roever

Technical Report RUU-CS-83-2

Januari 1983

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030-53
The Netherlands

A PROOF SYSTEM FOR CONCURRENT ADA PROGRAMS

Rob Gerth
Willem P de Roever

Technical Report RUU-CS-83-2
Januari 1983

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

A PROOF SYSTEM FOR CONCURRENT ADA PROGRAMS

Rob Gerth Willem P de Roever
Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508TA Utrecht, the Netherlands.

0. ABSTRACT

A subset of ADA is introduced, ADA-CF, to study the basic synchronization primitive of ADA, the rendezvous. Starting with the CSP proof system of Apt, Francez and de Roever, we develop a Hoare-style proof system for proving partial correctness properties which is sound and relatively complete. The proof system is then extended to deal with safety, deadlock, termination and failure. Two non-trivial example proofs are given of ADA-CF programs; the first one concerns a buffered producer-consumer algorithm, the second one a parallel sorting algorithm due to Brinch Hansen.

Keywords: Ada-tasking, concurrency, rendezvous, Hoare-style proof system, partial correctness, total correctness, safety, termination, deadlock, failure, cooperating proofs, blocking, soundness, completeness.

CR-categories: C2.4, C3.1, E3.2.

The first author is supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).

1. INTRODUCTION

In this paper, we study the proof theory of the basic ADA synchronization primitive, the rendezvous. A subset of ADA, the ADA concurrency fragment with acronym ADA-CF, is defined for which a Hoare-style proof system is developed to prove partial correctness properties, which is sound and relatively complete as proven in [9]. The proof system is based on the CSP proof system in [3] which has as key-notion, the notion of cooperating proofs: Initially, proofs of the concurrently executing processes, or tasks as they are called in ADA, which constitute some program, are constructed in isolation. In such component proofs, assumptions are made about the behaviour of the other tasks communicating with the task whose proof is being constructed. To obtain a proof of the whole program, the proofs of its component tasks have to be combined. Consequently, the component proofs should cooperate in validating their assumptions about the behaviour of the others.

Technically, the main contribution of this paper is (1) the generalization of the idea of cooperation, as developed for CSP-type communication of transmitting simple values, to ADA-type communication which is akin to procedure calls and (2) formalizing the use of proof outlines to derive safety properties from. The generalization of cooperation has initially been developed in [16] and [8] in the context of a different language, namely Brinch Hansen´s Distributed Processes ([5]).

The rest of the paper is organized as follows: Section 2 introduces the subset ADA-CF and its (informal) semantics (for a more formal semantics, the reader is referred to [15] or [9]). In ADA-CF, only the bare essentials of ADA-tasking have been retained. Notably, the subset does not admit shared variables, access-variables to tasks (or any other object) task-creation and entry queues. This last restriction is not as serious as one might think it to be; see section 9 of this paper or [15]. Section 3 is the heart of the paper in which the partial correctness proof system is developed. Section 4 contains the first large(r) example proof of a program implementing a buffered producer-consumer system. In section 5 the proof system is extended to deal with safety-properties which generalize partial correctness properties. Notably, no new proof rules have to be introduced for this; instead, we show how to extract more information from the same proofs. This section also introduces the necessary terminology and techniques which are used in section 6 to deal with deadlock freedom and in section 7 to deal with termination and absence of failure (or clean termination). For these three properties, new proof rules and tests are needed. All this culminates in section 8 which contains the second large example proof. We consider a version of a linear time parallel sorting algorithm of Brinch Hansen ([5]) and prove it correct and deadlock and failure free. In fact, we prove that the algorithm can be used as a priority queue. Section 9, discusses some ADA constructs which can be added to our subset without much trouble. Notably, we show how to incorporate the terminate-statement of ADA, which introduces a distributed termination convention not unlike that of CSP [11]. Also, the absence of entry queues and some syntactic restrictions on the variables in ADA-CF are discussed.

2. THE SUBSET, ADA-CF

The syntax of ADA-CF is described, using a BNF-grammar augmented with the following embellishments (see also [1]):

(a) *italicized* prefixes in the nonterminals are irrelevant. I.e., '*var*_id' and '*entry*_id' are both equivalent to the nonterminal 'id'
(b) square brackets enclose optional items. I.e., the production 'decl ::= [entry_decl] [var_decl]' also produces the empty string
(c) braces enclose a repeated item, which can be repeated zero or more times. I.e., the production 'id_list ::= id {,id}' produces lists of one ore more id's.

The reader who is familiar with ADA, will notice that some liberties have been taken with the ADA syntax which is verbose at times.

```
program      ::= begin task {; task} end
task         ::= task task_id decl {label} begin stats end {label}
label        ::= label_id:
decl         ::= [entry_decl] [var_decl]
entry_decl   ::= entry entry_id_list;
var_decl     ::= int var_id_list;
id_list      ::= id {, id}
stats        ::= [label] stat {; [label] 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 endif
while_st     ::= while bool_expr do stats endwhile
call_st      ::= call task_id.entry_id( actual_part )
actual_part::= {expr} [#var_id_list]
acc-st       ::= accept entry_id( formal_part ) do stats endaccept
formal_part::= [var_id_list] [#var_id_list]
sel_st       ::= select sel_branch {or sel_branch} endselect
sel_branch ::= bool_expr: acc_st [; stats]
expr         ::= "expression"
bool_expr    ::= "boolean expression"
id           ::= "identifier"
```

There are some syntactical restrictions on the variables appearing in an ADA-CF program (if 'S' denotes an ADA-CF statement, then FV(S) denotes the set of its variables):

R1. for any two tasks T and T' in an ADA-CF program, $FV(T) \cap FV(T') = \emptyset$,
R2. within a task no name-clashes may occur either between the formal parameters, in the formal_parts of the task, themselves or between the formal variables and the global variables of the task,
R3. no formal in parameter may appear on the left-hand-side of any assignment or as in out parameter on any call,
R4. for any call-statement, call $T.a(e_1,\dots,e_n \# x_1,\dots,x_m)$
 (i) $x_1,\dots, x_m$ must be all distinct,
 (ii) $FV(e_1,\dots,e_n) \cap \{x_1,\dots,x_m\} = \emptyset$.

These restrictions will be discussed later on in sections 3 and 9.

Next, we give an informal description of the semantics.
An ADA-CF program consists of a fixed set of tasks. These tasks are all activated simultaneously and executed in parallel. When execution reaches the end of the task-body, this task terminates. Each task can have declarations of variables (all of type integer) and of entries, which may be 'called' by other tasks. The actions to be performed, when such an entry is called, are specified by matching accept-statements for this entry. Execution of an accept is

synchronized with the execution of a matching entry call. Consequently, a task executing an accept or entry call, will be suspended until another process reaches a matching entry call or accept, after which the statements of the accept-body are executed by the called task, while the calling task remains suspended. This action is called a **rendezvous** and is the only means of communication between and synchronization of tasks; in particular, there are no global (i.e., shared) variables. After a rendezvous, the two tasks engaged in this rendezvous continue their execution in parallel again. A program aborts (or fails) if (1) an entry is called of an already terminated task or (2) a task reaches the end of its body, while other tasks are still waiting for a rendezvous with this terminated task (the ADA reference manual ([1,§9.5]) is not quite clear what should happen in the second case, but failure is presumably what is intended).

Apart from the synchronization involved, the rendezvous-action is similar to an ordinary call for a procedure, having as body the body of the accept participating in the rendezvous. A task may only contain accepts for one of its own entries, but it may contain more than one accept for the same entry. Each accept specifies a formal_part for its entry; all accepts for the same entry should specify the same formal_part. The first set of parameters in such a formal_part, closed-off by the ´#´-sign, is of mode in (i.e., are value parameters); the second set is of mode in out (i.e., are initialized result parameters). Hence, in the actual_part of a matching call, the first set of actual parameters may be (integer) expressions, the second set must be variables. The parameters specified by an accept are local in scope w.r.t. the accept-body. Execution of a rendezvous between an entry call and a (matching) accept starts by assigning the values of all actuals to all formals. Then, the accept-body is executed after which the computed values of the formal result parameters are assigned to the actual result parameters.

The select-statement allows a task to wait for synchronization with one of a set of alternatives. First, all boolean expressions, ´guarding´ the branches of the select, are evaluated to determine which branches of the select are open (i.e., which expressions evaluate to true). If all are closed, the program aborts. 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.) In many cases, more than one rendezvous may be possible because several entries of a task may have been called or several tasks may have called the same entry. Similarly, several open branches may start with an accept for the same entry. In such cases, one of these alternatives is selected arbitrarily. In particular, this means that there are no entry or calling queues associated with entries as in ADA (*).

Finally, we assume that execution of an ADA-CF program can be modelled as an arbitrary interleaving of the actions of the component tasks. I.e., we assume an interleaving semantics (INT) and, in this respect we do not distinguish ourselves among other researchers in the proof theory of concurrent programs. However, this is not the only possibility and one could also assume **maximal** paralellism semantics (MAX). In such a semantics, component tasks execute truly in parallel whenever this is possible; in particular, execution of a task is never unnecessarily suspended. Both types of semantics are reasonable: MAX corresponds to a situation in which the component tasks execute on identical dedicated processors; INT corresponds to a situation in which this is not the

----

(*) That the presence or absence of entry queues has no influence on the semantics of our subset is proved in [15]; see also section 9.

case or in which time-sharing occurs. The reader is referred to [17] for a more formal exposition. While our proof system is sound under both INT and MAX, it is complete under INT only (see [17] for an example).

Example 2.0. This example illustrates the ADA-CF susbset and the liberal way in which ADA-CF is augmented with extra data types when this is deemed necessary to code non-trivial example programs. The proof system will be developed for the integer data type only, though.

The program is a straightforward solution of a producer-consumer problem with a buffer in between to smooth out speed-variations and is a slightly adapted version of the solution in [1,§9.12] (´n´ denotes an arbitrary positive integer constant):

```
begin
   task producer
      array (1..n) of int vec1; int i;
   begin i := 1 -- and initialize vec1 to some arbitrary values
      while i ≠ n+1 do
         call buffer.put( vec1(i) ); i := i+1
      endwhile;
      call buffer.term()
   end

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

   task buffer
      entry put, get, term;
      array (0..99) of int pool; int in, out, count, terms;
   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 endaccept;
         or count > 0 :
               accept get(#y) do y := pool(out mod 100) endaccept;
         or true :
               accept term() do null endaccept;
               terms := terms+1
         endselect
      endwhile
   end
end
```

The extra entry ´term´ and variable ´terms´ in the buffer-task, are needed to determine when buffer may terminate (terms=2). Remember that ADA-CF does not have the ADA terminate-statement. □

3. THE PROOFSYSTEM

The proof system is similarly structured as the one of Owicki in [14] or the CSP-system of Apt et al [3]: In order to prove a property about a program, one first constructs separate proofs for the component tasks in isolation and then combines these component proofs to obtain a proof of this property. In general the component tasks will influence each other. Consequently, within a component proof one has to make assumptions about the behaviour of the environment of the task. Therefore, if these component proofs are to be combined, these assumptions should be consistent and must be checked. This explains the need for tests such as the interference freedom test of [14] and the cooperation test of [3]. Because of the close relationship between ADA-CF and CSP communication, the consistency test on component proofs of the ADA-CF system will be based on the CSP cooperation test. Such tests introduce a meta-element in Hoare-style proof systems, because they refer to properties of proofs. The natural notion of proof for which such tests can (formally) be defined is that of proof outlines; first introduced by Owicki for her language GPL in [14] and subsequently used for CSP in [3]. In the case of GPL and CSP, it is a rather trivial problem which consistency tests have to be imposed upon the proof outlines (of course, the specific form such a test takes may be less trivial to find). In the case of ADA-CF, the reader will see that there is a subtle problem involved in this choice.

To ´separate´ the component proofs from each other, the following axiom and proof rule are adopted:

A1. call: $\{p\}$ <u>call</u> T.e $(\vec{t}\#\vec{x})$ $\{q\}$,

provided $FV(p) \cap \{\vec{x}\} = \emptyset$.
$\vec{t}$ and $\vec{x}$ denote respectively the value expression list and the value result variable list; the domain of FV has been extended so as to yield the set of free variables of its argument assertion(s). This axiom expresses that in a component proof, anything may be assumed about the result of an entry call. Of course such an assumption must be checked later on. The restriction on the free variables of the pre-assertion will be discussed later.

R1. accept:

$$\frac{\{p'\}\ S\ \{q'\}}{\{p\}\ \underline{\text{accept}}\ e\ (\vec{u}\#\vec{v})\ \underline{\text{do}}\ S\ \underline{\text{endaccept}}\ \{q\}},$$

provided $\{\vec{u},\vec{v}\} \cap FV(p,q) = \emptyset$.
First of all, the rule forces a proof of the accept-body to be given. However, it does not enforce relationships between the pre and post-assertion of the body and the pre and post-assertion of the accept. This is reasonable as p´ and q´ must say something about the values of the formal parameters, which are (partly) determined by the environment. Consequently, these assertions have to be checked too, later on. The formal parameters are local w.r.t. the accept-body, whence the restriction on the variables free in p and q.

These are augmented by the following rules and axioms:

A2. null: $\{p\}$ null $\{q\}$.

A3. assignment: $\{p[t/x]\}\ x := t\ \{p\}$,

where $[t/x]$ denotes the usual substitution of the expression t for each (free) occurrence of x in p.

R2. select:

$$\frac{\{p \wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p \wedge b_n\}\ S_n\ \{q\}}{\{p\}\ \underline{\text{select}}\ b_1 :\ S_1\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ b_n :\ S_n\ \underline{\text{endselect}}\ \{q\}}\ .$$

Remember that waiting (until a rendezvous is possible) is not a partial correctness property.

R3. if:

$$\frac{\{p \wedge b\}\ S\ \{q\}\ ,\ \{p \wedge \neg b\}\ S'\ \{q\}}{\{p\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ S\ \underline{\text{else}}\ S'\ \underline{\text{endif}}\ \{q\}}\ .$$

R4. while:

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{endwhile}}\ \{p \wedge \neg b\}}\ .$$

R5. composition:

$$\frac{\{p\}\ S\ \{q\}\ ,\ \{q\}\ S'\ \{r\}}{\{p\}\ S;S'\ \{r\}}\ .$$

R6. consequence:

$$\frac{p \rightarrow p'\ ,\ \{p'\}\ S\ \{q'\}\ ,\ q' \rightarrow q}{\{p\}\ S\ \{q\}}\ .$$

R7. body:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \underline{\text{begin}}\ S\ \underline{\text{end}}\ \{q\}}\ .$$

In the sequel, a task will often be identified with its body, in the sense that $\{p\}$ task T $\{q\}$ or $\{p\}\ T\ \{q\}$ will be written where $\{p\}$ begin S end $\{q\}$ (being the body of task T) is meant.

Using these rules, properties about tasks (or task-bodies) in isolation, can be proved. Such proofs can be given an alternative form by annotating the task-body with the assertions generated by its proof; i.e., each sub-statement S of the task-body can be annotated with the assertions used in the application of one of the above rules or axioms to S. It is straightforward to make this precise:

Definition 3.0. A proof outline for an ADA-CF task (-body) S, associates with each sub-statement R of S (and with S itself) a unique pre-assertion, pre(R), and a unique post-assertion, post(R), and defines a bracketing for the task (*). Such a proof outline is called valid for a formula $\{p\}\ S\ \{q\}$ precisely if for each sub-statement R of S, the following verification conditions hold:

(*) The notion of bracketing will be explained later on. Until then, the reader may safely ignore this requirement.

(1) $p \rightarrow pre(S)$ and $post(S) \rightarrow q$,
(2) $pre(S) \rightarrow pre(R)$ and $post(R) \rightarrow post(S)$ if $S \equiv$ <u>begin</u> R <u>end</u>,
(3) $pre(R) \rightarrow post(R)$ if $R \equiv$ <u>null</u> ,
(4) $pre(R) \rightarrow post(R)[t/x]$ if $R \equiv x := t$,
(5) $pre(R) \wedge b \rightarrow pre(R')$, $pre(R) \wedge \neg b \rightarrow pre(R'')$, $post(R') \rightarrow post(R)$ and $post(R'') \rightarrow post(R)$ if $R \equiv$ <u>if</u> b <u>then</u> R´ <u>else</u> R" <u>endif</u> ,
(6) $pre(R) \wedge b \rightarrow pre(R')$, $post(R') \rightarrow pre(R)$ and $pre(R) \wedge \neg b \rightarrow post(R)$ if $R \equiv$ <u>while</u> b <u>do</u> R´ <u>endwhile</u> ,
(7) $pre(R) \wedge b_i \rightarrow pre(R_i)$ and $post(R_i) \rightarrow post(R)$ for i=1..n if $R \equiv$ <u>select</u> b_1: R_1 <u>or</u> ... <u>or</u> b_n: R_n <u>endselect</u> ,
(8) $FV(pre(R),post(R)) \cap \{\vec{u},\vec{v}\} = \emptyset$ if $R \equiv$ <u>accept</u> e $(\vec{u}\#\vec{v})$ <u>do</u> R´ <u>endaccept</u> ,
(9) $FV(pre(R)) \cap \{\vec{x}\} = \emptyset$ if $R \equiv$ <u>call</u> T.e $(\vec{t}\#\vec{x})$,
(10) $pre(R) \rightarrow pre(R')$, $post(R') \rightarrow pre(R'')$ and $post(R'') \rightarrow post(R)$ if $R \equiv R';R''$. □

Such proof outlines correspond with the purely sequential part of an ordinary proof. It is easy to see that a proof outline is valid for a formula {p} S {q}, precisely when its pre and post-assertions can be used in an ordinary proof for {p} S {q}: The conditions (1)...(10) restrict the assertions to those that can be obtained by using one of the proof rules or axioms given.

Apparently, with proof outlines a special kind of proof corresponds; namely proofs in which no two applications of a proof rule or axiom refer to the same statement; otherwise the pre and post-assertions of this statement would not be unique. We will return to this fact later on.

Subsequent discussions will always refer to proofs in this form; an example will shortly follow.

In the proof outline of a component task T, assumptions are made about the behaviour of the tasks, T communicates with. To be more specific, T makes assumptions about the values it receives, both for the value-result parameters on termination of an entry call and for the formal parameters, when T enters an accept. Using these assumptions the proof outline for T specifies in a sense the behaviour to which T commits itself; i.e., the appropriate pre-assertions specify the values sent off to a task which becomes engaged in a rendezvous with T. In essence, the consistency test must show that the behaviour of each task satisfies the assumptions concerning its behaviour, made by the task communicating with it. This discussion makes the following more formal statement of the cooperation test plausible:

First formulation of cooperation of ADA-CF proofs.
The proof outlines of $\{p_1\}$ <u>task</u> T_1 $\{q_1\}$, ..., $\{p_n\}$ <u>task</u> T_n $\{q_n\}$ cooperate if

(1) for any ´matching communication pair´
    $C \equiv$ <u>call</u> T_j.e $(\vec{t}\#\vec{x})$ and
    $A \equiv$ <u>accept</u> e $(\vec{u}\#\vec{v})$ <u>do</u> S <u>endaccept</u> (A within T_j),
    the formula $\{pre(C) \wedge pre(A)\}$ $C \| A$ $\{post(C) \wedge post(A)\}$ holds, whenever C and A become engaged in a rendezvous. ("$C \| A$" denotes the execution of this rendezvous).
(2) the assertions of the proof outlines of $\{p_i\}$ T_i $\{q_i\}$ (i=1..n) have no free variables subject to change in any T_j ($j \neq i$). I.e., have no free variables which apear free on the left-hand-side of any assignment in T_j or as value-result parameter of any entry call in T_j. □

The first clause is clear enough, asking to derive the post-assertions of the entry call and accept, if a rendezvous between these two occurs, (necessarily) in a state obeying the two pre-assertions. The discussion what formulae like {p}

$C \| A$ $\{q\}$ precisely denote and how they are proved, is deferred for a while.

The second clause forces the independence of the proof outlines: No proof outline may ´talk´ about variables of other tasks; hence, a proof outline cannot be invalidated by actions elsewhere. However, this restriction does not apply to variables that are not changed in the program. Such so called freeze variables are needed to prove relations between variables of different tasks. As a consequence, only post-assertions of entry calls and the pre-assertions of accept-bodies make assumptions about the behaviour of other tasks (so that only these assertions have to be checked). This is reasonable because at these places only, outside information is injected into a task.

Example 3.1. Consider the following ADA-CF program:

```
begin task T int x; begin call T´.a (x) end
      task T´ entry a; int y; begin accept a (u) do y := u endaccept end
end
```

Clearly, $\{true\}$ begin task T; task T´ end $\{x=y\}$. (3.2)

To prove this, introduce a freeze variable ´z´ and proof outlines

```
      task T int x;                          task T´ entry a; int y
{x=z} begin                           {true} begin
   {x=z} call T´.a(x) {x=z}              {true} accept a(u) do
      end {x=z}                              {u=z} y := u {y=z}
                                                 endaccept {y=z}
                                             end {y=z}
```

It is easy to see that the proof outlines are valid. Do they cooperate too? Well, clause 2 clearly holds; a little thought makes satisfaction of the first clause plausible too. As the proof rule for such formulae has not yet been given, the actual ´proof´ can only be rendered in the following form: Clause 1 asks for the proof of $\{x=z \wedge true\}$ $C \| A$ $\{x=z \wedge y=z\}$ (C denotes the entry call in T, A the accept in T´ and $C \| A$ the actual rendezvous). According to the semantics of a rendezvous (cf. section 2), C A is (roughly) equivalent with

$u:=x;\ y:=u$

($u:=x$ is the assignment of the actual to the formal parameters). Consequently, one has to show that the formula

$\{x=z\}\ u:=x\ \{u=z\}\ y:=u\ \{x=z \wedge y=z\}$

can be ´completed´ so as to yield a valid proof outline. In particular, the intermediate assertion, u=z, should be retained, as this assertion embodies the assumption of T´ about T´s behaviour. To show this, is rather trivial; the following proof outline is the required completion (*):

$\{x=z\}\ \{x=z \wedge x=x\}\ u:=x\ \{x=z \wedge u=x\}$

$\{x=z \wedge u=z \wedge u=u\}\ y:=u\ \{x=z \wedge u=z \wedge y=u\}\ \{x=z \wedge y=z\}.$

So the outlines may be combined:

$\{x=z \wedge true\}$ begin task T; task T´ end $\{x=z \wedge y=z\}$.

Application of the consequence rule yields

$\{x=z\}$ begin task T; task T´ end $\{x=y\}$.

As the value of z and hence of x has not been specified, (3.2) holds too. □

(*) Juxtaposition of two assertions in a proof outline denotes implication of the rightmost assertion by the leftmost

The last deduction in the example was not formalized and indicates another missing proof rule:

R8. substitution:

$$\frac{\{p\}\ S\ \{q\}}{\{p[t/z]\}\ S\ \{q\}}\ ,\ \text{provided } z \notin FV(S,q).$$

Using this rule, the last step in the example proof can be formalized using the substitution [x/z] (and then applying the consequence rule).

The above simple-minded approach to cooperation is too weak in general: The first clause of its definition requires one to prove a formula involving an entry call C and an accept A, but only if C and A can actually become engaged in a rendezvous. This now, cannot be inferred from the program text alone but has to be semantically characterized. Consequently, it raizes the dual problem of characterizing the absence of a rendezvous.

A rendezvous between A and C can only occur if there is a computation of the program, reaching simultaneously both A and C. Now, in a (valid) proof outline, the pre-assertion of some statement, by definition, characterizes all computations reaching this statement. Consequently, a rendezvous between A and C cannot occur, whenever pre(C) $\wedge$ pre(A) $\rightarrow$ false.

The following example and subsequent discussion addresses the question whether assertions can be made strong enough to express the impossibility of a particular rendezvous.

Example 3.3.

```
begin
   task T begin call T´.a(1); call T".b() end
   task T´ entry a; int x;
      begin
         accept a(#u) do x := u endaccept;
         accept a(#v) do x := v endaccept;
      end
   task T" entry b;
      begin accept b() do null endaccept; call T´.a(2) end
end
```

The formula

{true} begin task T; task T´; task T" end {x=2}

clearly holds. In order to prove it, the post-assertion of the second accept in a proof outline of T´ necessarily must imply x=2. If this post-assertion is to pass the cooperation test, the conjunction of the pre-assertion, p, of the first entry call in T with the pre-assertion, q, of the second accept in T´ must somehow yield false, expressing that this rendezvous will not take place during execution, thus trivializing the cooperation test for this pair. Consequently, q must express something like "if T´ is at the second accept then T must be after its first entry call". But precisely this type of assertion is ruled out by the second clause of the cooperation test! Besides, there is the moot point how to express such conditions at all. □

In other words, this example suggests the proof system to be (still) incomplete in the sense that not every operationally true property can be proved. It illustrates the difference between a syntactically matching pair - such as the call in T" and the first accept in T´ -, and a semantically matching pair - such as the call in T" and the second accept in T´.

To determine which of the syntactic matches also match semantically, the example suggests that it needs relating states of different tasks to each other. For this purpose, the proof system is augmented with a global invariant, GI, which may also carry other global information needed for a proof. GI expresses in general which rendezvous´ occurred and which values were sent and received during these rendezvous´; in short, it expresses (or encodes) the communication-history. As is well-known, to express relations between the states of different tasks in general, either the state of each task has to be explicitely extended with a location counter or the tasks have to be extended by statements involving fresh, so called auxiliary variables. For this proof system the latter option is chosen, as in [3] and [14]. For ADA-CF it is an open problem whether location counters can be used, too. Presumably it is possible because for CSP, the question has an affirmative answer, as proved in [12].

For example, if the tasks T and T´ in example 3.3 are augmented with auxiliary variables i and j respectively (both initialized to 0), the fact that T has executed its first call can be encoded in i by inserting the assignment i:=1 between the two calls. Likewise, to encode that T´ has executed its first accept, an assignment j:=1 can be inserted between the two accepts in task T´. Then the pre-assertion, p, of the first call in T can be chosen so as to imply j=0 (j is initialized to 0); The pre-assertion, q, of the second accept in T´, can be chosen so as to imply j=1. A global invariant, $I \equiv j=1 \rightarrow i=1$, would express the property "if T´ is after its first accept (j=1) then T must be after its first call (i=1)". Consequently, if control would be at the first call in T and simultaneously at the second accept in T´, the state would obey $p \wedge q \wedge I$ (I is assumed to be globally invariant), which implies $i=0 \wedge j=1 \wedge (j=1 \rightarrow i=1)$, which is equivalent to false; this shows that this situation can in fact not occur during execution and it trivializes the cooperation test for this matching pair.

Unfortunately, I is not a global invariant for the program because of the problem of updating its free variables. Since the assignments to i and j need not (and in general will not) be executed simultaneously, I can be invalidated. To resolve this problem, the range over which a general invariant, GI, must hold is restricted as in [3] by introducing a bracketing for each program; the updatings of GI-variables are then confined to bracketed sections (in which GI consequently need not hold) which are associated with entry calls and accepts as these are the only statements at which the execution of differen tasks synchronize:

First definition of bracketing:
A task is called bracketed if the brackets ´<´ and ´>´ are interspersed in its text, so that

(1) for each bracketed section <S>, S is of the form
 S´ ;<u>call</u> T.a (e#x); S" or <u>accept</u> b (u#v) <u>do</u> S´; S; S" <u>endaccept</u>
 (where S´ and S" update the variables of the global invariant and may be <u>null</u>-statements),

(2) each call and accept is contained in a bracketed section. □

Introduction of a global invariant (and associated bracketing) enables a reformulation of the cooperation test in order to refine the notion of matching:

Second formulation of cooperating ADA-CF proofs:
The proof outlines of $\{p_i\}$ <u>task</u> T_i $\{q_i\}$ (i=1..n) cooperate w.r.t. GI if

(1) for any syntactically matching pair <C> and <A>, where
 $C \equiv S_1$; <u>call</u> $T_j.e$ $(\vec{t}\#\vec{x})$; S_2 and $A \equiv$ <u>accept</u> e $(\vec{u}\#\vec{v})$ <u>do</u> S_1'; S; S_2' <u>endaccept</u>

(A within T_j), the formula
$\{pre(C) \wedge pre(A) \wedge GI\}$ C ∥A $\{post(C) \wedge post(A) \wedge GI\}$ holds
(2) the assertions of the proof outlines of $\{p_i\}$ T_i $\{q_i\}$ (i=1..n) have no free variables subject to change in any T_j ($j \neq i$). □

Notice that confining the updating of GI-variables to bracketed sections only, implies that to ensure invariance of GI, only bracketed sections have to be checked and that GI may be assumed to hold when entering such a section (provided GI holds initially). This suggests the following parallel composition meta rule:

R9: parcom:

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

provided no variable free in GI is updated outside a bracketed section and GI does not contain formal or actual parameters as free variables.

The reason not to allow formal or actual parameters to appear free in GI, is to prevent some additional complications to occur: Allowing actual parameters, would introduce an aliasing problem, as a variable of GI then could be updated under a different name. Allowing formal parameters, would complicate the rendezvous rule, R11, to be developed below. In general, problems will arise if proofs are combined (using this rule) of tasks which share variable-names, because of possible name-clashes in the consequent of the rule. This motivates restriction R1 of section 2, which restricts the subset to programs in which no name-sharing occurs.

Finally, a rule is needed to remove auxiliary variables from a program again (this rule is similar to the ones in [3] and in [14]):

R10. AV: Let AVAR denote a set of variables such that $x \in AVAR \Rightarrow x$ appears in S´ only in assignments of the form y:=x with $y \in AVAR$. Then

$$\frac{\{p\}\ S'\ \{q\}}{\{p\}\ S\ \{q\}},$$

provided $FV(q) \cap AVAR = \emptyset$ and S is obtained from S´ by deleting assignments and declarations to variables in AVAR.

Example 3.4. Now, the formula

{true} begin task T; task T´; task T" end {x=z} (3.5)

of example 3.3 can be verified, indeed. To express the necessary assertions, three auxiliary variables, i, j and k, are introduced into the proof outlines of T, T´ and T", respectively. The proof outlines will be less detailed than in the previous example, but the reader will have no difficulty to fill in the missing details.

```
      task T int i;                        task T" entry b; int k;
{i=0} begin                          {k=0} begin
   {i=0} <call T´.a(1); i:=1>;          {k=0} <accept b() do k:=1; null endaccept>;
```

```
{i=1} <call T".b()>                {k=1} <call T´.a(2)>
   end {i=1}                          end {k=1}

          task T´ entry a; int x,j;
     {j=0} begin
        {j=0} <accept a(#u) do {j=0 ∧ u=1} x:=u; j:=1 endaccept>;
   {j=1 ∧ x=1} <accept a(#v) do {j=1 ∧ x=1 ∧ u=2} x:=u endaccept>
          end {j=1 ∧ x=2}
```

In this proof, the following global invariant is used:

$GI \equiv (j=1 \leftrightarrow i=1) \wedge (k=1 \rightarrow j=1)$.

Now clearly, the individual proof outlines are correct and the second clause of the cooperation test holds too. As for clause 1 , first consider the not semantically matching pairs: The first call in T with the second accept in T", and the call in T´ with the first accept in T´. It is easy to see that the conjunction of the pre-assertions with GI $-i=0 \wedge j=1 \wedge x=1 \wedge (j=1 \leftrightarrow i=1) \wedge (k=1 \rightarrow j=1)$ and $k=1 \wedge j=0 \wedge (j=1 \leftrightarrow i=1) \wedge (k=1 \rightarrow j=1)$ respectively - both yield false. Next the semantic matches:

(1) The first call in T with the first accept in T´.
 The formula

$$\{i=0 \wedge j=0 \wedge GI\}\ u:=1\ \{j=0 \wedge u=1\}\ x:=u;\ j:=1;\ i:=1\ \{i=1 \wedge j=1 \wedge x=1 \wedge GI\}$$

 should be completed. This is trivial:

$$\{i=0 \wedge j=0 \wedge GI\}\ u:=1\ \{j=0 \wedge u=1\}\ x:=u\ \{x=1 \wedge j=0\}$$
$$j:=1\ \{x=1 \wedge j=1\}\ i:=1\ \{x=1 \wedge i=1 \wedge j=1\}\ \{x=1 \wedge i=1 \wedge GI\}$$

(2) The second call in T with the accept in T".
 To complete $\{i=1 \wedge k=0 \wedge GI\}\ k:=1;\ \underline{\text{null}}\ \{i=1 \wedge k=1 \wedge GI\}$, is even more trivial:
 $\{i=1 \wedge k=0 \wedge GI\}\ \{i=1 \wedge k=0 \wedge GI\}\ k:=1\ \{i=1 \wedge k=1 \wedge j=1\}\ \underline{\text{null}}\ \{i=1 \wedge k=1 \wedge GI\}$.
 Notice that here, the implication $i=1 \rightarrow j=1$, part of GI, is needed; otherwise the second part of GI - $k=1 \rightarrow j=1$ - cannot be derived.

(3) The call in T" with the second accept in T´.
 This is left to the reader.

Application of R9, the parallel composition rule, yields

$$\{i=0 \wedge j=0 \wedge k=0 \wedge GI\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T;\ \underline{\text{task}}\ T';\ \underline{\text{task}}\ T''\ \underline{\text{end}}$$
$$\{i=1 \wedge j=1 \wedge x=2 \wedge k=1 \wedge GI\}.$$

Using the consequence rule to get the post-assertion $x=2$ and the AV-rule, which may be applied now, to remove the auxiliary variables, the formula reduces to

$$\{i=0 \wedge j=0 \wedge k=0 \wedge GI\}\ \underline{\text{begin}}\ \underline{\text{task}}\ T;\ \underline{\text{task}}\ T';\ \underline{\text{task}}\ T''\ \underline{\text{end}}\ \{x=2\}.$$

Now, the substitution rule can be used to substitute 0 for i, j and k in the pre-assertion. Formula (3.5) is obtained by reducing the pre-assertion to true with a final application of the consequence rule. It should be remarked here that, although in this example, GI only relates locations in different tasks with each other, in general GI also caries other state information; see for instance the example proof in section 5. □

The above development mirrors the development of the CSP-system in [3]. In fact, all of the above examples and problems have their counterpart in that proof system. However, the construction of a proof system for ADA-CF also introduces problems which are particular for that language, and it is to these problems that the rest of this section addresses itself. They result from the possibility of having occurrences of calls or accepts within the body of another accept; such a nesting of communication statements is not possible in CSP. This will enforce a refinement of the notion of bracketing. It has also consequences for the formulation of the final - still missing - proof rule to derive the {p}

C || A {q}-type formulae of the cooperation test.

First an example will show that, although introducing bracketings (and global invariants and auxiliary variables) has made the proof system (seemingly) complete, at the same time it has made it unsound!

Example 3.6. Consider the following proof outlines (h is an auxiliary variable):

```
       task T                          task T´ entry a; int h;
{true} begin                    {h=0} begin
          <call T´.a()>;            {h=0} <accept a() do h:=1; {h=1}
       end {true}                                <call T".b(1)>; h:=0
                                              endaccept> {h=0}
       task T" entry b; int y;            end {true}
{true} begin
          <accept b(x) do
     {x=0} y:=x endaccept>              GI ≡ h=0
       end {y=0}
```

The individual proof outlines are correct and if they are combined, they ´prove´

{true} begin task T; task T´; task T" end {y=0}.

However, the reader easily sees that after termination, y=1 holds. The problem is of course the assumption of T", x=0, (which follows from rule R1) about the value it receives from T´. This assertion should not pass the cooperation test for the accept in T" with the entry call in T´. Unfortunately it does, and vacuously so, as the conjunction of the respective pre-assertions with GI yields false: $h=1 \wedge true \wedge h=0$. The test for the other matching pair holds too (this time rightly so). Hence, the outlines can be combined and the proof system allows ´proofs´ of invalid formulae and is consequently not sound. □

Analyzing the example, shows this disparity to be caused by the nested occurrence of the entry call within the body of the accept in T´, because GI should also hold when such inner calls or accepts are reached. As these appear within the bracketed section of the outer accept in which GI need not hold, indeed cannot hold as its variables are being updated, this means that the range of the bracketed sections is too large and must somehow be restricted so as to contain precisely one call or accept each.

GI encodes, in general, the communication-history of the computation. This suggests that updating its free variables is only necessary when communication actually takes place. During a rendezvous, communication only occurs at the start and at the end of such a period. This suggests the following refined definition of bracketing:

Definition 3.7. A task is called bracketed if the brackets ´<´ and ´>´ are interspersed in its text, so that

(1) for each bracketed section, <S>, S is of the form
  (a) S´; call T.a (e#x); S",
  (b) accept b (u#v) do S´ or
  (c) S" endaccept;
  where S´ and S" do not contain any entry calls or accepts and may be null statements,
(2) each call and accept is bracketed as above. □

Clause 1a of this definition has remained the same; the other clauses have changed. Clearly, the intention of this change is that GI must be shown to hold again, whenever S' (in clause 1b) has been executed and hence before another call or accept can be encountered. (This implies of course a new interpretation of the validity of a $\{p\}\ C \| A\ \{q\}$-type formula.)

Now consider example 3.6 again and the accept in task T'. There is only one possibilty to bracket this accept properly according to the new definition, namely:

$\{h=0\}$ <accept a() do h:=1> $\{h=1\}$
<call T".b(1);> <h:=0 endaccept> $\{h=0\}$.

But now it becomes immediately clear that $GI \equiv h=0$ is not a global invariant anymore, because for this accept (A) and the call (C) in T the cooperation test (or rather the proof of $\{true \wedge h=0 \wedge GI\}\ C\|A\ \{true \wedge h=0 \wedge GI\}$) would also require one to show that $\{true \wedge h=0\}\ h:=1\ \{h=1 \wedge h=0\}$ which is evidently false. Hence, this - at least - suggests that the proof system is once more sound.

Third, and final, formulation of cooperation of ADA-CF proofs.

Definition 3.8. The proof outlines of $\{p_i\}$ task T_i $\{q_i\}$ (i=1..n) cooperate w.r.t. GI if

(1) for any syntactically matching pair, <C> and <A>, where
$C \equiv S_1$; call T_j.a $(\vec{e}\#\vec{x})$; S_2 and
$A \equiv$ accept a $(\vec{u}\#\vec{v})$ do S_1';> S; <S_2' endaccept (A within T_j),
the formula
$\{pre(C) \wedge pre(A) \wedge GI\}\ C\|A\ \{post(C) \wedge post(A) \wedge GI\}$
as defined below, holds,

(2) the assertions of the proof outline of $\{p_i\}\ T\ \{q_i\}$ contain no free variables subject to change in any T_j ($j \neq i$), for i=1..n. □

Having obtained the correct notion of bracketing and cooperation, the last task is to define formally how to prove the formulae of the cooperation test. During the remainder of this section, the following entry call and matching accept will be fixed, with pre and post-assertions as indicated:

$$\{p_1\} <S_1;\ \{\bar{p}_1\}\ \underline{call}\ T'.a\ (\vec{e}\#\vec{x})\ \{\bar{q}_1\};\ S_2>\ \{q_1\}$$
$$\{p_2\} <\underline{accept}\ a\ (\vec{u}\#\vec{v})\ \underline{do}\ \{p_2'\}\ S_1';>\ \{\bar{p}_2\}\ S\ \{\bar{q}_2\};\ <S_2'\ \{q_2'\}\ \underline{endaccept}>\ \{q_2\} \qquad (3.9)$$

These bracketed sections are denoted by <C> and <A> respectively (the call is part of a task T).

The question is, how to prove

$$\{p_1 \wedge p_2 \wedge GI\}\ C\|A\ \{q_1 \wedge q_2 \wedge GI\}. \qquad (3.10)$$

According to the semantics of a rendezvous and the intention of the bracketing and the cooperation test (and as suggested in the various examples), proof of this formula requires that the following partial proof outline can be completed:

$$\{p_1 \wedge p_2 \wedge GI\}$$
$$S_1;\ \vec{u},\vec{v}:=\vec{e},\vec{x};\ S_1';\ \{p \wedge GI\}\ S\ \{q \wedge GI\};\ S_2';\ \vec{x}:=\vec{v};\ S_2 \qquad (3.11)$$
$$\{q_1 \wedge q_2 \wedge GI\}$$

($\vec{x}:=\vec{v}$ denotes the simultaneous assignment of the v_i's to the x_i's; likewise for the assignment $\vec{u},\vec{v}:=\vec{e},\vec{x}$).

In this partial outline, only p_1, p_2, q_1, q_2 and GI are known assertions; p, q and the other assertions which are not shown, have to be found.

Completion of (3.11) in this form turns out to be not that satisfactory a solution. The problem is, that the cooperation test would force for each accept A, a set of different proof outlines to be completed, one for each call matching with A. This is, because the to-be-guessed assertions in (3.11) have to relate the states of the task containing the call, T, and the task containing the accept, T´, to each other; i.e., it is not possible just to substitute the assertions from the ´regular´ proof outline of the accept-body in (3.9) for the missing ones in (3.11). A second reason for not adopting this solution, is that the whole format of the cooperation test would break down:

Operationally, the cooperation test must show that whenever execution reaches a state in which control is simultaneously at some matching call-accept pair, the assumptions about the resulting rendezvous in the respective proof outlines are correct (and that GI holds again after updating its variables). Formulation of this test essentially hinges on the assumption that such states are all characterized by the appropriate (unique) pre-assertions of the proof outlines and GI. Now suppose the accept A to contain an inner call C´. As more than one proof outline has to be constructed for the outermost accept, more than one pre-assertion is obtained for the inner call. In other words, a particular pre-assertion does not fully characterize anymore, the states in which control is at C´. Consequently, the cooperation test - in this form - breaks down, because for each matching pair as many tests must be generated as there are different pre-assertions. The effect is self-propagating: Each of these tests results in a new proof outline to be completed for an accept A´ matching with the call C´. On its turn, A´ may contain an inner call too and still more checks have to be generated (although the total number of necessary tests remains finite).

These phenomena clearly show that in order to obtain a usable proof system for ADA-CF, another approach to the proof of (3.10) has to be found. An approach that retains the notion of proof outline in the sense that for an accept too, only one proof outline has to constructed; the cooperation test should not need additional ones. This means that the proof outline of the body of some accept must be canonical in the sense that its constituent assertions must be strong enough to justify the assumptions of each matching call and, symmetrically, must be weak enough to remain valid under the ´value-injection´ of each matching call.

Disregarding synchronization, a rendezvous is equivalent to an ordinary procedure-call. A similar quest for canonical proofs can be found in the literature dealing with proof rules for procedure-calls. There, the simplest approach is the simulation of parameter transfer by syntactical substitution of actual for formal parameters. To achieve this, a restriction must be imposed on the actual parameters allowed; see alse [2]. The same approach is adopted in the current case, and hinges on the following

Theorem 3.12. Let S be some ADA-CF statement, p and q two assertions; $\vec{u}$, $\vec{v}$ and $\vec{x}$ denote sequences of distinct variables and $\vec{e}$ denotes a sequence of expressions.
If (a) $FV(\vec{e}) \cap \{\vec{x}\} = \emptyset$, $\{\vec{u}\} \cap \{\vec{v}\} = \emptyset$, $(FV(S) \cup \{\vec{u},\vec{v}\}) \cap (FV(\vec{e}) \cup \{\vec{x}\}) = \emptyset$,

(b) the variables in $\vec{u}$ do not appear on the left-hand-side of any assignment in S or as <u>in out</u> parameter of any call in S

Then (1) $\{p\}\ S\ \{q\} \Rightarrow \{p[\cdot]\}\ S[\cdot]\ \{q[\cdot]\}$, provided $FV(q) \cap \{\vec{x}\} = \emptyset$

($[\cdot]$ denotes the variable substitution $[\vec{e},\vec{x}/\vec{u},\vec{v}]$),

(2) $\{p\}\ \vec{u},\vec{v}:=\vec{e},\vec{x};\ S;\ \vec{x}:=\vec{v}\ \{q\} \Rightarrow \{p\}\ S[\cdot]\ \{q\}$,
provided $FV(p,q) \cap \{\vec{u},\vec{v}\} = \emptyset$. □

We do not prove this theorem here, but instead refer the reader to the soundness and relative completeness proofs in [9]; a similar theorem due to E. R. Olderog is embodied in rule 26 in [2].

The restrictions 3.12(a) and (b) correspond precisely to the restrictions R1, R3 and R4 in section 2 (the third one in 3.12(a) is subsumed by R1). Under these restrictions, 3.12(2) shows actual parameter assignment and substitution to be equivalent; 3.12(1) shows that a canonical proof (outline) for S can be used to obtain information about any acceptable ´call´ (acceptable, meaning that assigning the actual parameters to the formal ones leaves the pre-assertion, p, valid).

We turn again to the formulation of the rendezvous-rule, to be used in proving the formulae in the cooperation test. These paragraphs, upto the formulation of the rule itself, are quite technical in content and the reader may safely skip them if he so wishes.

Using this theorem, formula (3.11) can immediately be rewritten as

$$\begin{aligned}&\{p_1 \wedge p_2 \wedge GI\}\\ &\quad S_1;\ S_1'[\cdot];\ \{p[\cdot] \wedge GI\}\ S[\cdot]\ \{q[\cdot] \wedge GI\};\ S_2'[\cdot];\ S_2 \qquad (3.13)\\ &\{q_1 \wedge q_2 \wedge GI\}\end{aligned}$$

(remember that $FV(p_1, q_1, p_2, q_2, GI, S_1, S_2) \cap \{\vec{u},\vec{v}\} = \emptyset$; p and q, still have to be determined).

Now consider the proof outline in (3.9) for $\{\bar{p}_2\}\ S\ \{\bar{q}_2\}$. Then theorem 3.12(1) implies the existence of a proof (outline) for $\{\bar{p}_2[\cdot]\}\ S[\cdot]\ \{\bar{q}_2[\cdot]\}$, too. This proof outline is not yet strong enough to be used in (3.13) because p and q have to contain state-information of both T´ (containing the accept of 3.9) and T (containing the call).

During execution of S, the state of T remains fixed and (hence) is characterized by the pre-assertion of the call, $\bar{p}_1$. Consequently, $\bar{p}_1$ is invariant over S; $\bar{p}_1$ is even invariant over S[·], because $FV(\bar{p}_1) \cap \{\vec{x}\} = \emptyset$ (this explains the role of this restriction in the call-axiom A1). GI, too, may be assumed to be invariant over S and hence over S[ ] (remember, GI does not contain formal parameters as free variables) because inner calls or accepts are dealt with separately. Now, it is a fact that, using auxiliary variables and GI, an assertion such as $\bar{p}$, ´talking´ about the state of two different tasks, T´ and T, can always be split into two assertions, $\bar{p}_1$ and $\bar{p}_2$, each talking about the state of only one task (i.e., $\bar{p}_1$ about T and $\bar{p}_2$ about T´); see e.g., the completeness proof in [9]. Consequently, formula (3.13) can be rewritten as:

$$\begin{aligned}&\{p_1 \wedge p_2 \wedge GI\}\\ &\quad S_1;\ S_1'[\cdot];\ \{\bar{p}_1 \wedge \bar{p}_2[\cdot] \wedge GI\}\ S[\cdot]\ \{\bar{p}_1 \wedge \bar{q}_2[\cdot] \wedge GI\}\ S_2'[\cdot];\ S_2\\ &\{q \quad q \quad GI\}.\end{aligned}$$

And, as far as the accept-body is concerned, there only remains the proof of $\{\bar{p}_2[\cdot]\}\ S[\cdot]\ \{\bar{q}_2[\cdot]\}$ for which it suffices to prove $\{\bar{p}_2\}\ S\ \{\bar{q}_2\}$ which is already part of the proof outline of T.

These arguments lead up to the last rule of the ADA-CF proof system, the rendezvous-rule:

R11. rendezvous:

$$\frac{\begin{array}{l}\{pre(S_1) \wedge pre(S_1') \wedge GI\}\ S_1;\ S_1'[\cdot]\ \{pre('call') \wedge pre(S)[\cdot] \wedge GI\} \\ \{pre('call') \wedge post(S)[\cdot] \wedge GI\}\ S_2'[\cdot];\ S_2\ \{post(S_2) \wedge post(S_2') \wedge GI\}\end{array}}{\{pre(S_1) \wedge pre(S_2') \wedge GI\}\ C \| A\ \{post(S_2) \wedge post(S_2') \wedge GI\}},$$

Where $C \equiv S_1$; <u>call</u> $T'.a\ (\vec{e}\#\vec{x})$; S_2 (within a task T)
$A \equiv$ <u>accept</u> a $(\vec{u}\#\vec{v})$ <u>do</u> S_1';> S <S_2' <u>endaccept</u>,
$[\cdot] \equiv [\vec{e},\vec{x}/\vec{u},\vec{v}]$,
'call' denotes the entry call within C.

Recapitulating, the premisses in this rule embody the cooperation test over the two bracketed sections. Assigning the actual to the formal parameters has been modelled by syntactic substitution (due to theorem 3.12 and the restrictions on the actual parameters of section 2). The same theorem implies that a new proof for the accept-body, S, need not be constructed for every matching call and instead we may just substitute the actual for the formal parameters in the proof outline for S in task T'. In other words, it is always possible to give a canonical proof for an accept-body which suffices for the cooperation test for all matching entry calls. In the first premiss, we must, among other things, show that the actual parameters obey the assumptions of the accept, i.e., we must derive pre(S)[·]. If they do, post(S)[·] specifies the result of executing the accept-body. The intermediate assertion, pre('call'), retains information about the variables in task T, other than the actual parameters; i.e., it retains information about those variables of T that cannot be changed by executing S.

Canonicity of the proof of an accept-body, is essential. We already indicated that while discussing the cooperation test. When constructing a proof outline, one constructs unique pre and post-assertions for every statement. Consequently, the assumption, permeating this section, that the proof of a component-task can always be rendered in the form of a proof outline, only now has been substantiated by the particular form of the rendezvous-rule.

The bodies of accept-statements can be proved canonically, but we did have to compromize: The rendezvous-rule clearly shows that for the bracketed sections associated with an accept, we do have to construct multiple proofs (similarly for entry calls). However, the completeness proof of the proof system ([9]) shows that bracketed sections need only contain one assignment each, so this seems a small price to pay.

4. PROOF OF THE BOUNDED BUFFER PROGRAM

In this section the example program in section 2 is proved correct w.r.t. the specification

{true}
<u>begin</u> <u>task</u> producer; <u>task</u> consumer; <u>task</u> buffer <u>end</u>
$\{\forall i=1..n\ vec1(i)=vec2(i)\}$.

For the proof, auxiliary variables are introduced:

- in the producre task, h_1; recording the sequence of values sent off,
- in the consumer task, h_2; recording the sequence of values received,
- in the buffer task $\bar{h}_1$ and $\bar{h}_2$; recording the sequence of values received, respectively, sent off.

These auxiliary variables denote sequences. In the proof outline, ´a˜b´ denotes the concatenation of sequences ´a´ and ´b´, or of the sequence ´a´ and the element ´b´. In the assertions, arrays or array-slices will also be used as sequences. Finally, the expression ´pool(x•y)´ is defined as follows (pool is a variable of type <u>array</u> (0..99) <u>of</u> <u>int</u>):

```
pool(x•y) = pool(x mod 100 .. (y-1) mod 100) , if x mod 100 ≤ y mod 100
            pool(x mod 100 .. 99)˜pool(0 .. (y-1) mod 100) , otherwise
```

Here follow the proof outlines (the labels are used in the next sections; the invariant of the while-loop in task buffer´ is denoted by I):

```
           task producer´
             array (1..n) of int vec1; int i; sequence of int $h_1$;
{$h_1$=Λ}  begin i:=1; — and initialize vec1 to some arbitrary values
             {$h_1$=vec1(1..i-1) ∧ i≤n+1}
             while i≤n do {$h_1$=vec1(1..i-1) ∧ i≤n}
               <$h_1$:=$h_1$˜vec1(i); {$h_1$=vec1(1..i) ∧ i≤n}
$l_1$:            call buffer´.put( vec1(i) );> {$h_1$=vec1(1..i) ∧ i≤n}
               i:=i+1; {$h_1$=vec1(1..i-1) ∧ i≤n+1}
             endwhile; {$h_1$=vec1(1..n)}
$l_2$:         <call buffer´.term()>
           end {$h_1$=vec1(1..n)} $l_3$:

           task consumer´
             array (1..n) of int vec2; int j; sequence of int $h_2$;
{$h_2$=Λ}  begin j:=1
             {$h_2$= vec2(1..j-1) ∧ j≤n+1)}
             while j≤n do
$l_4$:           <call buffer´.get( #vec2(j) ); $h_2$:= $h_2$˜vec2(j);>
               j:=j+1
             endwhile {$h_2$=vec2(1..n)}
$l_5$:         <call buffer´.term()>
           end {$h_2$=vec2(1..n)} $l_6$:

           task buffer´
             entry put, get, term;
             array (0..99) of int pool; int in, out, count, terms;
             sequence of int $\bar{h}_1$, $\bar{h}_2$;
{$\bar{h}_1$=Λ ∧ $\bar{h}_2$=Λ}
           begin in:=0; out:=0; count:=0; terms:=0;
             {count=in-out ∧ 0≤count≤100 ∧ $\bar{h}_1$=$\bar{h}_2$˜pool(out•in)} — {I}
             while terms≠2 do
$l_7$:           select count<100: {I ∧count<100}
                 <accept put(x) do $\bar{h}_1$:=$\bar{h}_1$˜x>
                   {count=in-out ∧ 0≤count<100 ∧$\bar{h}_1$=$\bar{h}_2$˜pool(out•(in+1))}
$l_8$:               pool(in mod 100):=x;
                   {count=in-out ∧ 0≤count<100 ∧$\bar{h}_1$=$\bar{h}_2$˜pool(out•(in+1))}
                 <endaccept>; in:=in+1; count:=count+1 {I}
               or count>0: {I ∧count>0}
                 <accept get(#y) do>
                   y:=pool(out mod 100)
                 <$\bar{h}_2$:=$\bar{h}_2$˜y endaccept>;
                 {count=in-out ∧ 0<count≤100 ∧ $\bar{h}_1$=$\bar{h}_2$˜pool((out+1)•in)}
                 out:=out+1; count:=count+1 {I}
```

```
        or true: {I}
          <accept term() do> null <endaccept>;
          terms:=terms+1 {I}
        endselect {I}
      endwhile {I}
    end {$\bar{h}_1=\bar{h}_2$^pool(out∘in)}
```

The general invariant is the obvious one, stating that each value that is sent is also received:

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

We show that the proof outlines cooperate w.r.t. this GI:

Consider the entry ´put´. There is only one matching pair to consider, and for this pair the rendezvous-rule requires the proofs of

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

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

$\{h_1=\text{vec1}(1..i) \wedge i\leq n \wedge \text{count}=\text{out}-\text{in} \wedge 0\leq\text{count}<100 \wedge$
$\bar{h}_1=\bar{h}_2\,\hat{}\,\text{pool}(\text{out}\bullet\text{in})\,\hat{}\,\text{vec1}(i) \wedge GI\}$

(2) $\{h_1=\text{vec1}(1..i) \wedge \text{count}=\text{in}-\text{out} \wedge 0\leq\text{count}<100 \wedge \bar{h}_1=\bar{h}_2\,\hat{}\,\text{pool}(\text{out}\bullet(\text{in}+1)) \wedge GI\}$

null

{ idem }

Clause (1) follows by applying the assignment-axiom twice; clause (2) by applying the null-axiom. Consequently, cooperation is established for this matching pair. The cooperation test for the entry ´get´ is an analogon of the above test and the test for the entry ´term´is trivial. So, the parallel composision rule can be applied:

$\{h_1=\Lambda \wedge h_2=\Lambda \wedge \bar{h}_1=\Lambda \wedge \bar{h}_2=\Lambda \wedge h_1=\bar{h}_1 \wedge \bar{h}_2=h_2\}$ -

begin task producer´; task consumer´; task buffer´ end

$\{h_1=\text{vec1}(1..n) \wedge h_2=\text{vec2}(1..n) \wedge \bar{h}_1=\bar{h}_2\,\hat{}\,\text{pool}(\text{out}\bullet\text{in}) \wedge \bar{h}_1=h_1 \wedge \bar{h}_2=h_2\}$

The post-assertion can be reduced to ´$\forall$ i=1..n vec1(i)=vec2(i)´ by applying the consequence-rule. Next, the auxiliary variables can be removed. Finally, substituting Λ, the empty sequence, for h_1, h_2, $\bar{h}_1$ and $\bar{h}_2$ and using the consequence-rule again, reduces the pre-assertion to true, thus completing the proof.

Although, the buffer-task has an entry ´term´ for the sole purpose of letting the task terminate, the proof does not refer to it. This is because we have only shown partial correctness of the program. In fact, in section 7 where, as an example, termination of the program is proved, the current proof outlines have to be extended.

5. SAFETY PROPERTIES

Section 3 presented a proof system for proving partial correctness properties of ADA-CF programs; i.e., properties expressing that if a program terminates, a certain assertion will hold afterwards. However, nonterminating concurrent programs are perfectly respectable (see section 8 for one such program); also, even if a program terminates, intermediate states such as those in which control is at some select, waiting for a rendezvous, may still be interesting.

Therefore, partial correctness properties are generalized by introducing safety properties. Such a property expresses that, in Lamport´s parlance ([13]), "during the computation of some program nothing bad happens". Partial correctness is a safety property because it expresses that a program does not terminate in an incorrect state. In general, a safety property, or safety assertion, is an invariant over the computation of a program, asserting what the program-state should obey when control arrives at some (or all) intermediate points in the program.

The principal question is whether the proof system has to be extended to prove such properties. The answer is, perhaps at first somewhat surprising: No; proof outlines as they are, are ´strong´ enough to derive safety properties from. On the other hand, it is not that surprising because, as indicated before, the pre-assertion of some statement (in some valid proof outline of a task T), characterizes the state of T whenever control arrives at this statement. The only moot point concerns the proof outlines of the accept-bodies in T. These, by definition, cannot specify the values of the formal parameters during a particular rendezvous and, consequently, do not fully characterize the state of T at such a time.

The rest of this section shows how to derive descriptions about the state at intermediate points, from a proof outline. Then, showing that some safety assertion, SA, holds for a program, means constructing a proof outline and showing that the state-descriptions derived from this outline imply the corresponding state-assertions of SA.

First some notation has to be introduced in order to (syntactically) specify such intermediate points, called frontiers of computation. This is not altogether trivial, as tasks communicate: Specifying that a task T is within some accept implies that some other task is at an entry call engaged in a rendezvous with this accept. Likewise, if this entry call is within another accept there must be a third task engaged in a rendezvous with that accept. So, in general there can be a chain of tasks, waiting for T to finish (executing the accept); a so-called calling chain for T. Evidently, not every set of ´points´ within a program is a frontier of computation which can (potentially) be reached during execution of this program.

Frontiers of computation are built up as follows:
First, control points are introduced to specify points in isolated tasks. Next, control points are combined into multi control points; these specify a point in some task T which is ´active´, in general together with a specific calling chain for T. Finally, a frontier of computation consists of a set of ´non conflicting´ multi control points. Such control points, multi control points and frontiers of computation do not appear, however, in the assertions of a proof outline. This contrasts with [13], in which Lamport introduces location predicates (these correspond to our location points) into his assertion-language so as to obtain a safety proof system.

To refer to a particular statement S, a unique name, ´S´, is introduced; e.g., to distinguish between two occurrences of an assignment x:=1. If ´C´ denotes an entry call, the bracketed section surrounding ´C´ is denoted by ´<C>´. Such names will not be further specified and the reader may think of some form of labeling.

Definition 5.0. Let ´S´ denote a statement, ´C´ an entry call and let T be the name of some task. A control point (c.p.) is one of the folowing:
(1) at(´S´) , (2) at(T) , (3) after(T) , (4) in(´C´).
A c.p. belongs to a task T, if the statement it refers to is part of the task T. □

The interpretation of these c.p.´s is suggested by their form: at(´S´) denotes the point just before ´S´; likewise, at(T) and after(T) denote the points just before and after the body of T; in(´C´) is somewhat special and denotes the ´point´ which is reached when ´C´ becomes engaged in a rendezvous (until this happens, the task would be at(´C´)). Such points are used to specify calling chains. Notice, that in(´C´) does not correspond to an actual point in the program text, although it is clearly a well-defined point which is reached during execution of a rendezvous when the actual parameters have been sent over but the rendezvous has not terminated yet; see also [15] and [4] in which similar observations are made.

Next, dependencies between c.p.´s of different tasks are described.

Definition 5.1. Let ´C_1´, ´C_2´, ..., ´C_n´ be a list of calls; each call within a different task. A calling chain (c.c.) is a list in(´C_1´), ..., in(´C_n´) such that

(1) ´C_1´ does not appear within an accept,

(2) ´C_{i+1}´ appears within an accept (syntactically) matching with ´C_i´ (i=1..n-1). □

Notice that the rendezvous´ specified in a c.c., are syntactically possible ones and nothing is implied about their actual occurrence.

Definition 5.2. Let x_1, x_2, ..., x_n be a list of c.p.´s, each x_i belonging to a different task. A multi control point (m.c.p.) is a tuple $\langle x_1, \ldots, x_n \rangle$ such that

(1) x_1, ..., x_{n-1} is a c.c.,

(2) n=1: x_n does not reference a statement appearing within an accept,
n>1: x_n is of the form at(´S´), and ´S´ appears within an accept matching with the entry call in x_{n-1}.

The task to which x_n belongs, is called the **frontier task** of the m.c.p. □

Definition 5.3. Let $X^{(1)}$, ..., $X^{(n)}$ be a list of m.c.p.´s. A **frontier of computation** (f.o.c.) is a set $\{X^{(1)}, \ldots, X^{(n)}\}$, such that in the sequence of c.p.´s which make up the m.c.p.´s in the above list, no two c.p.´s belong to the same task. □

This definition does not require a f.o.c. to specify progress in every task. Also notice that the set of f.o.c.´s of some ADA-CF program is always finite.

Having obtained enough notation to specify f.o.c.´s, the next assignment is to generate from a (valid) proof outline a description of the state at some f.o.c.

At first, disregard control points of the form at(´C´) (´C´ an entry call) and the fact that a state description should also include the values of the formal parameters (when appliccable). Then, it is clear what assertions to associate with any of the other c.p.´s: With a c.p. of the form at(´S´) associate pre(´S´), and associate post(´S´) with a c.p. of the form after(´S´). The discussion in the last part of section 3 indicates that pre(´C´) characterizes the state of the task containing the call ´C´, when a rendezvous (´through´ ´C´) is in progress. Consequently, the assertion to associate with in(´C´) is pre(´C´). A little thought makes it it clear that with a m.c.p., the conjunction of GI and the assertions associated with its constituent c.p.´s should be associated; with a f.o.c., the conjunction of the assertions associated with its constituent m.c.p.´s (and GI). GI is needed (1) to relate the states of the different tasks, referenced in the f.o.c., with each other, and (2) to express that the syntactic matches, as specified in the m.c.p.'s, match semantically; if some of them in a m.c.p. do not match, the conjunction can be made to yield false (by strengthening GI if necessary), which - as usual - is interpreted as stating that the m.c.p. cannot be reached in any computation of the program.

The use of GI in these formulae introduces, at first sight, an awkwardness because it restricts the set of f.o.c.'s for which such formulae can be derived from a proof outline, to those in which the c.p.'s belonging to one of the frontier tasks of the m.c.p.'s, specify points outside the bracketed sections of that proof outline. Fortunately, the completeness proof for the proof system in [9] shows that bracketed sections need only contain assignments to auxiliary variables, which are not part of the program proper. This implies that it remains possible to generate state descriptions at any f.o.c. of the original program, i.e., of the program without the statements involving auxiliary variables.

Finally, what assertions should be associated with c.p.'s of the form at('C') ('C' an entry call)? Certainly not pre('C') for the reason stated above, since no rendezvous involving 'C' is in progress as yet. In fact, the same discussion in the last part of section 3, suggests that the pre-assertion of the bracketed section surrounding 'C', '<C>', be associated with 'C'. As bracketed sections need only contain assignments to auxiliary variables, this seems a reasonable choice. Hence:

Definition 5.4. Let some program be given, together with a proof outline for it, valid w.r.t. some GI.

(1) With each c.p. x, an assertion, A(x), is associated as follows

if x≡at('S'),	'S' not an entry call,	then A(x) ≡ pre('S')
in('C'),	'C' an entry call,	pre('C')
at('C'),		pre('<C>')
at(T),	T the name of a task (-body),	pre(T)
after(T),		post(T)

(2) Let χ be some f.o.c. of the program, such that no c.p. owned by a frontier task specifies a point within a bracketed section. Let $x_1, x_2, \ldots, x_n$ be a list of all c.p.'s which are part of the m.c.p.'s in χ. Then, the assertion $R(\chi)$, characterizing the program state if control arrives at χ (i.e., if χ is reachable), is defined by

$$R(\chi) = A(x_1) \wedge \ldots \wedge A(x_n) \wedge GI. \qquad \square$$

One question remains unanswered. Namely, how to include the value of formal parameters in the state descriptions. In fact, we already have, because one of the functions of GI is to encode which values are communicated during a particular rendezvous; hence, GI can always be strengthened so as to encode these values (given the completeness of the proof system).

Example 5.5. Consider the example proof of section 4. We show that whenever control is at the f.o.c. $\{\langle in(l_1), at(l_8)\rangle\}$, x=vecl(i) holds. This is in fact quite trivial: According to definition 5.4.,

$$R(\{\langle in(l_1), at(l_8)\}\rangle) \rightarrow (h_4 = vecl(1..i) \wedge \bar{h}_4 = \bar{h}_2 \char`~ pool(out \bullet in) \char`~ x \wedge \bar{h}_1 = h_4).$$

This implies that $vecl(1..i) = \bar{h}_2 \char`~ pool(out \bullet in) \char`~ x$ and hence that x=vecl(i). $\square$

The contents of this section will be extensively used in the remainder of the paper.

6. DEADLOCK FREEDOM

As in [3], the concept of blocking is introduced. It originated with Owicki in [14]. In our context, a blocking of a program is a f.o.c. in which no component task can proceed (but in which the program has not terminated yet). Consequently, a program is deadlock free (w.r.t. some pre-assertion, p, characterizing the initial state in which execution starts), precisely when no blocking is semantically possible.

To simplify the definitions in the sequel somewhat, a restriction on ADA-CF programs is introduced: accepts may only appear in a program as the initial statement of a branch of a select. Notice, that an accept, A, is trivially equivalent to

select true: A endselect.

Consider a f.o.c. $\mathcal{X}$ for some program P. Intuitively (and roughly), P cannot proceed in $\mathcal{X}$ when the frontier tasks of the m.c.p.'s in $\mathcal{X}$ cannot proceed. I.e., when each frontier task is either terminated or at some entry call or select, but there are no syntactic matches between the entry calls and any accept in an open branch of one of the selects in these frontier tasks. This characterization is partly syntactic and partly semantic in nature. The syntactic part is the subject of:

Definition 6.0. Let $\mathcal{X}$ be a f.o.c. for a program begin task T_1; ... task T_n end. Let x_1, ..., x_s be the sequence of c.p.'s in $\mathcal{X}$; c.p. x_i belonging to task T_i. Furthermore, let T_1', ... T_t' be the sequence of frontier tasks of $\mathcal{X}$.
Then, $\mathcal{X}$ is a blocking frontier of computation (b.f.o.c.) iff
(1) $\mathcal{X} \neq \{\langle after(T_1)\rangle, \ldots, \langle after(T_n)\rangle\}$,
(2) for each T_i, there is a c.p. x_i belonging to it, i=1..n (hence s=n),
(3) each x_k' (in frontier task T_k') is either of the form after(T_k') or of the form at('S'), where 'S' is an entry call or a select, k=1..t,
(4) if x_k' is of the form at('call T_ℓ.a(...)') then $x_\ell \not\equiv$ after(T_ℓ) (k=1..t). □

Clause (1) tells us that the program should not have terminated yet; clause (2) enforces that a b.f.o.c. takes each component task into account and clause (3) indicates that a task can always proceed if it is not at an accept or entry call. Clause (4) is necessary because calling an entry of an already terminated task results in failure.

In order to formulate that execution cannot proceed in some b.f.o.c., an auxiliary predicate is introduced:

Let 'S' denote a statement select b_1: S_1 or ... or b_n: S_n endselect and let $I \subseteq \{1..n\}$. Then

$$CB(at('S'),I) = \bigwedge\{\neg b_i \mid i \in I\} \wedge \bigvee\{b_i \mid i \notin I\}.$$

Definition 6.1. Let $\mathcal{X}$ be a b.f.o.c.. The sequence consisting of c.p.'s in $\mathcal{X}$ of the form at('C'), respectively, at('S') ('C' an entry call, 'S' a select) are denoted by x_1, ..., x_n, respectively, y_1, ..., y_m. For each y_i, define a set $I(y_i)$ by
$k \in I(y_i)$ iff the k'th branch of the select y_i is for an entry called by one of the x_j's.
The blocking assertion for $\mathcal{X}$ is defined as

$$B(\mathcal{X}) = CB(y_1, I(y_1)) \wedge \ldots \wedge CB(y_m, I(y_m)).$$ □

Now, a program is deadlock free, simply if each b.f.o.c. either cannot be reached or is not blocked:

Definition 6.2. A program P is deadlock free w.r.t. a pre-assertion p, iff proof outlines can be constructed 'starting' in p, such that for each b.f.o.c. χ for P,

$$R(\chi) \wedge B(\chi) \rightarrow \text{false}. \qquad \square$$

Example 6.3. Consider the buffer-example in section 4 again. We show deadlock freedom.

It is a simple exercise to show that the b.f.o.c.'s are the f.o.c.'s of the form $\{\langle at(l_i)\rangle, \langle at(l_j)\rangle, \langle at(l_7)\rangle\}$ for $i=1,2,3$ and $j=4,5,6$ (l_3 and l_6 denote the points after the task bodies, i.e. $at(l_3)=after(producer')$ and $at(l_6)=after(consumer')$). Only the first b.f.o.c. ($i=1$, $j=4$) and the last one ($i=3$, $j=6$) will be considered; the others are left to the reader.

(1) $\chi = \{\langle at(l_1)\rangle, \langle at(l_4)\rangle\ \langle(at(l_7)\rangle\}$:
$B(\chi) = count \geq 100 \wedge count \leq 0 \wedge true$, which is false, independent of the truth-value of $R(\chi)$ (which is true incidently). So, although χ can be reached, χ will not be blocked and no deadlock occurs.

(2) $\chi = \{\langle at(l_3)\rangle, \langle at(l_6)\rangle, \langle at(l_7)\rangle\}$:
$B(\chi) = count<100 \vee count>0 \vee true$,
$R(\chi) = h_1=vec1(1..n) \wedge h_2=vec2(1..n) \wedge count=in-out \wedge 0 \leq count \leq 100 \wedge$
$\bar{h}_1 = \bar{h}_2 \,\bar{}\, pool(out \circ in) \wedge \bar{h}_1 = h_1 \wedge \bar{h}_2 = h_2$.
So, $B(\chi) \wedge R(\chi)$, well, does not evaluate to false!

Does this mean that the program deadlocks? No of course, but the proof outlines are too weak to prove otherwise! The problem is, that the exit condition of the while statement in the buffer task has not been taken into account: The statement necessarily terminates if both other tasks have executed their call for the term entry.

This is remedied as follows. The producer and consumer tasks are both extended with a new auxiliary variable; k_1 and k_2 respectively. The proof outlines are changed as follows (only the parts that change are shown; the changes to consumer' are analogous to the changes to producer'):

```
  task producer'
  --------; int $k_1$;
{$h_1=\Lambda \wedge k_1=0$}
begin
  -------

  -------
  endwhile {$h_1=vec1(1..n) \wedge k_1=0$}
  <call buffer'.term(); $k_1$:=1>
end {$h_1=vec1(1..n) \wedge k_1=1$}
```

```
  task buffer'
  ----------
{$I \wedge terms \neq 2$}
  select
    -------
  or true: {$I \wedge terms \neq 2$}
    <accept term () do> null
    <terms:=terms+1 endaccept {I}
    -------
end {$I \wedge terms=2$}
```

The reader will have no difficulties checking that these changes leave the proof outlines valid and that they cooperate w.r.t. the new general invariant

$$GI' \equiv GI \wedge terms=k_1+k_2.$$

Now consider the above b.f.o.c. χ again. $B(\chi)$ remains the same, but now

$$R(\chi) \equiv R'(\chi) \wedge k_1=1 \wedge k_2=1 \wedge terms \neq 2 \wedge terms=k_1+k_2$$

(where $R'(\chi)$ denotes the f.o.c. assertion as determined by the older proof outlines). It is easy to show that now $B(\chi) \wedge R(\chi) \rightarrow false$. $\square$

This example clearly shows that to prove deadlock freedom, in general, the proof outlines have to be stronger that the ones needed for proving partial correctness properties.

7. TERMINATION AND ABSENCE OF FAILURE

In this section, the proof system is extended (for the last time) in order to reason about termination and failure. To this end, proof rules have to be replaced by new ones. These changes also enforce adaptation of the notion of proof outlines. As these adaptations are straightforward, they are left to the reader.

First consider termination. A program terminates if it does not admit infinite computations; i.e., if each computation terminates either properly (by reaching the end of the program) or in failure or in deadlock. Notice that we implicitly make the assumption here, that execution of a program only halts when nothing else is possible. Clearly, without this assumption a program that does not loop or fail or deadlock need not terminate either, as execution might just stop in the middle of the program. In the terminology of [15], we assume that (1) execution of a program is fair in the sense that if a task T executes a call for some entry, only finitely many other calls for this entry can be accepted before the call of T is, and (2) execution of a program is just in the sense that if execution of a task can proceed, it will proceed in finite time. We will come back to this remark in section 9.

Obviously, the only source of non-termination is the while statement. The technique to prove termination of a while statement is well-known (cf. [2]): Find a quantity which decreases every iteration, but cannot decrease indefinitely. This is embodied in the following rule, taken from [2], which replaces the older rule for while statements, R4:

R4´. while:

$$\frac{p(n+1)\rightarrow b,\ \{p(n+1)\}\ S\ \{p(n)\},\ p(0)\rightarrow\neg b}{\{\exists n\ p(n)\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{endwhile}}\ \{p(0)\}},$$

where p(n) is an assertion with a free variable n, ranging over the natural numbers, such that $n \notin FV(S)$.

This well-known rule appears in various forms throughout the literature on proof systems for sequential languages. One might ask why it suffices in this concurrent context, too. The reason is simply that the behaviour of a task´s environment can be fully specified by the assertions associated with the task´s accepts and calls. Given these assertions, a component proof is constructed as were it for a sequential program.

Next, we turn to absence of failure. Ignoring failure caused by operations on data (e.g. division by 0), there remain two sources of failure: (1) a select without an open branch and (2) a call for an entry of a task already terminated (or about to terminate). Hence, these two situations must be proved never to occur.

Basically, proving absence of failure is quite straightforward. One simply strengthens the assertions of the proof outline so that the pre-assertion of any statement implies that execution of that statement does not result in failure.

As for (1), one must consequently show that the pre-assertion of any select implies the existence of at least one open branch of that select. This is embodied in the following rule which replaces the select rule R2:

R2´. select:
$$\frac{p \rightarrow (b_1 \vee \ldots \vee b_n)\ ,\ \{p \wedge b_i\}\ S_i\ \{q\}\ (i=1..n)}{\{p\}\ \underline{\text{select}}\ b_1 : S_1\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ b_n : S_n\ \underline{\text{endselect}}\ \{q\}}$$

Regarding the second possibility of failure, we can proceed as with the deadlock freedom test, showing that certain f.o.c.´s cannot semantically be reached:

Definition 7.0. A program P does not fail w.r.t a pre-assertion p, iff proof outlines can be constructed, starting in p, such that for each f.o.c.
$\chi = \{\langle x_1,\ldots,x_n, at(´\underline{call}\ T.a(\ldots)´)\rangle, \langle after(T)\rangle\}$
(x_1, ..., x_n ($n \geq 0$) a calling chain), the formula R(χ) yields false. □

Example 7.1. This is illustrated (for the last time) on the buffer example of section 4, for which termination and absence of failure is proved. First, consider termination of the while-loop in the buffer´ task. With every iteration, either a value is received or sent away, or the entry ´term´ is called. In the first two cases, $\bar{h}_1$ respectively, $\bar{h}_2$ is extended; in the last case the variable ´term´ increases. Hence, the quantity
$2n+2-|\bar{h}_1|-|\bar{h}_2|-terms$
($|\cdot|$ denotes the number of values making up its argument) would be a likely candidate to prove termination from. Correspondingly, if I denotes the loop invariant in the proof outline in section 4, the new parametrized one will be:
$I´(m) = I \wedge (m=2n+2-|\bar{h}_1|-|\bar{h}_2|-terms) \wedge (|\bar{h}_1| \leq n \wedge |\bar{h}_2| \leq n)$
$terms = |\{i \mid |\bar{h}_i|=n,\ i=1..2\}|$.
The last conjunct is necessary for showing that $I´(m+1) \rightarrow terms \neq 2$, the penultimate one for showing that $I´(0) \rightarrow terms=2$. As $I´(2n+2)$ holds before entering the loop, this proves termination of the while statement. The reader will have no difficulties showing that I´(m) is a loop invariant according to the new definition (cf. rule R4´). So, proving this formally, as well as proving termination of the loops in producer´ and consumer´, will be left to him. Notice that the last two conjuncts of I´(m) constitute a further refinement of the specification of the behaviour of the other task communicating with buffer´.
Next, absence of failure. Firstly, the third branch of the select in buffer´ is always open, so there is no problem here. Secondly, the f.o.c.
$\chi = \{\langle at(l_1)\rangle, \langle after(buffer´)\rangle\}$
should not be reachable. Using the proof outline of buffer´ strengthened as above, we get
$R(\chi) \rightarrow h_1 = vecl(1..i-1) \wedge i \leq n \wedge terms=2 \wedge terms=|\{i \mid |\bar{h}_i|=n,\ i=1..2\}| \wedge \bar{h}_1 = h_1$
which implies false. Reachability of $\{\langle at(l_4)\rangle, \langle after(buffer´)\rangle\}$ is treated completely analogously. To show that $\{\langle at(l_2)\rangle, \langle after(buffer´)\rangle\}$ and $\{\langle at(l_5)\rangle, \langle after(buffer´)\rangle\}$ are not reachable either, we have to resort to the same trick as in example 6.3, this time left to the reader.

8. CORRECTNESS OF A DISTRIBUTED PRIORITY QUEUE

This priority queue is based on Brinch Hansen´s sorting algorithm in [5]. In order to code the algorithm and its driver-task, some trivial extensions to ADA-CF are made by
(1) introducing task-arrays: If sort denotes some task, then sort(1..10) denotes an array of 10 identical tasks, denoted by sort(1), sort(2), ..., sort(10) respectively (; 10 can of course be replaced by any other integer constant).

The variables and labels in each of these component tasks are implicitly assumed to be indexed with the task-index to avoid name-clashes. Executing a task-array simply means executing all component tasks in parallel.

Two nullary functions, ´this´ and ´succ´, are introduced. Evaluation of ´this´, respectively, ´succ´ in a component of a task-array, returns the index of this component, respectively, the index of its successor, ´this´+1. This also holds for the last component of a task-array. However, such a last component will abort when it tries to call an entry of its nonexisting successor.

As the values of ´this´ and ´succ´ are syntactically determined, no changes of the proof system are necessary. We do need a rather obvious extension of the absence-of-failure test, though.

(2) introducing Dijkstra´s guarded loops ([7]):

$\underline{\text{do}}\ c_1 : S_1 [] \cdots [] c_n : S_n\ \underline{\text{od}}$,

where $c_1, \ldots, c_n$ are boolean expressions, guarding the ADA-CF statements $S_1, \ldots, S_n$. Execution of the loop-body is iterated as long as some boolean guard evaluates to true (on loop entrance). The loop-body is executed by arbitrarily choosing an S_i, whose guard, c_i, evaluates to true, and excuting it.

A moment of reflection will make it clear that for proving an assertion p to be a loop-invariant (for the above guarded loop), one should prove that $\{p \wedge c_i\}\ S_i\ \{p\}$ holds (i=1..n).

<u>Description of the algorithm and its implementation.</u>

The priority queue consists of a row of n identical tasks and can sort up to n elements (n is an arbitrary positive integer constant). The elements are input through the first task, which stores the smallest element so far encountered and passes on the rest to its successor. The latter task keeps the second smallest item and passes on the rest, and so on. The elements are output (in increasing order) through the first task. After each output, a task receives one of the remaining elements from its successor. A task is in equilibrium when it holds a single elements or when it holds none and neither do its successors. When the equilibrium of a task is disturbed (by its predecessor), it takes one of the following actions:

(1) if the task now has two elements, it keeps the smaller one and passes on the larger one to its successor, or

(2) if the task now has no elements but its successor does, it takes the (smallest) element from its successor.

The priority queue is implemented by a task-array sort(1..n). The elements of each task are kept in an array ´here´; ´len´ contains the number of elements currently present, while ´rest´ contains the number of elements which have been passed on. Each component task has two entries, ´put´ and ´get´; to put elements into, respectively, to get elements from a task. When a call for ´put´ is accepted, the received element is placed in the array ´here´. Then, if the task finds itself having two elements, it sorts the elements in ´here´ into increasing order and sends off the larger one (contained in here(1)). An entry call for ´get´ is only accepted by a task, if len=1 holds. In that case, the task sends back its element, after which it obtains the element from its successor (if it has any).

```
task sort(1..n)
  entry put, get;
  array (1..2) of int here; int rest, len, temp;
```

```
begin rest:=0; len:=0;
  while true do
k:   select true:
        accept put(u) do len:=len+1; here(len):=u endaccept;
        if len=2 then
           if here(2)<here(1) then
              temp:=here(2); here(2):=here(1); here(1):=temp;
           endif;
l:         call sort(succ).put(here(2));
           rest:=rest+1; len:=1;
        endif
     or len=1:
        accept get(#v) do v:=here(1) endaccept; len:=0;
        if rest>0 then
m:         call sort(succ).get(#here(1)); rest:=rest-1; len:=1
        endif
     endselect
  endwhile
end
```

To drive the priority queue, the following driver-task is used:

```
task driver
  int x; bag of int bag;
begin bag:=∅;
  do |bag|<n: x:=?; l₀: call sort(1).put(x); bag:=bag⊕[[x]]
  [] |bag|>0: m₀: call sort(1).get(#x); bag:=bag⊖[[x]]
  od
end
```

Here, x:=? denotes the assignments of an arbitrary (integer) value to x. The variable ´bag´ is of type bag of int, which means that it is a set in which the same value may appear more than once. The operators ⊕ and ⊖ denote the union and the splitting of bags (no values are thrown away); ´[[...]]´ is our bag-constructor and ∅ denotes the empty bag. The variable ´bag´ retains all values which have entered the queue but have not left it as yet, and is needed to express the safety property we want to prove below. Notice that the nondeterministic way in which a branch is choosen during each iteration of the guarded loop, forces the task-array to function as a priority queue rather than as a sorter.

Correctness proof.

Consider the program

begin task driver; task sort(1..n) end.

We want to derive for this program, the safety assertion

$R(\ \{\langle after(m_0)\rangle\}\ ) \rightarrow x=min(bag)$.

I.e., whenever a value is removed from the queue, it is minimal amongst the values which have entered the queue up till now. Notice, that to say that it is minmal amongst the values which are still in the queue, would be a weaker assertion, as this would allow the program to forget some values. This motivates the use of the bag-variable in the driver.

In the proof outline(s) of the task-array, auxiliary variables, ´kept´ and ´sent´, are used; all of type bag of int. The values which are present in sort(i) are kept in $kept_i$ (remember, the variables are assumed to be indexed);

$sent_i$ contains the values which have been sent to sort(i)´s successor. In the proof outline of the driver-task only one auxiliary variable is introduced, $sent_0$, of the same type and with the same function as the other $sent_i$´s.

The general invariant expresses that no transmitted value is lost:

$$GI \equiv \bigwedge_{i=0}^{n-1}(sent_i = kept_{i+1} \oplus sent_{i+1}).$$

The proof outlines of the component tasks are all the same. Hence we will give a ´canonical´ one. To obtain the proof outline of a component task, the reader should substitute the task-index for all appearances of the function ´this´ in the assertions.

Finally, the loop-invariant of the while-loop in the task-array is split into two parts, L and R (by convention, $\min(B)=\infty$ if $B=\emptyset$):

$$L \equiv kept=[\![here(i) \mid i=1..|kept|]\!] \wedge len=|kept| \wedge here(1)\leqslant\min(sent) \wedge rest=|sent|\geqslant 0,$$
$$R \equiv (len=0 \rightarrow rest=0) \wedge rest\leqslant n-this \wedge 0\leqslant len\leqslant 1.$$

```
task driver´
  int x; bag of int bag, sent₀;
{sent₀=∅}
begin bag:=∅;
  {bag=sent₀ ∧ 0≤|bag|≤n} — the loop-invariant
  do |bag|<n:
    x:=?; {bag=sent₀ ∧ 0≤|bag|<n}
l₀: <call sort(1).put(x); sent :=sent₀⊕[[x]];>
    {bag=sent₀⊖[[x]] ∧ 0≤|bag|<n}
    bag:=bag⊕[[x]]
    |bag|>0: {bag=sent₀ ∧ 0<|bag|≤n}
m₀: <call sort(1).get(#x); sent₀:=sent₀⊖[[x]];>
    {bag=sent₀⊕[[x]] ∧ 0<|bag|≤n ∧ x=min(bag)}
    bag:=bag⊖[[x]]
  od {false}
end {false}

task sort(1..n)´
  entry put, get;
  array (1..2) of int here; int rest,len,temp;
  bag of int kept, sent;
{kept=sent=∅}
begin rest:=0; len:=0;
  {L ∧ R}
  while true do
k:  select true: {L ∧ R}
      <accept put(u) do> {L ∧ R}
        len:=len+1; here(len):=u
        {(L ∧ R)[len-1/len] ∧ here(len)=u}
      <kept:=kept⊕[[u]] endslect;>
      {L ∧ (len≠2 → R) ∧ (len=2 → R[len-1/len] ∧ rest<n-this)}
      if len=2 then
        if here(2)<here(1) then
          {L ∧ R[len-1/len] ∧ rest<n-this ∧ len=2 ∧ here(2)<here(1)}
          temp:=here(2); here(2):=here(1);
          {kept=[[here(2)]]⊕[[temp]] ∧ len=|kept| ∧ here(1)≤min(sent) ∧ rest=|sent|≥0 ∧
          R[len-1/len] ∧ rest<n-this ∧ len=2 ∧ here(1)=here(2) ∧ temp<here(2)}
          here(1):=temp
        endif;
        {L ∧ R[len-1/len] ∧ rest<n-this ∧ len=2 ∧ here(1)≤here(2)}
1:      <call sort(succ).put(here(2)); kept:=kept⊖⟦here(2)⟧;
         sent:=sent⊕⟦here(2)⟧;>
        {(L[rest+1/rest] ∧ R)[len-1/len] ∧ rest<n-this ∧ len=2}
        rest:=rest+1; len:=1
      endif [L ∧ R]
    or len=1: {L ∧ R ∧ len=1}
      <accept get(#v) do> {L ∧ R ∧ len=1}
        v:=here(1) {L ∧ R ∧ len=1 ∧ v=here(1)}
      <kept:=kept⊕⟦v⟧ endaccept;> {L[len-1/len] ∧ R ∧ len=1}
      len:=0; {L ∧ (rest=0 → R) ∧ (rest>0 → R[len+1/len]) ∧ len=0}
      if rest>0 then {L ∧ R[len+1/len] ∧ rest>0 ∧ len=0}
m:      <call sort(succ).get(#here(1)); kept:=kept⊕⟦here(1)⟧;
         sent:=sent⊖⟦here(1)⟧;>
        {(L[rest-1/rest] ∧ R)[len+1/len] ∧ len=0 ∧ rest>0}
        rest:=rest-1; len:=1
      endif {L ∧ R}
    endselect {L ∧ R}
  endwhile {false}
end {false}
```

Next, we prove cooperation w.r.t. GI:

(1) Consider the call for put in sort(i) and the corresponding accept in sort(i+1) (i<n).
The first premiss of the rendezvous rule trivially holds, as no auxiliary variables are updated and the pre-assertion of the accept body makes no assumption about the value of the actual parameter. For the second premiss, one should prove:

$$\{L_i \wedge R_i[len_i-1/len_i] \wedge rest_i<n-i \wedge len_i=2 \wedge here_i(1)\leq here_i(2) \wedge$$
$$(L_{i+1} \wedge R_{i+1})[len_{i+1}-1/len_{i+1}] \wedge here_{i+1}(len_{i+1})=here_i(2) \wedge GI\}$$
$$kept_{i+1}:=kept_{i+1}\oplus[\![here_i(2)]\!];\ kept_i:=kept_i\ominus[\![here_i(2)]\!];$$
$$sent_i:=sent_i\oplus[\![here_i(2)]\!]$$
$$\{(L_i[rest_i+1/rest_i] \wedge R_i)[len_i-1/len_i] \wedge rest_i>n-i \wedge len_i=2 \wedge$$
$$L_{i+1} \wedge (len_{i+1}\neq 2 \rightarrow R_{i+1}) \wedge (len_{i+1}=2 \rightarrow R_{i+1}[len_{i+1}-1/len_{i+1}] \wedge rest_{i+1}<n-i)\}$$

This is a simple but arduous exercise. Notice, that
$$(len\neq 2 \rightarrow R) \wedge (len=2 \rightarrow R[len-1/len]$$
is just a rewriting of R[len-1/len].

(2) Consider the call for get in the driver and the accept in sort(1).
Again the first premiss is easy to prove, so there remains the proof of

$$\{bag=sent_0 \wedge 0<|bag|\leq n \wedge L_1 \wedge R_1 \wedge len_1=1 \wedge x=here_1(1) \wedge GI\}$$
$$kept_1:=kept_1\ominus[\![here_1(1)]\!];\ sent_0:=sent_0\ominus[\![x]\!]$$
$$\{bag=sent_0\oplus[\![x]\!] \wedge 0<|bag|\leq n \wedge x=min(bag) \wedge L_1[len_1-1/len_1] \wedge R_1 \wedge len_1=1 \wedge GI\}$$

This too, is a simple exercise. The crusial fact that x=min(bag), is a consequence of the following conjunction which is part of the pre-assertion:

$$bag=sent_0 \wedge kept_1=[\![here_1(i)\,|\,i=1..|kept_1|]\!] \wedge len_1=|kept_1| \wedge$$
$$here_1(1)\leq min(sent_1) \wedge len_1=1 \wedge x=here_1(1) \wedge sent_0=sent_1\oplus kept_1.$$

The other cooperation tests are left to the reader.

Hence, the proof outlines cooperate and can be combined so as to yield
$$R(\{<after(m_0)>\} \rightarrow bag=sent_0\oplus[\![x]\!] \wedge 0<|bag|\leq n \wedge x=min(bag) \wedge GI\},$$
which trivially implies x=min(bag), the required safety property.

Next, we show absence of failure. As none of the tasks terminate and all selects have a branch guarded by true, the only source of failure are the entry calls in sort(n). However, it is easy to show that these can never be reached, as:
(1) $R(\{\langle at(l_n)\rangle\}) \rightarrow rest_n < n-n \wedge rest_n \geqslant 0$, and
(2) $R(\{\langle at(m_n)\rangle\}) \rightarrow rest_n \leqslant n-n \wedge rest_n > 0$.

This leaves us with deadlock freedom. Because the tasks do not terminate, the only blocking f.o.c.´s are of the form

$\{\langle at(x_0)\rangle, \langle at(x_1)\rangle, \ldots, \langle at(x_n)\rangle\}$, where $x_0 \in \{l_0, m_0\}$ and $x_i \in \{k_i, l_i, m_i\}$ (i=1..n).

As we showed that sort(n) cannot be $at(l_n)$ or $at(m_n)$, this means that the blocking f.o.c.´s can be partitioned into segments $sort(1..i_1)$, $sort(i_1+1..i_2)$, ..., $sort(i_k+1..n)$ $(1 \leqslant i_1 < i_2 < \ldots < i_k < n)$, such that in each segment sort(i..j) the tasks sort(i), ..., sort(j-1) are at one of their entry calls while sort(j) is at its select. Consider a segment sort(i..j) (i<j); we show that it cannot be blocked. If sort(j-1) is $at(l_{j-1})$ no blocking can occur as the corresponding branch of the select in sort(j) is open. So, we only need to consider b.f.o.c.´s of the form

$\chi = \{\langle at(x_i)\rangle, \ldots, \langle at(x_{j-2})\rangle, \langle at(m_{j-1})\rangle, \langle at(k_j\rangle\}$, $x_h \in \{l_h, m_h\}$.

For such f.o.c.´s

$B(\chi) \wedge R(\chi) \rightarrow rest_{j-1} > 0 \wedge rest_{j-1} = |sent_{j-1}| \wedge len_j = 0 \wedge rest'_j = 0 \wedge len_j = |kept_j| \wedge rest_j = |sent_j| \wedge sent_{j-1} = kept_j \oplus sent_j$,

which implies false. Consequently, such b.f.o.c.´s cannot be reached. Next, we should check segment of the form sort(i..i) and we should take the driver into account. However, these are dealt with just as easily and are left to the reader.

9. EXTENSIONS

We discuss some additional ADA-constructs which can be accommodated for by the proof system. We also discuss the nature of some of the restrictions imposed upon the proof system.

There is of course a definite bound on what can be added without necessitating major changes or extensions to the proof system. For one, the fact that a program consists of a fixed set of tasks is quite essential; otherwise, the general invariant cannot be formulated. Also, the possibility in full ADA of having access-variables referencing tasks is quite outside the scope of this proof system.
It is possible to extend ADA-CF with a rudimentary block-structure by allowing programs to appear in a task-body. I.e., by defining begin task {;task} end (cf. section 2) to be a valid stat (-ement), too. As it is not possible to allow communication to occur between a task within a block and a task outside that block (for the reason stated above), such blocks are of limited value and we will not discuss the extensions needed for our system.

There are some ADA-statements which only need trivial extensions to the proof system. These are (1) the delay, (2) the conditional and timed entry call and (3) the conditional and timed accept. The effect of these statements is either not expressible in our assertion language (as for the delay, wich suspends execution of the task that executes it for some time) or we do not want to take their effect into account (as for the other statements).

Consider, for instance, the conditional entry call

<u>select</u> call_st; stats <u>else</u> stats <u>endselect</u>.

This statement has the following semantics: If a rendezvous with the called task is immediately possible, it is performed and the statements after the entry call are executed. Otherwise, the else-part is executed.

Consider two tasks, T and T´. T executes a conditional entry call; at the same time T´ executes a matching accept and a rendezvous consequently occurs. By judiciously slowing down execution of T´ or by judiciously speeding up execution of T, such a rendezvous can always be caused not to happen. As we certainly do not want to make any assumptions about the differences in the speed of execution between the various tasks, this means that we can never be sure whether the entry call or the else-part is taken in a conditional entry call. Consequently, the following proof rule is obtained:

R12. cond. call:

$$\frac{\{p\}\ C;S'\ \{q\},\ \{p\}\ S''\ \{q\}}{\{p\}\ \underline{\text{select}}\ C;S'\ \underline{\text{else}}\ S''\ \underline{\text{endselect}}\ \{q\}},$$

where ´C´ denotes an entry call.

Finally, we consider the <u>terminate</u>-statement. This statement introduces a so-called distributed termination convention in ADA-CF and requires some less trivial extensions to the proof system.

<u>terminate</u>

A terminate-statement (abbreviated to termstat) may appear as the sole statement in a branch of a select. If such a branch is executed, it causes the task containing the select to terminate (normally). An (open) branch containing a termstat can only be selected when all other tasks of the program are either terminated or waiting at an accept with an open branch containing a termstat, too.

Execution of a termstat results in control being transferred to the end of the body of the task executing it. Consequently, control will never arrive at the location immediately after the termstat. This suggests the following axiom to be used when constructing component proofs:

A5. terminate: {p} <u>terminate</u> {false}.

But this is not enough. The post-assertion of a task-body characterizes the state of that task when it terminates. If a task terminates by executing a termstat, it does so in a state characterized by the pre-assertion of this termstat. So, to be consistent, the post-assertion of a task-body must imply the pre-assertion of every termstat in the task-body which may indeed be executed. This necessitates a modification of the cooperation test, which has to be extended with the following additional clause:

(3) For a select ´S´, define TP(´S´) = $b_{k_1} \vee \ldots \vee b_{k_\ell}$, where $b_{k_1}, \ldots, b_{k_\ell}$ ($k_\ell \geq 0$) is the (possibly empty) list of boolean expressions, guarding the branches of ´S´ that contain a termstat.
Then, for each f.o.c. χ of the form $\{\langle x_1\rangle, \ldots, \langle x_n\rangle\}$, where x_i is a c.p. of the form after(T_i) or at(´S_i´), ´S_i´ a select within task T_i, the following should hold:
If $x_{i_1}, \ldots, x_{i_k}$ is the list of c.p.´s in this f.o.c., referencing selects,

then for each $x_i \in \{x_{i_1},\ldots,x_{i_k}\}$, the formula
$$(R(\chi) \wedge TP(x_{i_1}) \wedge \ldots \wedge TP(x_{i_k}) \wedge post(T_i)) \rightarrow pre(x_i),$$
must hold. □

Notice, that the above set of **distributed termination** f.o.c.´s (d.t.f.o.c.´s) that has to be considered, is particulary simple, because a task cannot select a termstat to be executed if the program is at a f.o.c. in which a calling chain exists.

The deadlock freedom test of section 6 has to be adapted too, as a d.t.f.o.c. can also be a b.f.o.c. (cf. definition 6.0). Consequently, each b.f.o.c. of this form which may be reached and may block (cf. definition 6.2), should lead to termination. This results in the following reformulation of the deadlock freedom test.

Definition 9.0. A program P is deadlock free w.r.t. a pre-assertion p, iff proof outlines can be constructed starting in p, such that for each b.f.o.c. χ of P, either

(1) $\neg(R(\chi) \vee B(\chi))$ holds, if χ is not a d.t.f.o.c., or,

(2) $R(\chi) \wedge B(\chi) \rightarrow TP(x_{i_1}) \wedge \ldots \wedge TP(x_{i_k})$ holds, if χ is a d.t.f.o.c. (notation as in (3) above). □

Next, some of the restrictions of ADA-CF are discussed. For the connaisseur of ADA, perhaps the most noticable restriction of ADA-CF is the absence of entry queues. In full ADA, each entry has an entry queue associated with it. A task executing an entry call is put on the queue associated with it. When a task is ready to accept a call for an entry, the call of the task which is on top of the queue for this entry, is accepted first. An entry queue for some entry e, has an attribute, e´count, associated with it, which equals the number of tasks currently on the queue. Let us ignore such attributes for the moment.

Entry queues implement a mechanism for selecting entry calls to be accepted, which is fair (in the sense of section 7). As is well-known, fairness assumptions about the execution of a program do not alter the set of valid safety properties of the program. However, as indicated in section 7, the property of termination does depend on fairness assumptions. Consequently, an important question is whether there are programs which need not terminate under the fairness assumption of section 7, but do under the (seemingly) stronger form of fairness as implemented by entry queues (;stronger, because when a task executes an entry call and is consequently put on an entry queue, it is exactly known how many calls will be accepted before his call is accepted).

In [15] the following theorem is proved:

Theorem. Let P be an ADA-CF program (which consequently does not refer to any e´count attribute). Then, the set of possible executions for P (under the fairness assumptions of section 7) is equivalent with the set of possible executions of P under the explicit queuing model. □

Hence, the above question can be answered in the negative. This theorem is proved by formalizing the observation that it is impossible for a program (not using queue attributes) to arrange for a specific order of tasks on an entry queue.

Prohibiting queue attributes is quite essential. If a program may use them, it can influence the ordering of tasks on entry queues and consequently, even its set of valid safety properties may change. This is illustrated in the following

Example 9.1. Consider the program below (due to Job Zwiers):

```
task T                                task T´ int c;
begin call T".e(1) end                begin c:=0;
                                        while c=0 do call T".f(#c) endwhile;
task T" entry e, f; int x, y;           call T".e(2)
begin x:=0;                           end
  while x=0 do
    accept f(#u) do x:=e´count; u:=x endaccept
  endwhile;
  accept e(v) do x:=v endaccept;
  accept e(w) do y:=w endaccept
end
```

In this program, task T´ suspends executing its entry call until T has executed his. It does so, by inspecting the entry queue of entry e and by looping until the queue is not empty anymore. Consequently, the following formula is valid:

{true} begin task T; task T´; task T" end $\{x=1 \wedge y=2\}$. □

Such queue attributes act as a set of hidden variables which are shared between the component tasks of a program. So, one might expect that using these attributes will force the proof system to be extended with some form of interference freedom test ([14]). We did not follow up on this suggestion as yet.

Finally, consider the syntactic restrictions in section 2 on the actual and formal parameters. As a result of these restrictions, parameter transfer can be modelled using syntactic substitution. Recent research ([6] and [10]) has shown that these restrictions can be relaxed (for sequential procedure calls) and that some forms of aliasing in the actual parameters can be allowed, by refining the notion of syntactic substitution in assertions. These techniques can be applied in the current context, too. The resulting rendezvous rule will be somewhat less elegant, as the second clause of theorem 3.12 breaks down if aliasing occurs in the actual parameters. This is however outside the scope of this paper.

ACKNOWLEDGEMENTS.

We would like to thank the members of the RUU-MC working group on semantics for the privilege of presenting an earlier version of the proof system to them; Marly Roncken, Adrie van Bloois and Job Zwiers for some fruitful discussion and A. Salwicki for pointing out the difference between ´interleaving semantics´ and ´maximum paralellism semantics´. Finally, the first author would like to thank the Dutch and German railways and the occupants of compartement 224 of the IC124 from Munchen to Utrecht on october 24, 1981 for the hospitable and inspiring athmosphere in which the initial stage of the research culminating in this paper was carried out.

REFERENCES.

[1] ADA: The Programming Language ADA. Reference Manual. LNCS 106, Springer Verlag, New York, 1981.

[2] Apt K.R: Ten Years of Hoare´s Logic: A Survey - Part 1. TOPLAS 3-4, p.431-484, 1981.

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

[4] Astesiano E., E. Zucca: Semantics of Distributed Processes Derived by Translation. Proceedings of the 11th GI-Jahrestagung, Informatik Fachberichte 50, p.78-87, Springer Verlag, New York, 1981.

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

[6] Cartwright R., D. Oppen: The Logic of Aliasing. Acta Inf. 15, p.365-384, 1981.

[7] Dijkstra E.W.: A Discipline of Programming. Prentice Hall, 1976.

[8] Gerth R.T., W.P. de Roever, M. Roncken: Procedures and Concurrency: A study in Proof. Proceedings of the Vth International Symposium on Programming, LNCS 137, p.132-163, Springer Verlag, New York, 1982.

[9] Gerth R.T.: A Sound and Complete Hoare Axiomatization of the ADA Rendezvous. Proceedings of the 9th ICALP, LNCS 140, p.252-265, Springer Verlag, New York, 1982.

[10] Gries D., G.M. Levin: Assignment and Procedure Call Proof Rules. TOPLAS 2-4, p.564-579, 1980.

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

[12] Kuiper R.: Private Communication, 1982.

[13] Lamport L.: The Hoare´s Logic of Concurrent Programs. Acta Inf. 14, p.21-37, 1980.

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

[15] Pnueli, A., W.P. de Roever: Rendezvous with ADA - A Proof Theoretical View. Proceedings of the ADATEC Conference, 1982.

[16] Roncken M., N. van Diepen ,M. Kramer, W.P. de Roever: A Proof System for Brinch Hansen´s Distributed Processes. Technical Report RUU-CS-81-5, Department of Computer Science, University of Utrecht, 1981.

[17] Salwicki A.: Critical Remarks on MAX Model of Concurrency. Proceedings of the Logics of Programming Workshop 1981, LNCS 131, p.397-405, Springer Verlag, New York, 1982.