

A PROOF SYSTEM FOR CONCURRENT ADA PROGRAMS*

Rob GERTH and Willem P. DE ROEVER

Department of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA Utrecht, Netherlands

Communicated by K. Apt

Received January 1983

Revised October 1983

Abstract. A subset of ADA is introduced, ADA-CF, to study the basic synchronization and communication primitive of ADA, the rendezvous. Basing ourselves on the techniques introduced by Apt, Francez and de Roever for their CSP proof system, 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. No prior exposure of the reader to parallel program proving techniques is presupposed. 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. Features of ADA expressing dynamic process creation and realtime constraints are not covered by our proof methods. Consequently, we do not claim that the methods described can be extended to full ADA without serious additional further research.

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 [15]. Thus, we take Hoare's recommendation [18] to work out simple consistent subsets of ADA seriously. The proof system is based on the CSP proof system in [5] which has as key-notion, the notion of cooperation between proofs. For languages for distributed computing, this notion expresses on the level of proofs that a process functions correctly, provided the assumptions which the environment of that process meets, allow it to honor the commitment the environment needs.

The ADA rendezvous mechanism simply combines the procedure concept with synchronized communication of its parameters. Now, one may wonder whether this simple combination allows for an equally simple combination of proofs. Our proof

* These investigations were supported (in part) by the Foundation for Computer Science Research in the Netherlands (SION) with financial aid from the Netherlands Organization for the Advancement of Pure Research (ZWO).

system demonstrates that this is the case, indeed. It shows that proofs for this combination simply split into separate proofs for

- (1) the procedure-body (assuming no communication) and
- (2) the synchronization and communication involved.

We take the absence of additional proof theoretical complexity as an indication that, seen in the broader perspective of developing programming concepts in general, the rendezvous mechanism provides us with an elegant communication primitive. This situation is a particularly lucky one, since the given characterization of the rendezvous extends to so-called remote procedure calls in general (see [2]). Consequently, our proof techniques apply in principle to a whole family of languages for distributed communication, including e.g. Concurrent Pascal [8], Distributed Processes (DP) [9], *MOD [11] and Mesa [22]; this thesis receives additional support from the fact that our proof techniques were initially developed for another language, DP [26, 14].

Technically, the main contribution of this paper is the generalization of the idea of cooperation, as developed for CSP-type communication of transmitting simple values, to ADA-type communication. A second contribution of this paper is that it describes a method for proving invariance (or safety) properties in general. Notably, this method does not require further strengthening of our proof techniques. It describes how to derive more information from the same proofs, instead.

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 [25] or [15]. 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 and [25]. 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. 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 (i.e., 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 [9] 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 [17]. Also, the absence of entry queues and some syntactic restrictions on the variables in ADA-CF are discussed. Finally, Section 10 formulates a conclusion and discusses some related work.

2. The subset ADA-CF

The syntax of ADA-CF is described, using a BNF-grammar augmented with the following embellishments (see also [1, § 1.5]):

(a) Script 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 or 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_part [label] begin stats end [label]
label      ::= label_id:
decl_part  ::= [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 | acc_st | call_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
acc_st     ::= accept entry_id (formal_part) do stats endaccept
formal_part ::= [in_id_list] [# in_out_id_list]
call_st    ::= call task_id.entry_id (actual_part)
actual_part ::= [expr{, expr}] [# in_out_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 variables in the *decl_part* of the task;

R3. No variable in the *in_id_list* of any accept-statement (*acc_st*) may appear on the left-hand side of any assignment or in the *in_out_id_list* of 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 a task-body, that 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 task 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 a task that has already terminated or
- (2) a task terminates, while other tasks are still waiting for a rendezvous with one of the entries of this terminated task.¹

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, consists of value parameters (i.e., are of mode **in**, using ADA-parlance); the second set consists of initialized result parameters (i.e., are of mode **in out**). 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. More, precisely, if $C \equiv \text{call } T.e(f \# x)$ and $A \equiv \text{accept } e(u \# v) \text{ do } S \text{ endaccept}$, then executing a rendezvous between C and A can be seen as executing the statement " $u := f; v := x; S; x := v$ "; this statement will in the sequel be denoted by " $C \| A$ ". Note that the association of the actual with the formal parameters (call-by-value-result) is in full agreement with the ADA reference manual [1, § 6.2].

¹ In full ADA, an exception would have been raised in the calling task only, in these cases, possibly causing it to fail but otherwise not influencing execution of the other tasks [1, § 9.5].

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.²

Executions (or computations) of ADA-CF programs are modelled as arbitrary interleavings of the indivisible or atomic actions of the component tasks. I.e., we assume an *interleaving semantics* (INT) and, in this respect, 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 parallelism 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 such an assumption is not warranted or in which time-sharing occurs. In fact, it is the intended execution model of the ADA reference manual [1, § 9]. The reader is referred to [27] for a more complete exposition. While our proof system is sound under both INT and MAX, it is complete under INT only (see [27] for a counter-example).

Finally, in the sequel we will at times refer to 'program-states' or 'states' reached by a computation of a program. This notion is defined as usual. In this context, the state of a task is the program-state restricted to the variables of and the location in this task.

Example 2.0. This example illustrates the ADA-CF subset 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 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 program in [1, § 9.12] (n denotes an arbitrary positive integer constant):

```
begin
  task producer
    array (1..n) of int vec1; int i;
```

² That the presence or absence of entry queues has no influence on the partial correctness semantics of the subset is proved in [25]; see also Section 9.

```

begin  $i := 1$ —and initialize vec1 to some arbitrary values
  while  $i \leq n$  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  $j \leq n$  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  $terms \neq 2$  do
    select
       $count < 100$ :
        accept put( $x$ ) do  $pool(in \bmod 100) := x$  endaccept;
         $in := in + 1$ ;  $count := count + 1$ 
      or  $count > 0$ :
        accept get( $\# y$ ) do  $y := pool(out \bmod 100)$  endaccept;
         $out := out + 1$ ;  $count := count - 1$ 
      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.

In Section 4 we will show that this program satisfies

```

{true}
begin task producer || task consumer || task buffer end
{ $\forall i = 1..n \text{ } vec1(i) = vec2(i)$ }.

```

Here, $\{p\} S \{q\}$ expresses, as usual, that any computation of S starting in a state that satisfies the assertion p , terminates in a state that satisfies q or does not terminate at all. I.e., it expresses partial correctness.

The assertion-language in which p and q are expressed is an ordinary first-order one, with the usual logical connectives and quantifiers. The non-logical symbols of the assertion-language, are the functions and predicates that are used in the program in question. The proof system will be formally developed for ADA-CF programs though. Hence, it will deal with the integer data type only (and so must the corresponding assertion language).

3. The proof system

The proof system is similarly structured as the one of Owicki in [24] or the CSP-system of Apt et al. [5]: 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 [24] and the cooperation test of [5]. 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 Ashcroft in [6] and subsequently used for Owicki's 'general programming language', GPL, in [24] and for CSP in [5]. In the case of GPL and CSP, it is a rather trivial problem what consistency test has 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:

$$\{p\} \text{ call } T.e(\vec{t} \neq \vec{x}) \{q\},$$

provided $\text{FV}(p) \cap \{\vec{x}\} = \emptyset$.

The arguments \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 executing 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 in the sequel.

R1. accept:

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

provided $\{\vec{u}, \vec{v}\} \cap \text{FV}(p, q) = \emptyset$.

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:

$$\{p\} \text{ null } \{p\}.$$

A3. assignment:

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

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

R2. select:

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

Remember that waiting (until a rendezvous is possible) does not influence the truth-value of partial correctness properties.

R3. if:

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

R4. while:

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

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:

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

In the sequel, a task will often be identified with its body, in the sense that $\{p\} \text{task } T \{q\}$ or $\{p\} T \{q\}$ will be written where $\{p\} \text{begin } S \text{ end } \{q\}$ (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.³ 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:

- (1) $p \rightarrow pre(S)$ and $post(S) \rightarrow q$,
- (2) $pre(S) \rightarrow pre(R)$ and $post(R) \rightarrow post(S)$ if $S \equiv \text{begin } R \text{ end}$,
- (3) $pre(R) \rightarrow post(R)$ if $R \equiv \text{null}$,
- (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 $R \equiv \text{if } b \text{ then } R' \text{ else } R'' \text{ endif}$,
- (6) $pre(R) \wedge b \rightarrow pre(R')$, $post(R') \rightarrow pre(R)$ and $pre(R) \wedge \neg b \rightarrow post(R)$ if $R \equiv \text{while } b \text{ do } R' \text{ endwhile}$,
- (7) $pre(R) \wedge b_i \rightarrow pre(R_i)$ and $post(R_i) \rightarrow post(R)$ for $i = 1..n$ if $R \equiv \text{select } b_1 : R_1 \text{ or } \dots \text{ or } b_n : R_n \text{ endselect}$,
- (8) $FV(pre(R), post(R)) \cap \{\bar{u}, \bar{v}\} = \emptyset$ if $R \equiv \text{accept } e(\bar{u} \neq \bar{v}) \text{ do } R' \text{ endaccept}$,
- (9) $FV(pre(R)) \cap \{\bar{x}\} = \emptyset$ if $R \equiv \text{call } T.e(\bar{i} \neq \bar{x})$,
- (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.

³ The notion of bracketing will be explained later on. Until then, the reader may safely ignore this requirement.

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 that 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 precise statement of the cooperation test plausible:

First formulation of cooperation of ADA-CF proofs.

The proof outlines of $\{p_i\}$ task $T_i \{q_i\}, \dots, \{p_n\}$ task $T_n \{q_n\}$ cooperate if

(1) for any 'matching communication pair'

$$C \equiv \text{call } T_j.e(\vec{i} \# \vec{x})$$

and

$$A \equiv \text{accept } e(\vec{u} \# \vec{v}) \text{ do } S \text{ endaccept } (A \text{ within } T_j),$$

the formula $\{pre(C) \wedge pre(A)\}C \parallel A\{post(C) \wedge post(A)\}$ holds, whenever C and A become engaged in a 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 appear 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 of how formulae like $\{p\} C \parallel A \{q\}$ are proved, is deferred for a while.

The second clause forces independence of the proof outlines: No proof outline may refer to variables of other tasks; hence, a proof outline cannot be invalidated by actions elsewhere and, consequently, no 'interference freedom' need be established [24]. 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,

$$\{true\} \text{ begin task } T \parallel \text{task } T' \text{ end } \{x = y\}. \quad (3.2)$$

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

<pre>task T int x; {x = z} begin {x = z} call T'.a(x){x = z} end {x = z}</pre>	<pre>task T' entry a; int y; {true} begin {true} accept a(u) do {u = z} y := u {y = z} endaccept {y = z} end {y = z}</pre>
--	--

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 $\{x = z \wedge true\} C \parallel A \{x = z \wedge y = z\}$ (C denotes the entry call in T and A the accept in T'). According to the semantics of a rendezvous (cf. Section 2), $C \parallel A$ is 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:⁴

$$\begin{aligned} &\{x = z\} \{x = z \wedge x = x\} u := x \{x = z \wedge u = x\} \\ &\{x = z \wedge u = z \wedge u = u\} y := u \{x = z \wedge u = z \wedge y = u\} \{x = z \wedge y = z\}. \end{aligned}$$

So the outlines may be combined:

$$\{x = z \wedge true\} \text{ begin task } T \parallel \text{task } T' \text{ end } \{x = z \wedge y = z\}.$$

⁴ Juxtaposition of two assertions in a proof outline denotes implication of the rightmost assertion by the leftmost.

Application of the consequence rule yields

$$\{x = z\} \text{begin task } T \parallel \text{task } T' \text{ end } \{x = y\}.$$

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

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

R8. substitution:

$$\frac{\{p\} S \{q\}}{\{p[t/z]\} S \{q\}}$$

provided $z \notin \text{FV}(S, q)$.

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 cannot be inferred from the program text alone but has to be semantically characterized. Consequently, it raises the dual problem of characterizing the absence of a rendezvous.

A rendezvous between A and C can only occur if there is an execution of the program that reaches a state in which control resides both at 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 $\text{pre}(C) \wedge \text{pre}(A) \rightarrow \text{false}$.

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

$$\{\text{true}\} \text{begin task } T \parallel \text{task } T' \parallel \text{task } T'' \text{ end } \{x = 2\}$$

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 of 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*, G_I , which may also carry other global information needed for a proof. G_I 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 explicitly 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 [5] and [24].

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 could be at the first call in T and simultaneously at the second accept in T' , the program-state would satisfy $p \wedge q \wedge I$ (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, G_I , must hold is restricted as in

[5] by introducing a *bracketing* for each program; the updatings of G_I -variables are then confined to *bracketed sections* (in which G_I consequently need not hold) which are associated with entry calls and accepts as these are the only statements at which the execution of different tasks synchronize:

First definition of bracketing

A task is called *bracketed* if the brackets ' \langle ' and ' \rangle ' are interspersed in its text, so that

- (1) for each *bracketed section* $\langle S \rangle$, S is of the form

$$S'; \text{ call } T.a(\vec{e} \# \vec{x}); S'' \quad \text{or} \quad \text{accept } b(\vec{u} \# \vec{v}) \text{ do } S' \text{ endaccept}$$

(where S' and S'' update the variables of the global invariant and may be null-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 $\{p_i\}$ **task** $T_i\{q_i\}$ ($i = 1..n$) *cooperate* w.r.t. G_I if

- (1) for any syntactically matching pair $\langle C \rangle$ and $\langle A \rangle$, where

$$C \equiv S_1; \text{ call } T_j.e(\vec{i} \# \vec{x}); S_2 \quad \text{and} \quad A \equiv \text{accept } e(\vec{u} \# \vec{v}) \text{ do } S \text{ endaccept}$$

(A within T_j), the formula

$$\{pre(C) \wedge pre(A) \wedge G_I\} C \parallel A \{post(C) \wedge post(A) \wedge G_I\}$$

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

R9. parcom:

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

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

The reason not to allow formal or actual parameters to appear free in G_I , is to prevent some additional complications to occur: Allowing actual parameters, would introduce an aliasing problem, as a variable of G_I then could be updated under a

different name. Allowing formal parameters, would complicate the rendezvous rule, R11, to be developed below.⁵ 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.

In the sequel we will refer to proofs for a program **begin task** $T_1 \parallel \dots \parallel T_n$ **end**, which *start in an assertion* p , meaning that there exist a set of valid proof outlines T'_1, \dots, T'_n , which cooperate w.r.t. some G1 and such that

$$p \rightarrow \text{pre}(T'_1) \wedge \dots \wedge \text{pre}(T'_n) \wedge G1.$$

Finally, a rule is needed to remove auxiliary variables from a program again (this rule is similar to the ones in [5] and in [24]):

R10. AV: Let AVAR denote a set of variables, elements of which appear in S' only in assignments of the form $x := t$ with $x \in \text{AVAR}$ and t any expression. Then

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

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

Example 3.4. Now, the formula

$$\{\text{true}\} \text{begin task } T \parallel \text{task } T' \parallel \text{task } T'' \text{ end } \{x = 2\} \quad (3.5)$$

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 in providing the missing details.

```

    task T int i;
    {i = 0} begin
      {i = 0} <call T'.a(1); i := 1>;
      {i = 1} <call T''.b( )>
    end {i = 1}

    task T'' entry b; int k;
    {k = 0} begin
      {k = 0} <accept b( ) do k := 1; null endaccept>;
      {k = 1} <call T'.a(2)>
    end {k = 1}

```

⁵ These two restrictions do not impair completeness of the system, because such actual and formal parameters may always be assigned to auxiliary variables.

```

    task  $T'$  entry  $a : \text{int } x, j;$ 
    { $j = 0$ } begin
      { $j = 0$ } (accept  $a(u)$  do { $j = 0 \wedge u = 1$ }  $x := u; j := 1$  endaccept);
      { $j = 1 \wedge x = 1$ } (accept  $a(v)$  do { $j = 1 \wedge x = 1 \wedge v = 2$ }  $x := v$  endaccept)
      end { $j = 1 \wedge x = 2$ }
  
```

In this proof, the following global invariant, G_I , is used:

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

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 G_I , $i = 0 \wedge j = 1 \wedge x = 1 \wedge (j = 1 \leftrightarrow i = 1) \wedge (k = 1 \rightarrow j = 1)$ 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

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

should be completed. This is trivial:

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

(2) The second call in T with the accept in T'' . To complete $\{i = 1 \wedge k = 0 \wedge G_I\} k := 1; \text{null } \{i = 1 \wedge k = 1 \wedge G_I\}$, is even more trivial:

$$\{i = 1 \wedge k = 0 \wedge G_I\} k := 1 \{i = 1 \wedge k = 1 \wedge j = 1\} \text{null } \{i = 1 \wedge k = 1 \wedge G_I\}.$$

Notice that here, the implication $i = 1 \rightarrow j = 1$, part of G_I , is needed; otherwise the second part of G_I , $k = 1 \rightarrow j = 1$, 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

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

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

$$\{i = 0 \wedge j = 0 \wedge k = 0 \wedge G_I\} \text{begin task } T \parallel \text{task } T' \parallel \text{task } T'' \text{ end } \{x = 2\}.$$

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,

G_1 only relates locations in different tasks with each other, in general G_1 also carries other state information; see for instance the example proof in Section 4.

The above development mirrors the development of the CSP-system in [5]. 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 $\{p\}C \parallel A\{q\}$ -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):

<pre> task T {true} begin <call T'.a()>; end {true} task T'' entry b; int y; {true} begin <accept b(x) do {x = 0} y := x endaccept> end {y = 0} </pre>	<pre> task T' entry a; int h; {h = 0} begin {h = 0} <accept a() do h := 1; {h = 1} <call T''.b(1)>; h := 0 endaccept> {h = 0} end {true} GI $\equiv h = 0$ </pre>
--	--

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

$\{true\} \text{ begin task } T \parallel \text{task } T' \parallel \text{task } T'' \text{ end } \{y = 0\}.$

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 G_1 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 G_1 should also hold when such inner calls or accepts are reached. As these appear within the bracketed

section of the outer accept in which G_I 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.

G_I 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 ' \langle ' and ' \rangle ' are interspersed in its text, so that

- (1) for each *bracketed section*, $\langle S \rangle$, S is of the form
 - (a) $S'; \text{ call } T.a(\tilde{e} \# \tilde{x}); S''$,
 - (b) $\text{ accept } b(\tilde{u} \# \tilde{v}) \text{ do } S' \text{ or}$
 - (c) $S'' \text{ endaccept}$;

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 1(a) of this definition has remained the same; the other clauses have changed. Clearly, the intention of this change is that G_I must be shown to hold again, whenever S' (in clause 1(b)) 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 $\{p\} C \parallel A \{q\}$ -type formula.)

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

$$\{h = 0\} \langle \text{accept } a(\) \text{ do } h := 1 \rangle \{h = 1\} \\ \langle \text{call } T''.b(1); \rangle \langle h := 0 \text{ endaccept} \rangle \{h = 0\}.$$

But now it becomes immediately clear that $G_I \equiv h = 0$ is no longer a global invariant, because for this accept (A) and the call (C) in T the cooperation test (or rather the proof of $\{true \wedge h = 0 \wedge G_I\} C \parallel A \{true \wedge h = 0 \wedge G_I\}$) 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. G_I if

- (1) for any syntactically matching pair, $\langle C \rangle$ and $\langle A \rangle$, where

$$C \equiv S_1; \text{ call } T_j.a(\tilde{e} \# \tilde{x}); S_2 \quad \text{and}$$

$$A \equiv \text{ accept } a(\tilde{u} \# \tilde{v}) \text{ do } S'_1; S; \langle S'_2 \text{ endaccept } (A \text{ within } T_j),$$

the formula

$$\{pre(C) \wedge pre(A) \wedge GI\} C \parallel A \{post(C) \wedge post(A) \wedge GI\}$$

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:

$$\begin{aligned} &\{p_1\} \langle S_1; \{\bar{p}_1\} \text{ call } T'.a(\bar{e} \# \bar{x}) \{\bar{q}_1\}; S_2 \rangle \{q_1\}, \\ &\{p_2\} \langle \text{accept } a(\bar{u} \# \bar{v}) \text{ do } \{p'_2\} S'_1; \{\bar{p}_2\} S \{\bar{q}_2\}; \{S'_2\} \{q'_2\} \text{ endaccept} \rangle \{q_2\}. \end{aligned} \quad (3.9)$$

These bracketed sections are denoted by $\langle C \rangle$ and $\langle A \rangle$ respectively (the call is part of a task T).

The question is, how to prove

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

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:

$$\begin{aligned} &\{p_1 \wedge p_2 \wedge GI\} \\ &\quad S_1; \bar{u}, \bar{v} := \bar{e}, \bar{x}; S'_1; \{p \wedge GI\} S \{q \wedge GI\}; S'_2; \bar{x} := \bar{v}; S_2 \\ &\{q_1 \wedge q_2 \wedge GI\} \end{aligned} \quad (3.11)$$

($\bar{x} := \bar{v}$ denotes the simultaneous assignment of the variables in the list \bar{v} to the corresponding variables in \bar{x} ; likewise for the assignment $\bar{u}, \bar{v} := \bar{e}, \bar{x}$). 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 , 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. Hence, 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 proof outlines of the respective tasks are correct (and that GI holds again after updating its variables). Formulation

of this test essentially hinges on the assumption that such task-states are characterized by the appropriate *unique* pre-assertions of the corresponding proof outlines and G1: The set of cooperating conditions is determined purely by the syntactic structure of the program and in no way depends on a particular program proof. 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 task-states in which control is at C' . Consequently, the cooperation test—using the approach of (3.11) for proving (3.10)—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' . In its turn, A' may contain an inner call too and still more checks have to be generated (although the total number of tests remains finite).⁶

These phenomena 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 need be constructed; the cooperation test should not require additional ones. This means that the proof outline of the body of such an 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, restrictions must be imposed on the actual parameters allowed; see also [3]. 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

$$(a) \text{ FV}(\vec{e}) \cap \{\vec{x}\} = \emptyset, \{\vec{u}\} \cap \{\vec{v}\} = \emptyset, (\text{FV}(S) \cup \{\vec{u}, \vec{v}\}) \cap (\text{FV}(\vec{e}) \cup \{\vec{x}\}) = \emptyset,$$

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

then

$$(1) \quad \{p\} S \{q\} \Rightarrow \{p[\cdot]\} S[\cdot] \{q[\cdot]\},$$

provided $\text{FV}(q) \cap \{\vec{x}\} = \emptyset$ ($[\cdot]$ denotes the variable substitution $[\vec{e}, \vec{x}/\vec{u}, \vec{v}]$),

$$(2) \quad \{p\} S[\cdot] \{q\} \Rightarrow \{p\} \vec{u}, \vec{v} := \vec{e}, \vec{x}; S; \vec{x} := \vec{v} \{q\}$$

provided $\text{FV}(p, q) \cap \{\vec{u}, \vec{v}\} = \emptyset$.

⁶ Note that these problems originate only in the way in which the cooperating conditions are defined and are, for instance, not influenced by our refinement of bracketing.

We do not prove this theorem here, but instead refer the reader to [15]; it is not a striking result and an analogous one is, e.g., embodied in E.R. Olderog's rule 26 in [3].

The restrictions 3.12(a) and (b) correspond precisely to the restrictions R1, ..., 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 one leaves the pre-assertion, p , valid).

We proceed with an informal deduction of the rendezvous-rule, to be used in proving the formulae in the cooperation test.

Theorem 3.12(2) suggests that instead of completing (3.11) one might try and complete

$$\begin{aligned} & \{p_1 \wedge p_2 \wedge G1\} \\ & S_1; S'_1[\cdot]; \{p[\cdot] \wedge G1\} S[\cdot] \{q[\cdot] \wedge G1\}; S'_2[\cdot]; S_2 \\ & \{q_1 \wedge q_2 \wedge G1\} \end{aligned} \quad (3.13)$$

(remember that $FV(p_1, q_1, p_2, q_2, G1, S_1, S_2) \cap \{\bar{u}, \bar{v}\} = \emptyset$; p and q , still have to be determined).

Now consider the proof outline in (3.9) for $\{\bar{p}_2\} S \{\bar{q}_2\}$. 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 $S[\cdot]$, because $FV(\bar{p}_1) \cap \{\bar{x}\} = \emptyset$ (this explains the role of the restriction in the call-axiom A1). $G1$, too, may be assumed to be invariant over S and hence over $S[\cdot]$ (remember, $G1$ 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 $G1$, an assertion such as 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 [15]. Consequently, formula (3.13) can be written as

$$\begin{aligned} & \{p_1 \wedge p_2 \wedge G1\} \\ & S_1; S'_1[\cdot]; \{\bar{p}_1 \wedge \bar{p}_2[\cdot] \wedge G1\} S[\cdot] \{\bar{p}_1 \wedge \bar{q}_2[\cdot] \wedge G1\} S'_2[\cdot]; S_2 \\ & \{q_1 \wedge q_2 \wedge G1\}. \end{aligned}$$

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:

$$\frac{\{pre(C) \wedge pre(A) \wedge G1\} S_1; S'_1[\cdot] \{pre('call') \wedge pre(S)[\cdot] \wedge G1\} \quad \{pre('call') \wedge post(S)[\cdot] \wedge G1\} S'_2[\cdot]; S_2 \{post(C) \wedge post(A) \wedge G1\}}{\{pre(C) \wedge pre(A) \wedge G1\} C \parallel A \{post(C) \wedge post(A) \wedge G1\}}$$

where

$$\begin{aligned} C &\equiv S_1; \text{call } T'.a(\vec{e} \# \vec{x}); S_2 \text{ (within a task } T), \\ A &\equiv \text{accept } a(\vec{u} \# \vec{v}) \text{ do } S'_1; S \langle S'_2 \text{ endaccept } (A \text{ within } T'),^7 \\ [\cdot] &\equiv [\vec{e}, \vec{x}/\vec{u}, \vec{v}], \\ \text{'call' denotes the entry call within } C. \end{aligned}$$

Recapitulating, the premises 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 $pre(S)[\cdot]$. If they do, $post(S)[\cdot]$ 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 Section 3, 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 compromise: 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 [15] 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

⁷ Remember that S obeys the clauses R1, ..., R4 of Section 2.

```

{true}
  begin task producer || task consumer || task buffer end
{ $\forall i = 1..n \text{ vec1}(i) = \text{vec2}(i)$ }.

```

For the proof, auxiliary variables are introduced:

- in the producer 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, ' $a \hat{b}$ ' 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 ' $\text{pool}(x \circ y)$ ' is defined as follows (pool is a variable of type **array** (0..99) of int):

$$\text{pool}(x \circ y) = \begin{cases} \text{pool}(x \bmod 100..(y-1) \bmod 100), & \text{if } x \bmod 100 \leq y \bmod 100, \\ \text{pool}(x \bmod 100..99) \hat{\text{pool}}(0..(y-1) \bmod 100), & \text{otherwise.} \end{cases}$$

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 h1;
{h1 =  $\Lambda$ } begin i := 1; —and initialize vec1 to some arbitrary values
  {h1 = vec1(1..i-1)  $\wedge$  i  $\leq$  n+1}
  while i  $\leq$  n do {h1 = vec1(1..i-1)  $\wedge$  i  $\leq$  n}
    <h1 := h1  $\hat{\text{vec1}}(i)$ ; {h1 = vec1(1..i)  $\wedge$  i  $\leq$  n}
l1:    call buffer'.put(vec1(i)); {h1 = vec1(1..i)  $\wedge$  i  $\leq$  n}
    i := i+1 {h1 = vec1(1..i-1)  $\wedge$  i  $\leq$  n+1}
  endwhile; {h1 = vec1(1..n)}
l2:    <call buffer'.term( )>
end {h1 = vec1(1..n)} l3:

```

```

task consumer'
  array (1..n) of int vec2; int j; sequence of int h2;
{h2 =  $\Lambda$ } begin j := 1;
  {h2 = vec2(1..j-1)  $\wedge$  j  $\leq$  n+1}
  while j  $\leq$  n do
l4:    <call buffer'.get( $\#$  vec2(j)); h2 := h2  $\hat{\text{vec2}}(j)$ ;
    j := j+1
  endwhile {h2 = vec2(1..n)}
l5:    <call buffer'.term( )>
end {h2 = vec2(1..n)} l6:

```

```

task buffer'
  entry put, get, term;
  array (0..99) of int pool; int in, out, count, terms;
  sequence of int  $\bar{h}_1, \bar{h}_2$ ;
 $\{\bar{h}_1 = \Lambda \wedge \bar{h}_2 = \Lambda\}$ 
  begin in := 0; out := 0; count := 0; terms := 0;
     $\{count = in - out \wedge 0 \leq count \leq 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ in)\} \text{---}\{I\}$ 
    while terms  $\neq$  2 do
 $l_7$ :      select count < 100:  $\{I \wedge count < 100\}$ 
           $\langle \text{accept } put(x) \text{ do } \bar{h}_1 := \bar{h}_1 \hat{=} x; \rangle$ 
             $\{count = in - out \wedge 0 \leq count < 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ in) \hat{=} x\}$ 
 $l_8$ :      pool(in mod 100) := x
             $\{count = in - out \wedge 0 \leq count < 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ (in + 1))\}$ 
           $\langle \text{endaccept} \rangle$ ; in := in + 1; count := count + 1  $\{I\}$ 
        or count > 0:  $\{I \wedge count > 0\}$ 
           $\langle \text{accept } get(\# y) \text{ do}$ 
            y := pool(out mod 100);
             $\langle \bar{h}_2 := \bar{h}_2 \hat{=} y \text{ endaccept} \rangle$ ;
             $\{count = in - out \wedge 0 < count \leq 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool((out + 1) \circ in)\}$ 
            out := out + 1; count := count - 1  $\{I\}$ 
          or true:  $\{I\}$ 
             $\langle \text{accept } term( ) \text{ do} \text{ null } \langle \text{endaccept} \rangle$ ;
            terms := terms + 1  $\{I\}$ 
          endselect  $\{I\}$ 
        endwhile  $\{I\}$ 
    end  $\{\bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ in)\}$ 

```

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) $\{h_1 = vec1(1..i-1) \wedge i \leq n \wedge I \wedge count < 100 \wedge GI\}$
 $h_1 := h_1 \hat{=} vec1(i); \bar{h}_1 := \bar{h}_1 \hat{=} vec1(i)$
 $\{h_1 = vec1(1..i) \wedge i \leq n \wedge count = in - out \wedge$
 $0 \leq count < 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ in) \hat{=} vec1(i) \wedge GI\}$
- (2) $\{h_1 = vec1(1..i) \wedge i \leq n \wedge count = in - out \wedge$
 $0 \leq count < 100 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} pool(out \circ (in + 1)) \wedge GI\}$

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 composition rule can be applied:

$$\begin{aligned} & \{h_1 = \Lambda \wedge \bar{h}_1 = \Lambda \wedge h_2 = \Lambda \wedge \bar{h}_2 = \Lambda \wedge h_1 = \bar{h}_1 \wedge h_2 = \bar{h}_2\} \\ & \quad \text{begin task producer' || task consumer' || task buffer' end} \\ & \{h_1 = \text{vec1}(1..n) \wedge h_2 = \text{vec2}(1..n) \wedge \bar{h}_1 = \bar{h}_2 \hat{pool}(\text{out} \circ \text{in}) \wedge h_1 = \bar{h}_1 \wedge h_2 = \bar{h}_2\}. \end{aligned}$$

The post-assertion can be reduced to “ $\forall i = 1..n \text{vec1}(i) = \text{vec2}(i)$ ” by applying the consequence-rule. Next, the auxiliary variables can be removed. Finally, substituting Λ , the empty sequence, for h_1 , \bar{h}_1 , h_2 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 strengthened.

In this proof, 4 auxiliary variables have been used which seems excessive. The two auxiliary variables in the buffer-task would be rendered unnecessary if the array *pool* could be used in GI. The problem is, that then the variables *in*, *out* and *count* would be needed too. Unfortunately, the bracketed sections cannot be extended so as to encompass all updatings of these variables; at least, not with the current definition of bracketing. On the other hand, it is not difficult to envisage an appropriate generalization of bracketing which would allow one to do just that. As for the other two auxiliary variables, these can obviously be removed (provided the initializations of *i* and *j* are ‘moved’ to the respective pre-assertions). For an actual proof along these lines, without the use of auxiliary variables, the reader is referred to [7]. We stipulate however, that the format of the above proof reflects the way in which proofs have to be structured in general.

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 Lampert’s parlance [19], “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. For such state-descriptions, we will need G_1 to be valid. While updating auxiliary variables that appear free in G_1 , G_1 need not hold. Consequently, we will not be able to derive state-descriptions at every intermediate point in a program. This is an inevitable consequence of the way our proof system works. On the other hand, the completeness proof [15] shows that bracketed sections need only contain assignments to auxiliary variables.

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, cannot 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 the 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 .⁸ Finally, a *frontier of computation* consists of a maximal 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 [19], in which Lamport introduces *location predicates* (these correspond to our location points) into his assertion-language so as to obtain a safety proof system.

⁸ For the connoisseur of CSP it may be interesting to know that for CSP, multi control points degenerate to control points as CSP does not admit calling chains.

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 ' $\langle C \rangle$ '. 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 following:

- (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 to the callee but the rendezvous has not yet terminated; see also [25] 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 $in('C_1'), \dots, in('C_n')$ such that

- (1) ' C_1 ' does not appear within an accept,
- (2) ' C_{i+1} ' appears within an accept that (syntactically) matches 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, \dots, 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 $\langle x_1, \dots, x_n \rangle$ such that

- (1) x_1, \dots, 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 $X^{(1)}, \dots, X^{(n)}$ all be m.c.p.'s. A *frontier of computation* (f.o.c.) for a program P , is a set $\{X^{(1)}, \dots, X^{(n)}\}$, such that

- (1) for each task in P there is a c.p. belonging to it, and
- (2) no two c.p.'s belong to the same task.

See Example 5.5 below, for an example of a f.o.c. 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 applicable). 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 $post(T)$ with a c.p. of the form $after(T)$. 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 clear that with a f.o.c., we should associate the conjunction of G_I and the assertions associated with its constituent c.p.'s. The presence of G_I is quite essential and 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 semantically match, the conjunction can be made to yield *false* (by strengthening G_I if necessary), which—as usual—is interpreted as stating that the m.c.p. cannot be reached in any computation of the program.

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. Conceptually, $at('C')$ is the point at which a task waits for a rendezvous with 'C' to take place: The updating of (auxiliary) variables in the first part of ' C ', indicates just that. This suggests that the pre-assertion of the bracketed section surrounding 'C', $pre('C')$, be associated with $at('C')$. Of course, bracketed sections may contain assignments to 'normal' task-variables, too, and so one may be less than happy with this suggestion. However, it is the only feasible choice as validity of G_I is needed. Hence,

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

(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 the only c.p.'s owned by a frontier task that specify points within bracketed sections, are of the form $at('C')$.⁹

⁹ As an alternative, c.p.'s referencing statements other than calls within bracketed sections might have been prohibited. This is consistent because $at('C')$ and $at('C')$ are apparently identified anyhow. Apart from the fact that the definition of a c.p. would then come to depend on a proof theoretical notion, various definitions in the sequel would become more cumbersome to state, too.

Let x_1, x_2, \dots, 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

$$R(\chi) = A(x_1) \wedge \dots \wedge A(x_n) \wedge GI.$$

Often when specifying f.o.c.'s, we will not bother to define c.p.'s for every task. In such cases, it is tacitly assumed that in each of such tasks, control resides at some arbitrary but specific c.p., obeying the restriction in 5.4(2) if necessary.

One question remains unanswered. Namely, how to include the value of formal parameters in the state descriptions. In fact, we already have provided the answer, 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 completeness of the proof system). This is illustrated in

Example 5.5. Consider the example proof of Section 4. We show that whenever control is at the f.o.c. $\{\langle in(1_1), at(1_8) \rangle, \langle at(1_4) \rangle\}$, $x = vec1(i)$ holds. This is in fact quite trivial: According to Definition 5.4.,

$$R(\{\langle in(1_1), at(1_8) \rangle, \langle at(1_4) \rangle\}) \rightarrow \\ (h_1 = vec1(1..i) \wedge \bar{h}_1 = \bar{h}_2 \hat{pool}(out \circ in) \hat{x} \wedge h_1 = \bar{h}_1).$$

This implies that $vec1(1..i) = \bar{h}_2 \hat{pool}(out \circ in) \hat{x}$ and hence that $x = vec1(i)$.

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

6. Deadlock freedom

As in [5], the concept of *blocking* is introduced. It originated with Owicki in [24]. 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. χ for some program P . Intuitively (and roughly), P cannot proceed in χ when the frontier tasks of the m.c.p.'s 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 χ be a f.o.c. for a program **begin task** $T_1 \parallel \dots \parallel$ **task** T_n **end**. Let x_1, \dots, x_n be the sequence of c.p.'s in χ ; c.p. x_i belonging to task T_i . Furthermore, let T'_1, \dots, T'_t be the sequence of frontier tasks of χ .

Then, χ is a *blocking frontier of computation* (b.f.o.c.) iff

- (1) $\chi \neq \{\langle \text{after}(T_1) \rangle, \dots, \langle \text{after}(T_n) \rangle\}$,
- (2) each x'_k (in frontier task T'_k) is either of the form $\text{after}(T'_k)$ or of the form $\text{at}('S')$, where ' S ' is an entry call or a select, $k = 1..t$,
- (3) if x'_k is of the form $\text{at}(\text{'call } T_i.a(\cdot \cdot \cdot)')$ then $x'_i \neq \text{after}(T_i)$ ($k = 1..t$).

Clause (1) implies that the program should not have terminated yet and clause (2) indicates that a task can always proceed if it is not at an accept or entry call. Clause (3) 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** \dots **or** $b_n : S_n$ **endselect** and let $I \subseteq \{1..n\}$. Then

$$\text{CB}(\text{at}('S'), I) = \bigwedge \{ \neg b_i : i \in I \} \wedge \bigvee \{ b_i : i \in \{1..n\} \}.$$

The predicate expresses that the branches whose indices appear in I are closed and that at least one of the (other) branches is open.

Definition 6.1. Let χ be a b.f.o.c. The sequence consisting of c.p.'s in χ of the form $\text{at}('C')$, respectively $\text{at}('S')$, (' C ' an entry call, ' S ' a select) are denoted by x_1, \dots, x_t , respectively y_1, \dots, y_m . For each y_i , define a set $I(y_i)$ by

$$k \in I(y_i) \text{ iff the } k\text{th branch of the select } y_i \text{ is for an entry called by one of the } x_j\text{'s.}$$

The *blocking assertion* for χ is defined as

$$B(\chi) = \text{CB}(y_1, I(y_1)) \wedge \dots \wedge \text{CB}(y_m, I(y_m)).$$

Note that for a b.f.o.c. χ in which no task is at a select, $B(\chi)$ is vacuously true. 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. an assertion p , iff proof outlines can be constructed starting in p , such that for each b.f.o.c. χ for P ,¹⁰

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

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

¹⁰ By definition, χ satisfies the additional assumption of Definition 5.4(2).

It is a simple exercise to show that the b.f.o.c.'s are the f.o.c.'s of the form $\{\langle at(1_i) \rangle, \langle at(1_j) \rangle, \langle at(1_7) \rangle\}$ for $i = 1, 2, 3$ and $j = 4, 5, 6$ (1_3 and 1_6 denote the points after the task bodies, i.e. $at(1_3) = \text{after}(\text{producer}')$ and $at(1_6) = \text{after}(\text{consumer}')$). 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.

$$(1) \chi = \{\langle at(1_1) \rangle, \langle at(1_4) \rangle, \langle at(1_7) \rangle\}:$$

$$B(\chi) = \text{count} \geq 100 \wedge \text{count} \leq 0 \wedge \text{true},$$

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.

$$(2) \chi = \{\langle at(1_3) \rangle, \langle at(1_6) \rangle, \langle at(1_7) \rangle\}:$$

$$B(\chi) = \text{count} < 100 \vee \text{count} > 0 \vee \text{true},$$

$$R(\chi) = h_1 = \text{vec1}(1..n) \wedge h_2 = \text{vec2}(1..n) \wedge \text{count} = \text{in-out} \wedge 0 \leq \text{count} \leq 100 \wedge$$

$$\text{terms} \neq 2 \wedge \bar{h}_1 = \bar{h}_2 \hat{=} \text{pool}(\text{out} \circ \text{in}) \wedge h_1 = \bar{h}_1 \wedge h_2 = \bar{h}_2.$$

So, $B(\chi) \wedge R(\chi)$, 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 ;

$\{h_1 = 1 \wedge k_1 = 0\}$

begin

endwhile $\{h_1 = \text{vec1}(1..n) \wedge k_1 = 0\}$

$\langle \text{call } \text{buffer}'.\text{term}(\) ; k_1 := 1 \rangle$

end $\{h_1 = \text{vec1}(1..n) \wedge k_1 = 1\}$

task buffer'

$\{I \wedge \text{terms} \neq 2\}$

select

or true: $\{I \wedge \text{terms} \neq 2\}$

$\langle \text{accept } \text{term}(\) \text{ do} \text{null};$

$\langle \text{terms} := \text{terms} + 1 \text{ endaccept} \rangle \{I\}$

end $\{I \wedge \text{terms} = 2\}$

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

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

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

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

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

This example clearly shows that to prove deadlock freedom, in general, the proof outlines have to be stronger than 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 [25], we assume that execution of a program is *just* in the sense that if execution of a task can proceed, it will proceed in finite time.

Obviously, the only source of non-termination is the while statement. The technique to prove termination of a while statement is well-known (cf. [3]): Find a quantity which decreases with every iteration, but cannot decrease indefinitely. This is embodied in the following rule, taken from [3], which replaces the older rule for while statements, R4:

R4'. while:

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

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 (this is the sole reason that makes it possible to construct task proofs in isolation). Given these assertions, a component proof is constructed as for a sequential program; i.e., one has complete information about the result of executing the task's statements.

The entry queues of full ADA induce a fairness constraint on the possible executions of a program. Such queues enforce a specific discipline of accepting entry calls

within select-statements, with the consequence that [25] “no task can wait forever on a call for some entry, e , while infinitely many other calls for e are accepted” (see also Section 9). Consequently, one can show that full ADA admits so-called *unbounded nondeterminism* [4], which implies that the above approach to termination does not suffice for full ADA.¹¹ This matter is, however, too technical for the current paper. We stress the fact, that it can be dealt with routinely, by adapting any of the rules for fair termination to the environment of distributed computing.

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 which has already terminated (or is 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:

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

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. χ of the form

$$\{\langle x_1, \dots, x_n, \text{at}(\text{'call } T.a(\dots) \text{'}) \rangle, \langle \text{after}(T) \rangle\},$$

the formula $R(\chi)$ yields *false*.¹²

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.¹³ With every iteration, either a value is received or transmitted, 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 *terms* increases. Hence, the quantity

$$2n + 2 - |\bar{h}_1| - |\bar{h}_2| - \text{terms}$$

¹¹ Nissim Francez suggested that we investigate this possibility.

¹² Note that $R(\chi)$ is indeed defined.

¹³ Observe that through the *terms*-variable, termination depends on the global state.

($|\cdot|$ denotes the number of values making up its argument) would be a likely candidate to prove termination from. As $G1 \rightarrow h_1 = \bar{h}_1$ and $h_1 = \text{vec } l(1: i)$ for some $i \leq n$ by the proof outline of *producer*, $|\bar{h}_1| \leq n$ holds; similarly, $|\bar{h}_2| \leq n$ holds. This shows that the quantity is bounded below by 0. Correspondingly, if I denotes the loop invariant in the proof outline in section 4, the new parameterized one will be:

$$I'(m) = I \wedge (m = 2n + 2 - |\bar{h}_1| - |\bar{h}_2| - \text{terms}) \wedge (|\bar{h}_1| \leq n \wedge |\bar{h}_2| \leq n) \wedge \\ \text{terms} = |\{i: |\bar{h}_i| = n, i = 1..2\}|.$$

The last conjunct is necessary for showing that $I'(m+1) \rightarrow \text{terms} \neq 2$, the penultimate one for showing that $I'(0) \rightarrow \text{terms} = 2$. 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 tasks 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.

$$\chi = \{\langle at(1_1) \rangle, \langle after(buffer') \rangle\}$$

should not be reachable. Using the proof outline of *buffer'* strengthened as above, we get

$R(\chi) \rightarrow h_1 = \text{vec } l(1..i-1) \wedge i \leq n \wedge \text{terms} = 2 \wedge \text{terms} = |\{i: |\bar{h}_i| = n, i = 1..2\}| \wedge h_1 = \bar{h}_1$ which implies *false*. Reachability of $\{\langle at(1_4) \rangle, \langle after(buffer') \rangle\}$ is treated completely analogously. To show that $\{\langle at(1_2) \rangle, \langle after(buffer') \rangle\}$ and $\{\langle at(1_5) \rangle, \langle after(buffer') \rangle\}$ 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 [9]. 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 nonexistent successor.

As the values of '*this*' and '*succ*' are syntactically determined, no changes to 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 [12]:

do $c_1 : S_1 \square \dots \square c_n : S_n$ **od**,

where c_1, \dots, c_n are boolean expressions, guarding the ADA-CF statements S_1, \dots, 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 executing 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 ($i = 1..n$).

8.1. Description of the algorithm and its implementation

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 element 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*(2)). 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 assignment 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 \oplus and \ominus denote the union and the splitting of bags (no values are thrown away); $\llbracket \cdot \cdot \cdot \rrbracket$ is our bag-constructor and \emptyset 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 chosen during each iteration of the guarded loop, forces the task-array to function as a priority queue rather than as a sorter.

8.2. Correctness proof

Consider the program

```
begin task driver || task sort(1..n) end.
```

We want to derive for this program, the safety assertion

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

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 minimal 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 and have not yet been returned. In the proof outline of the driver-task only one auxiliary variable is introduced, *sent₀*, 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 $B = \emptyset$):

$$L \equiv kept = \llbracket here(i) : i = 1..|kept| \rrbracket \wedge len = |kept| \wedge here(1) \leq \min(sent) \wedge rest = |sent| \geq 0,$$

$$R \equiv (len = 0 \rightarrow rest = 0) \wedge rest \leq n - this \wedge 0 \leq len \leq 1.$$

task driver'

int *x*; **bag of int** *bag*, *sent₀*;

{*sent₀* = \emptyset }

begin *bag* := \emptyset ;

{*bag* = *sent₀* \wedge $0 \leq |bag| \leq n$ }—the loop-invariant

do $|bag| < n$:

x := ?; {*bag* = *sent₀* \wedge $0 \leq |bag| < n$ }

l₀: <call *sort*(1).put(*x*); *sent₀* := *sent₀* \oplus [*x*];>

{*bag* = *sent₀* \oplus [*x*] \wedge $0 \leq |bag| < n$ }

bag := *bag* \oplus [*x*]

\square $|bag| > 0$: {*bag* = *sent₀* \wedge $0 < |bag| \leq n$ }

m₀: <call *sort*(1).get($\neq x$); *sent₀* := *sent₀* \ominus [*x*];>

{*bag* = *sent₀* \oplus [*x*] \wedge $0 < |bag| \leq n \wedge x = \min(bag)$ }

bag := *bag* \ominus [*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 =  $\emptyset$ }
  begin rest := 0; len := 0;
    { $L \wedge R$ }
    while true do
      k:    select true: { $L \wedge R$ }
        {accept put(u) do} { $L \wedge R$ }
          len := len + 1; here(len) := u
          {( $L \wedge R$ )[len - 1/len]  $\wedge$  here(len) = u}
          {kept := kept  $\oplus$  [u] endaccept; }
          { $L \wedge (\text{len} \neq 2 \rightarrow R) \wedge (\text{len} = 2 \rightarrow R[\text{len} - 1/\text{len}] \wedge \text{rest} < n - \text{this})$ }
          if len = 2 then
            if here(2) < here(1) then
              { $L \wedge R[\text{len} - 1/\text{len}] \wedge \text{rest} < n - \text{this} \wedge \text{len} = 2 \wedge \text{here}(2) < \text{here}(1)$ }
              temp := here(2); here(2) := here(1);
              {kept = [here(2)]  $\oplus$  [temp]  $\wedge$  len = |kept|  $\wedge$  here(1)  $\leq$  min(sent)  $\wedge$ 
                rest = |sent|  $\geq$  0  $\wedge$   $R[\text{len} - 1/\text{len}] \wedge \text{rest} < n - \text{this} \wedge \text{len} = 2 \wedge$ 
                here(1) = here(2)  $\wedge$  temp < here(2)}
              here(1) := temp
            endif;
            { $L \wedge R[\text{len} - 1/\text{len}] \wedge \text{rest} < n - \text{this} \wedge \text{len} = 2 \wedge \text{here}(1) \leq \text{here}(2)$ }
            l:    {call sort(succ).put(here(2)); kept := kept  $\ominus$  [here(2)];
              sent := sent  $\oplus$  [here(2)];
              {( $L[\text{rest} + 1/\text{rest}] \wedge R$ )[len - 1/len]  $\wedge$  rest < n - this  $\wedge$  len = 2}
              rest := rest + 1; len := 1
            endif [ $L \wedge R$ ]
            or len = 1: { $L \wedge R \wedge \text{len} = 1$ }
              {accept get( $\neq v$ ) do} { $L \wedge R \wedge \text{len} = 1$ }
                v := here(1) { $L \wedge R \wedge \text{len} = 1 \wedge v = \text{here}(1)$ };
                {kept := kept  $\ominus$  [v] endaccept; } { $L[\text{len} - 1/\text{len}] \wedge R \wedge \text{len} = 1$ }
                len := 0; { $L \wedge (\text{rest} = 0 \rightarrow R) \wedge (\text{rest} > 0 \rightarrow R[\text{len} + 1/\text{len}]) \wedge \text{len} = 0$ }
                if rest > 0 then { $L \wedge R[\text{len} + 1/\text{len}] \wedge \text{rest} > 0 \wedge \text{len} = 0$ }
                  m:    {call sort(succ).get( $\neq \text{here}(1)$ ); kept := kept  $\oplus$  [here(1)];
                    sent := sent  $\ominus$  [here(1)];
                    {( $L[\text{rest} - 1/\text{rest}] \wedge R$ )[len + 1/len]  $\wedge$  len = 0  $\wedge$  rest > 0}
                    rest := rest - 1; len := 1
                  endif [ $L \wedge R$ ]
                endselect { $L \wedge R$ }
              endwhile {false}
            end {false}

```

Next, we prove cooperation w.r.t. G1:

- (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:

$$\begin{aligned} & \{L_i \wedge R_i[|len_i - 1/len_i|] \wedge rest_i < n - i \wedge len_i = 2 \wedge here_i(1) \leq here_i(2) \wedge \\ & (L_{i+1} \wedge R_{i+1})[|len_{i+1} - 1/len_{i+1}|] \wedge here_{i+1}(len_{i+1}) = here_i(2) \wedge G1\} \\ & kept_{i+1} := kept_{i+1} \oplus [here_i(2)]; kept_i := kept_i \ominus [here_i(2)]; \\ & sent_i := sent_i \oplus [here_i(2)] \\ & \{(L_i[rest_i + 1/rest_i] \wedge R_i)[|len_i - 1/len_i|] \wedge rest_i < n - i \wedge len_i = 2 \wedge L_{i+1} \wedge \\ & (len_{i+1} \neq 2 \rightarrow R_{i+1}) \wedge (len_{i+1} = 2 \rightarrow R_{i+1}[|len_{i+1} - 1/len_{i+1}|] \wedge \\ & rest_{i+1} < n - i - 1) \wedge G1\}. \end{aligned}$$

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

$$\begin{aligned} & \{bag = sent_0 \wedge 0 < |bag| \leq n \wedge L_1 \wedge R_1 \wedge len_1 = 1 \wedge x = here_1(1) \wedge G1\} \\ & kept_1 := kept_1 \ominus [x]; sent_0 := sent_0 \ominus [x] \\ & \{bag = sent_0 \oplus [x] \wedge 0 < |bag| \leq n \wedge x = \min(bag) \wedge L_1[|len_1 - 1/len_1|] \wedge R_1 \wedge \\ & len = 1 \wedge G1\}. \end{aligned}$$

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

$$\begin{aligned} & bag = sent_0 \wedge kept_1 = [here_1(i) : i = 1..|kept_1|] \wedge len_1 = |kept_1| \wedge \\ & here_1(1) \leq \min(sent_1) \wedge len_1 = 1 \wedge x = here_1(1) \wedge sent_0 = sent_1 \oplus kept_1. \end{aligned}$$

The other cooperation tests are left to the reader.

Hence, the proof outlines cooperate and can be combined so as to yield

$$R(\{\langle after(m_0) \rangle\} \rightarrow bag = sent_0 \oplus [x] \wedge 0 < |bag| \leq n \wedge x = \min(bag) \wedge G1),$$

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 sources of failure are the entry calls in *sort*(*n*). However, it is easy to show that these can never be reached, as:

- (1) $R(\{\langle at(l_n) \rangle\}) \rightarrow rest_n < n - n \wedge rest_n \geq 0$, and
- (2) $R(\{\langle at(m_n) \rangle\}) \rightarrow rest_n \leq n - n \wedge rest_n > 0$.

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

$$\{\langle at(x_0) \rangle, \langle at(x_1) \rangle, \dots, \langle at(x_n) \rangle\},$$

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 $sort(1..i_1)$, $sort(i_1 + 1..i_2)$, \dots , $sort(i_k + 1..n)$ ($1 \leq i_1 < i_2 < \dots < i_k < n$), such that in each segment $sort(i..j)$ the tasks $sort(i), \dots, sort(j-1)$ 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

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

For such f.o.c.'s

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

which implies *false*. Consequently, such b.f.o.c.'s cannot be reached. Next, we should check segments 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 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 full ADA selective wait statement.

The effect of these statements is either not expressible in our assertion language (as for the delay, which 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).

Consequently, it is a rather meaningless exercise to add such extensions, as is illustrated for the conditional entry call:

select *call_st*; *stats* **else** *stats* **endselect**.

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:

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

where C denotes an entry call.

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

9.1. Terminate

A terminate-statement (abbreviated to *termstat*) may appear as the sole statement in a branch of a select; at most one branch may contain a termstat. 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 (cf. [1, § 9.7.1]).

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:

$$\{p\} \text{terminate } \{false\}.$$

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 be implied by the pre-assertion of every termstat in the task-body which may indeed be executed (see [23] for a general discussion of this type of problem). This necessitates a modification of the cooperation test (Definition 3.8), which has to be extended with the following additional clause:

(3) For a select ' S ', define

$$\text{TP}('S') = \begin{cases} b_j & \text{if } b_j \text{ guards the branch containing the termstat,} \\ \text{false} & \text{if there is no such branch.} \end{cases}$$

Then, for each f.o.c. χ of the form $\{\langle x_1 \rangle, \dots, \langle x_n \rangle\}$, 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}, \dots, x_{i_k} is the list of c.p.'s in this f.o.c., referencing selects, then for each $x_l \in \{x_{i_1}, \dots, x_{i_k}\}$, the formula

$$(R(\chi) \wedge \text{TP}(x_{i_1}) \wedge \dots \wedge \text{TP}(x_{i_k}) \wedge \text{pre}(x_l)) \rightarrow \text{post}(T_l),$$

must hold.

Notice, that the above set of *distributed termination f.o.c.'s* (d.t.f.o.c.'s) is particularly 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) $\neg(R(\chi) \wedge B(\chi))$ holds, if χ is not a d.t.f.o.c., or,

(2) $R(\chi) \wedge B(\chi) \rightarrow \text{TP}(x_{i_1}) \wedge \dots \wedge \text{TP}(x_{i_k})$ 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 connoisseur of ADA, perhaps the most noticeable 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 that no particular entry call will wait indefinitely, while arbitrary many (other) calls for the same entry may proceed. 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. An interesting question is whether the notion of fairness as stated here, is weaker than the (seemingly) stronger notion 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 [25] 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 assumption above) is equivalent to 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
begin call T".e(1) end

task T' int c;
begin c := 0;
  while c = 0 do call T".f(# c) endwhile;
  call T".e(2)
end

task T" entry e, f; int x, y;
begin x := 0;
  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:

$$\{true\} \text{ begin task } T \parallel \text{task } T' \parallel \text{task } T'' \text{ end } \{x = 1 \wedge y = 2\}.$$

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 [24]. We have not followed 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 ([10] and [16]) 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.

10. Conclusion and related work

A small concurrency fragment of ADA has been defined. For this fragment, the idea of cooperation between proofs, which captures at the level of proof the simplest form of communication of values between distributed processes, has been extended to deal with a more complicated type of communication: the ADA-rendezvous mechanism. Care has been taken to retain the notion of proof outline as capturing the idea that per control location, one assertion should suffice to characterize all states at such a location. We feel to have obtained a clean and elegant extension of the notion of cooperation and hence to have shown the rendezvous mechanism to have a simple and appealing semantics.

Alternative proof techniques for concurrent ADA programs do exist: In [7] and [21], Barringer and Mearns formulate another adaptation of cooperation to by-and-large the same fragment of ADA. There, the original notion of bracketing (as introduced in [5]) is retained. In [7], nesting of such sections is allowed; consequently, they informally postulate a set of extra conditions that assertions 'within' bracketed sections should obey, in order to counter the problem with nesting as discussed in our paper. In [21], nested bracketed sections are syntactically transformed into non-nested sections, and an appropriate proof rule is provided to handle such transformations.

A second alternative is Schlichting and Schneider's [28], in which the Levin/Gries [20] approach to proving communicating processes correct, is extended to deal, amongst other communication-primitives, with the ADA-rendezvous.

The second part of our paper develops a technique to extract invariance-assertions from (cooperating) proof outlines, which is then applied to deal with absence of deadlock, absence of failure and with distributed termination (absence of deadlock is treated in [7, 21], too).

To extract such invariance-properties, we show how to characterize the state at any frontier of computation. Typically, in such frontiers more than one task participates. This contrasts with the way in which proof outlines are constructed in isolation. At present, we are unsure as to how this phenomenon relates to programming-methodology. Thus, we end up with a question which is interesting, because it does not only concern ADA, but also any communication and synchronization mechanism, such as remote procedure-calls, which allows a task to engage in a communication with another task while still being synchronized with a third one.

Acknowledgment

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 discussions, A. Salwicki for pointing out the difference between 'interleaving semantics' and 'maximum parallelism semantics' and Nissim Francez and Ian Mearns for some valuable remarks. 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 atmosphere in which the initial stage of the research culminating in this paper was carried out.

References

Reference [13] is not cited in the text.

- [1] ADA, *The Programming Language ADA Reference Manual*, American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983, Lecture Notes in Computer Science 155 (Springer, Berlin, 1983).
- [2] G.R. Andrews and F.B. Schneider, Concepts and notations for concurrent programming, *ACM Comput. Surveys* 15(1) (1983) 3-43.
- [3] K.R. Apt, Ten years of Hoare's logic: A survey—Part I, *TOPLAS* 3(4) (1981) 431-484.
- [4] K.R. Apt, Ten years of Hoare's logic: A survey—Part II: Nondeterminism, in: J.W. de Bakker and J. van Leeuwen (Eds.), *Foundations of Computer Science IV. Distributed Systems: Part II, Semantics and Logic*, Mathematical Centre Tracts 159 (Mathematical Centre, Amsterdam, 1982).
- [5] K.R. Apt, N. Francez and W.P. de Roever, A proof system for communicating sequential processes, *TOPLAS* 2(3) (1980) 359-385.
- [6] E.A. Ashcroft, Program verification tableaux, Technical Report CS-76-01, Department of Computer Science, University of Waterloo (1976).
- [7] H.I. Barringer and I. Mearns, Axioms and proof rules for ADA tasks, Technical Report, Department of Computer Science, University of Manchester; also in: *Proc. IEE* 129, Part E, (2) (1982).
- [8] P. Brinch Hansen, The programming language concurrent Pascal, *IEEE Trans. Software Engrg.* 1 (1975) 99-207.
- [9] P. Brinch Hansen, Distributed processes: A concurrent programming concept, *Comm. ACM* 21(11) (1978) 934-941.
- [10] R. Cartwright and D. Oppen, The logic of aliasing, *Acta Informat.* 15 (1981) 365-384.
- [11] R.P. Cook, *Mod—A language for distributed programming, *IEEE Trans. Software Engrg.* 6 (1980) 563-571.
- [12] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

- [13] R.T. Gerth, A proof system for a subset of the concurrency section of ADA, Technical Report RUU-CS-81-17, Department of Computer Science, University of Utrecht (1981).
- [14] R.T. Gerth, W.P. de Roever and M. Roncken, Procedures and concurrency: A study in proof, *Proc. 5th International Symposium on Programming*, Lecture Notes in Computer Science 137 (Springer, Berlin, 1982) 132–163.
- [15] R.T. Gerth, A sound and complete Hoare axiomatization of the ADA rendezvous, Technical Report, Department of Computer Science, University of Utrecht (1984).
- [16] D. Gries and G.M. Levin, Assignment and procedure call proof rules, *TOPLAS* 2(4) (1980) 564–579.
- [17] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* 21(8) (1978) 666–677.
- [18] C.A.R. Hoare, 1980 ACM Turing Award Lecture, *Comm. ACM* 24(2) (1981) 75–84.
- [19] L. Lamport, The ‘Hoare logic’ of concurrent programs, *Acta Informat.* 14 (1980) 21–37.
- [20] G.M. Levin and D. Gries, A proof technique for communicating sequential processes, *Acta Informat.* 15 (1981) 281–302.
- [21] I. Mearns, A denotational semantics for concurrent ADA programs, Ph.D. Thesis, University of Manchester (1983).
- [22] J.G. Mitchell, W. Maybury and R. Sweet, Mesa Language Manual, XEROX, Palo Alto Research Center (1979).
- [23] M.J. O'Donnell, A critique of the foundations of Hoare-style programming logics, *Proc. Logics of Programs Workshop*, Lecture Notes in Computer Science 131 (Springer, Berlin, 1982) 349–374.
- [24] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Informat.* 6 (1976) 319–340.
- [25] A. Pnueli and W. P. de Roever, Rendezvous with ADA—A proof theoretical view, *Proc. ACM ADATEC Conference* (1982).
- [26] M. Roncken, N. van Diepen, M. Kramer and 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).
- [27] A. Salwicki, Critical remarks on MAX model of concurrency, *Proc. Logics of Programming Workshop 1981*, Lecture Notes in Computer Science 131 (Springer, Berlin, 1982) 397–405.
- [28] R.D. Schlichting and F.B. Schneider, Using message passing for distributed programming: Proof rules and disciplines, Technical Report, Carnegie-Mellon University, 1982. A substantially enlarged version is to appear in *TOPLAS*.