RENDEZVOUS WITH ADA - A Proof Theoretical View

Amir Pnueli

Willem P. de Roever

RUU-CS-82-12

July 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

9

RENDEZVOUS WITH ADA - A Proof Theoretical View

Amir Pnueli

Willem P. de Roever

Technical Report RUU-CS-82-12

July 1982

Department of Computer Science

University of Utrecht

P.O. Box 80.002, 3508 TA Utrecht

the Netherlands

Rendezvous with ADA - A Proof Theoretical View

Amir Pnueli
The Weizmann Institute of Science, Rehovot, Israel

Willem P. DeRoever*
University of Utrecht, The Netherlands

Abstract

A fragment of ADA abstracting the communication and synchronization part is studied. An operational semantics for this fragment is given, emphasizing the justice and fairness aspects of the selection mechanisms. An appropriate notion of fairness is shown to be equivalent to the explicit entry-queues proposed in the reference manual. Proof rules for invariance and liveness properties are given and illustrated on an example. The proof rules are based on temporal logic.

Introduction

In this paper we conduct a very preliminary investigation of the concurrency and synchronization aspects of the programming language ADA. Our aims in this investigation are the clarification of the issue of fairness in the execution of ADA tasking mechanism, and a development of temporal-logic based formalism for proving liveness (eventuality) and other temporal properties of ADA programs.

With this in view we study an extremely simplified fragment of ADA, retaining just the constructs which are relevant to tasking and synchronization, and not even all of these. For this fragment we define interleaving operational semantics which models the execution of concurrent tasks by a sequential execution of atomic instructions taken one at a time, from a single task each time. Such modelling of concurrency by interleaving has proved most fruitful in the past and will be shown to be valuable in our present investigation of ADA. In developing this semantics we will show that the concept of entry-queues used in the ADA definition in order to ensure fairness in the selection among tasks waiting on entry-calls for the same entry, is not really necessary. In our definition we will use the more abstract notion of fairness and show that it is equivalent to the one ensured by the queues. Queues in our opinion is a concept more appropriate in a discussion on the implementation level than in a language definition.

Next, we will formulate a very simple invariance principle which will enable us to prove properties of the invariance class [MP1]. We proceed then to define temporal proof principles which are analogous to the ones developed in [MP3] for the shared-variable model of concurrent programs. The principles introduced here, enable proofs of temporal properties of ADA programs. Their utility for such proofs is demonstrated by an example.

We believe that this fragment of semantics and proof-theory, concentrating on the issues of concurrency and communication in ADA would greatly enhance our understanding of these important aspects of the language. Combined with other proof theoretical efforts directed at the intra-task finer structure, it could yield a powerful comprehensive proof methodology for the complete language.

The basic synchronization and communication mechanism in ADA is that of the rendezvous in which one task issues an entry-call while another task reaches an accept statement for the entry named by the caller. This communication mechanism combines and improves on several features existing in previously suggested mechanisms. The actual entry-call, when executed, is similar to the monitor mechanism as introduced in [H] and expanded in [BH]., in that communication of values, is done via parameter transfer between the called and calling task. Also, the caller is suspended until the execution of the accept-body is completed. Similar to CSP [H] and CCS [M], the rendezvous requires coordination of the two tasks, that is, both the caller and called tasks must be actively interested in establishing communication in order for the contact to be established. However, significantly differently from CSP, the naming convention and selection of alternatives is asymmetric between caller and acceptor. The caller has to know the entry name which is restricted to an association with a single task, but the acceptor task need not know by name all its potential callers. In that, the entry concept is similar to that of a CCS channel with the restriction of having only one possible task as acceptor. Another restriction is that while an acceptor may have a selection of accept statements to choose from, being able to pick

* The second author's stay at the Weizmann Institute was made possible by a travel grant of the Netherlands Organization of Pure Research Z.W.O. The research was supported in part by a grant from the Israeli Academy of Science, the Basic Research Foundation.

one for which an entry-call is pending, the caller must issue one entry call at a time. This simplifies the implementation by introducing a tie-breaking asymmetry. It enables the selection to be done locally, at the acceptor's site, while the caller plays a more passive role - placing a request for a call and waiting until it is granted. The possibilities of conditional and timed entry calls introduce some more complications to this straightforward description but do not significantly alter the picture.

The ADA report emphasizes several times that the selection between open alternatives of a selective-wait statement is arbitrary and does not imply any fairness assumptions. The only fairness consideration mentioned is when several tasks issue an entry-call for the same entry. Then, the report states, these requests are queued, using one queue for each entry name, and once an accept statement for that entry is selected, the requests are to be honoured in the order of their arrival.

The Language Fragment

In order to concentrate on the basic essentials of the communication mechanism in ADA we restrict ourselves to a minimal stripped down fragment of the language. This fragment is referred to as ACF for "ADA Communication Fragment".

An ACF program P is a block containing a fixed number of tasks. No shared variables are allowed between tasks. New tasks may not be dynamically created. Except for the entries declared within each task, no other procedures, subprograms or nested blocks are allowed. The statements allowed within a task are: assignment statement, if statement, loop statements, entry calls, conditional entry calls and selective waits. Of the selective wait alternatives we only allow accept-statement and terminate. No delay statements are allowed anywhere. The program in Fig. 1 is an example of an ACF program.

Operational Semantics for ACF

Consider an ACF program P consisting of the tasks $T_1,\ldots,T_m$. Let all the variables declared in all of the tasks be $\bar{y} = (y_1,\ldots,y_n)$ with y_i ranging over $D_i \cup$ <'undef'>. A state in the execution of P has the form:

$$s = \langle (T_1\text{-location}) \wedge (T_2\text{-location}) \wedge \ldots \wedge (T_m\text{-location}); \eta_1,\ldots,\eta_n \rangle$$

$\eta_i \in D_i \cup$ <'undef'> is the current value of the variable y_i in state s. Each T_i-location, $i = 1,\ldots,m$ is a description of the location of the task T_i in its program (task body). It has the general form: T_i at S_i.

In general, S_i is a sequence of statements which are yet to be executed by T_i. It is the empty sequence Λ if T_i has terminated.

We define a succession relation, written $s \to s'$ and called a transition, to denote that a single computational step can lead from s to s'. The relation is defined by cases corresponding to the various types of possible statements:

Assignment Transition:

$$\langle \ldots (T_i \ \underline{\text{at}}\ \bar{y} := f(\bar{y}); S) \wedge \ldots;\ \bar{\eta} \rangle \to \langle \ldots (T_i \ \underline{\text{at}}\ S) \wedge \ldots;\ f(\bar{\eta}) \rangle$$

For convenience we use simultaneous assignments to all of $y_1,\ldots,y_n$.

This succession rule specifies that one possible computation step of the program consists of a single task performing an assignment statement. As a result of this action, T_i moves to the location immediately after tha assignment statement and the value of $f(\bar{\eta})$ is assigned to the variables $\bar{y}$.

Additional rules correspond to the local action of if and loop statements.

If Transition

$$\langle \ldots (T_i \ \underline{\text{at}}\ \underline{\text{if}}\ p(\bar{y})\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{end if}}; S) \wedge \ldots; \bar{\eta} \rangle \to \langle \ldots (T_i \ \underline{\text{at}}\ S_1; S) \wedge \ldots; \bar{\eta} \rangle$$

Provided $p(\bar{\eta})$ = true.

Similarly the 'else' clause may be taken:

$$\langle \ldots (T_i \ \underline{\text{at}}\ \underline{\text{if}}\ p(\bar{y})\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{end if}}; S) \wedge \ldots; \bar{\eta} \rangle \to \langle \ldots (T_i \ \underline{\text{at}}\ S_2; S) \wedge \ldots; \bar{\eta} \rangle$$

Provided $p(\bar{\eta})$ = false.

Loop Transition

$$\langle \ldots (T_i \ \underline{\text{at}}\ \underline{\text{while}}\ c(\bar{y})\ \underline{\text{do}}\ B; S) \wedge \ldots; \bar{\eta} \rangle \to \langle \ldots (T_i \ \underline{\text{at}}\ B;\ \underline{\text{while}}\ c(\bar{y})\ \underline{\text{do}}\ B; S) \wedge \ldots; \bar{\eta} \rangle$$

Provided $c(\bar{\eta})$ = true.

This transition corresponds to the case that the loop condition $c(\bar{y})$ is true fro the current values of the $\bar{y}$ variables. In such a case, the loop's body B is to be performed first, followed by a repeated execution of the loop.

$$\langle \ldots (T_i \ \underline{\text{at}}\ \underline{\text{while}}\ c(\bar{y})\ \underline{\text{do}}\ B; S) \wedge \ldots; \bar{\eta} \rangle \to \langle \ldots (T_i \ \underline{\text{at}}\ S) \wedge \ldots; \bar{\eta} \rangle$$

Provided $c(\bar{\eta})$ = false.

This corresponds to the case that the loop's condition is false, in which case the whole loop statement is skipped.

The above transitions correspond to local operations and involve the movement of a single task at a time. Following are joint transitions which involve simultaneous movement of two tasks at the same time. They are associated with communications. We consider next transition effected by communication:

Rendezvous Transition:

Let e be an entry declared within T_j. Then we have the following transition:

$<\ldots(T_i\ \underline{at}\ e(\overline{u};\overline{v});\ S_i) \wedge \ldots$

$(T_j\ \underline{at}\ \underline{select} \ldots \underline{or}\ \underline{when}\ c(\overline{y}) \rightarrow \underline{accept}\ e(\overline{f}{:}\underline{in};\ g{:}\underline{out});B$

$\underline{end}\ e\ ;\ S_j \ldots$

$\underline{end}\ \underline{select}\ ;\ S) \wedge \ldots ;\ \overline{\eta}> \rightarrow$

$<\ldots(T_i\ \underline{at}\ \underline{rendezvous}\ e\ ;\ S_i) \ldots$

$\ldots(T_j\ \underline{at}\ \overline{f}{:}=\overline{u}\ ;\ B\ ;\ \overline{v}{:}=\overline{g}\ ;\ \underline{end}\ e\ ;\ S_j\ ;\ S) \wedge \ldots ;\ \overline{\eta}>$

Provided $c(\overline{\eta}) = \underline{true}$.

Here $\overline{f}$ and $\overline{g}$ are all the formal parameters of modes in and out respectively. B is the body of the entry e within the selective wait statement. S_j is the sequence of statements to be performed by T_j after the rendezvous is over. Note that the transition places T_i in a special new state 'rendezvous e', and replaces the accept statement in T_i by elaboration including explicit parameter transfer.

The above rendezvous transition corresponds to a simple entry call of task T_i and a selective wait statement at task T_j. Similar rendezvous transitions are also defined for the cases that T_i is at a conditional entry call of the form:

$(T_i\ \underline{at}\ \underline{select}\ e(\overline{u};\overline{v})\ ;\ S_i$

$\underline{else}\ S'_i$

$\underline{end}\ \underline{select}\ ;\ S''_i$

In this case the T_i location descriptor in s' will have the form:

$(T_i\ \underline{at}\ \underline{rendezvous}\ e;\ S_i;\ S''_i).$

Similarly, a rendezvous transition exists for the case that T_j is at a simple accept statement of the form:

$(T_j\ \underline{at}\ \underline{accept}\ e(\overline{f}{:}\underline{in}\ ;\ \overline{g}{:}\underline{out})\ ;\ B\ ;\ S_j)$

Rendezvous situations are terminated by:

Rendezvous Termination Transition

$<\ldots(T_i\ \underline{at}\ \underline{rendezvous}\ e\ ;\ S_i) \wedge \ldots\ (T_j\ \underline{at}\ \underline{end}\ e\ ;\ S_j) \wedge \ldots$

$;\ \overline{\eta} > \rightarrow$

$<\ldots(T_i\ \underline{at}\ S_i) \wedge \ldots\ (T_j\ \underline{at}\ S_j)\ \wedge\ \ldots ;\ \overline{\eta}>$

This transition terminates the rendezvous situation in which T_i is suspended while T_j executes the body of an entry that was called by T_i.

The following transitions correspond to the option of taking the 'else' clause of a select-statement. In a given state s we define e'COUNT(s) to be the number of tasks currently waiting in front of a simple or conditional entry call for the entry e. Then we have the transitions:

Else Transitions:

$<\ldots(T_i\ \underline{at}\ \underline{select}\ e(\overline{u};\overline{v})\ ;\ S_i$

$\underline{else}\ S'_i$

$\underline{end}\ \underline{select}\ ;\ S''_i) \wedge \ldots\ \overline{\eta}> \rightarrow$

$<\ldots(T_i\ \underline{at}\ S'_i\ ;\ S''_i) \wedge \ldots ;\ \overline{\eta}>$

Provided no task is currently in front of an accept statement for e or a selective wait with an open alternative of accepting e.

This transition corresponds to choosing the 'else' clause of a conditional entry call.

$<\ldots(T_j\ \underline{at}\ \underline{select}$

$\underline{when}\ c(y) \rightarrow \underline{accept}\ e(\overline{f};\overline{g}) \ldots$

$\vdots$

$\underline{else}\ S'_j$

$\underline{end}\ \underline{select}\ ;\ S''_j) \wedge \ldots ;\ \overline{\eta}> \rightarrow$

$<\ldots(T_j\ \underline{at}\ S'_j\ ;\ S''_j) \wedge \ldots ;\ \overline{\eta}>$

Provided e'COUNT(s) = 0 for every open alternative e. This means that no other task is waiting on an entry call for any of the open alternatives of T_j.

A special transition allows termination of the complete program.

Termination Transition:

$< \bigwedge_i (T_i\ \underline{at}\ \Lambda) \wedge \bigwedge_j (T_j\ \underline{at}\ \underline{select} \ldots \underline{or}\ \underline{terminate} \ldots)$

$;\ \overline{\eta}> \rightarrow$

$< \bigwedge_i (T_i\ \underline{at}\ \Lambda)\ ;\ \overline{\eta}>.$

Thus, if all tasks that have not terminated yet are waiting at selective-wait statements which contain a 'terminate' alternative, then the whole program is allowed to terminate.

An initialized computation is a sequence of states:

$$\sigma:\ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$$

which satisfies the following conditions:

Proper Initialization

The first state s_0 has the form:

$s_0 = <(T_1\ \underline{at}\ P_1) \wedge \ldots (T_m\ \underline{at}\ P_m)\ ;\ \underline{undef}>$

Thus, initially each T_i is at the beginning of its program P_i, and all variables are uninitialized.

Proper State to State Transition

Every two consecutive states s_i, s_{i+1} in σ, are related by the succession relation defined above

$$s_i \rightarrow s_{i+1}$$

A *legal computation* is any suffix of an initialized computation.

The notion of legal computation enables us to study the behavior of a concurrent program starting at an arbitrary observation instant, not necessarily the initial one.

We are only interested in *maximal* computations, that is, computations which can not be extended. Such computations are either infinite or are finite and end in a state s_k which is terminal, i.e., having no possible successor s' such that $s_k \to s'$.

Justice and Fairness

An essential restriction that has to be imposed on execution sequences is a consequence of the fact that we use interleaving in order to model concurrency. In real concurrency every task will eventually finish the execution of one instruction and inevitably start the execution of the next one. It can be held up only by communication instructions. To model the same behavior by interleaving executions we introduce the notion of *justice* [LPS].

A task T_i is said to *move* during a transition $s \to s'$ if the location description of T_i in s is different from its location description in s'. Given a state s and an entry e we denote by e'COUNT(s) the number of tasks currently waiting on an entry call for e. A task T_i is said to be *enabled* in a state s if one of the following conditions is met:

a) T_i is in front of a local statement, i.e. assignment, if or loop statement.
b) T_i is in front of a selective wait statement with an open alternative accepting the entry e while e'COUNT(s) > 0.
c) T_i is in front of an *end* e statement.
d) T_i is in front of a conditional entry call or a selective wait containing an 'else' clause.

Intuitively, a task is enabled if it is in front of an instruction whose eventual termination depends only on the task itself. In particular, a task waiting in front of an entry call is not considered enabled. This is because for the call to be accepted, a selection of the particular calling task has to be performed by the task potentially accepting this entry call.

A computation σ is defined to be *just* if it is either finite or every task which is continuously enabled from a certain point on in σ, moves infinitely many times in σ.

This captures the notion of eventual movement in each of the tasks. However, it does not guarantee the requirement of honouring different calls for the same entry in their order of arrival. We therefore stipulate also the requirement of *fairness*.

An execution sequence σ is defined *to be fair* if no process T_i may wait forever on an entry-call for the entry e while infinitely many entry calls for e are accepted in σ.

At first appearance this concept seems weaker than the first-in-first-out discipline required in the reference manual.

In the section below we will show that under appropriate restrictions the requirement of fairness is equivalent to the discipline of accepting calling tasks in the order of their arrival.

We therefore define *admissible computations* to be all legal computations which are both just and fair.

Fairness vs. Explicit Queues

In the reference manual it is stated that queues are maintained in order to ensure that entry calls are honoured in the order of their arrival. All tasks issuing an entry call for a particular entry are queued on a separate queue dedicated to that entry. Then when a task selects to accept an entry call, the task being first on the queue for that entry is accepted first.

It is straightforward to incorporate the explicit queuing mechanism into our semantics. Let $e_1, \ldots e_r$ be all the entries accepted (and called) in the program. We augment our states by r queues, denoted by $q_1, \ldots, q_r$ respectively. Thus, a state will have now the form:

$$s = \langle (T_1\text{-location}) \wedge \ldots \wedge (T_m\text{-location}) ; \eta_1, \ldots, \eta_n ; \chi_1, \ldots, \chi_r \rangle$$

where $\chi_1, \ldots, \chi_r$ are the current values of the queue variables $q_1, \ldots, q_r$. Each χ_i is a (possibly empty) list of tasks.

All the transitions considered above remain the same with the additional requirement that they retain the current values of the queue variables $\chi_1, \ldots, \chi_r$.

In addition we add the following transition:

Queuing Transition

$$\langle \ldots (T_i \ \underline{at} \ e_\ell(\bar{u};\bar{v}) \ldots) \wedge \ldots ; \bar{\eta} ; \chi_1, \ldots, \chi_r \rangle \to$$

$$\langle \ldots (T_i \ \underline{at} \ e_\ell(\bar{u};\bar{v}) \ldots) \wedge \ldots ; \bar{\eta} ; \chi_1, \ldots (\chi_\ell \cdot T_i), \ldots \chi_r \rangle$$

Provided $T_i \notin \chi_\ell$.

This transition corresponds to the step of adding the task T_i to the end of the queue q_ℓ provided it is not already there.

The rendezvous transitions have to be modified so that the first task on the queue will be accepted. We will only present the simplest case where a task T_i is waiting at an entry call on an entry e_ℓ, T_i is at the head of the q_ℓ queue and a task T_j is ready to accept a call for entry e_ℓ.

Rendezvous Transition

$$\langle \ldots (T_i \ \underline{at} \ e_\ell(\bar{u};\bar{v}) ; S_i) \wedge \ldots$$
$$(T_j \ \underline{at} \ \underline{accept} \ e_\ell \ (\bar{f}{:}\underline{in} ; \bar{g}{:}\underline{out}) ; B$$
$$\underline{end} \ e_\ell ; S_j) \wedge \ldots ; \bar{\eta} ; \chi_1, \ldots (T_i \cdot \chi_\ell), \ldots \chi_r \rangle$$
$$\to$$
$$\langle \ldots (T_i \ \underline{at} \ \underline{rendezvous} \ e_\ell ; S_i) \wedge \ldots$$
$$(T_j \ \underline{at} \ \bar{f}:=\bar{u} ; B ; \bar{v}:=\bar{g} ; \underline{end} \ e_\ell ; S_j) \wedge \ldots ;$$
$$\bar{\eta} ; \chi_1, \ldots, \chi_\ell, \ldots, \chi_r \rangle$$

This corresponds to the initiation of a rendezvous between T_j and T_i which is the first task on the queue q_ℓ. The transition also removes T_i from q_ℓ. Similar rules apply to the more general cases that T_i is at a conditional entry call and q_ℓ is

A legal computation is any suffix of an initialized computation.

The notion of legal computation enables us to study the behavior of a concurrent program starting at an arbitrary observation instant, not necessarily the initial one.

We are only interested in maximal computations, that is, computations which can not be extended. Such computations are either infinite or are finite and end in a state s_k which is terminal, i.e., having no possible successor s' such that $s_k \to s'$.

Justice and Fairness

An essential restriction that has to be imposed on execution sequences is a consequence of the fact that we use interleaving in order to model concurrency. In real concurrency every task will eventually finish the execution of one instruction and inevitably start the execution of the next one. It can be held up only by communication instructions. To model the same behavior by interleaving executions we introduce the notion of justice [LPS].

A task T_i is said to move during a transition $s \to s'$, if the location description of T_i in s is different from its location description in s'. Given a state s and an entry e we denote by e'COUNT(s) the number of tasks currently waiting on an entry call for e. A task T_i is said to be enabled in a state s if one of the following conditions is met:

a) T_i is in front of a local statement, i.e. assignment, if or loop statement.
b) T_i is in front of a selective wait statement with an open alternative accepting the entry e while e'COUNT(s) > 0.
c) T_i is in front of an end e statement.
d) T_i is in front of a conditional entry call or a selective wait containing an 'else' clause.

Intuitively, a task is enabled if it is in front of an instruction whose eventual termination depends only on the task itself. In particular, a task waiting in front of an entry call is not considered enabled. This is because for the call to be accepted, a selection of the particular calling task has to be performed by the task potentially accepting this entry call.

A computation σ is defined to be just if it is either finite or every task which is continuously enabled from a certain point on in σ, moves infinitely many times in σ.

This captures the notion of eventual movement in each of the tasks. However, it does not guarantee the requirement of honouring different calls for the same entry in their order of arrival. We therefore stipulate also the requirement of fairness.

An execution sequence σ is defined to be fair if no process T_i may wait forever on an entry-call for the entry e while infinitely many entry calls for e are accepted in σ.

At first appearance this concept seems weaker than the first-in-first-out discipline required in the reference manual.

In the section below we will show that under appropriate restrictions the requirement of fairness is equivalent to the discipline of accepting calling tasks in the order of their arrival.

We therefore define admissible computations to be all legal computations which are both just and fair.

Fairness vs. Explicit Queues

In the reference manual it is stated that queues are maintained in order to ensure that entry calls are honoured in the order of their arrival. All tasks issuing an entry call for a particular entry are queued on a separate queue dedicated to that entry. Then when a task selects to accept an entry call, the task being first on the queue for that entry is accepted first.

It is straightforward to incorporate the explicit queuing mechanism into our semantics. Let $e_1,\ldots e_r$ be all the entries accepted (and called) in the program. We augment our states by r queues, denoted by $q_1,\ldots,q_r$ respectively. Thus, a state will have now the form:

$$s = \langle (T_1\text{-location})\wedge\ldots\wedge(T_m\text{-location});\eta_1,\ldots,\eta_n; \chi_1,\ldots,\chi_r\rangle$$

where $\chi_1,\ldots,\chi_r$ are the current values of the queue variables $q_1,\ldots,q_r$. Each χ_i is a (possibly empty) list of tasks.

All the transitions considered above remain the same with the additional requirement that they retain the current values of the queue variables $\chi_1,\ldots,\chi_r$.

In addition we add the following transition:

Queuing Transition

$$\langle\ldots(T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v})\ldots)\wedge\ldots;\bar{\eta};\chi_1,\ldots,\chi_r\rangle \to$$
$$\langle\ldots(T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v})\ldots)\wedge\ldots;\bar{\eta};\chi_1,\ldots(\chi_\ell\cdot T_i),\ldots\chi_r\rangle$$

Provided $T_i \notin \chi_\ell$.

This transition corresponds to the step of adding the task T_i to the end of the queue q_ℓ provided it is not already there.

The rendezvous transitions have to be modified so that the first task on the queue will be accepted. We will only present the simplest case where a task T_i is waiting at an entry call on an entry e_ℓ, T_i is at the head of the q_ℓ queue and a task T_j is ready to accept a call for entry e_ℓ.

Rendezvous Transition

$$\langle\ldots(T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v});S_i)\wedge\ldots$$
$$(T_j \ \underline{at}\ \underline{accept}\ e_\ell\ (\bar{f}:\underline{in};\bar{g}:\underline{out});B$$
$$\underline{end}\ e_\ell;S_j)\wedge\ldots;\bar{\eta};\chi_1,\ldots(T_i\cdot\chi_\ell),\ldots\chi_r\rangle$$
$$\to$$
$$\langle\ldots(T_i \ \underline{at}\ \underline{rendezvous}\ e_\ell;S_i)\wedge\ldots$$
$$(T_j \ \underline{at}\ \bar{f}:=\bar{u};B;\bar{v}:=\bar{g};\underline{end}\ e_\ell;S_j)\wedge\ldots;$$
$$\bar{\eta};\chi_1,\ldots,\chi_\ell,\ldots,\chi_r\rangle$$

This corresponds to the initiation of a rendezvous between T_j and T_i which is the first task on the queue q_ℓ. The transition also removes T_i from q_ℓ. Similar rules apply to the more general cases that T_i is at a conditional entry call and q_ℓ is

A legal computation is any suffix of an initialized computation.

The notion of legal computation enables us to study the behavior of a concurrent program starting at an arbitrary observation instant, not necessarily the initial one.

We are only interested in maximal computations, that is, computations which can not be extended. Such computations are either infinite or are finite and end in a state s_k which is terminal, i.e., having no possible successor s' such that $s_k \to s'$.

Justice and Fairness

An essential restriction that has to be imposed on execution sequences is a consequence of the fact that we use interleaving in order to model concurrency. In real concurrency every task will eventually finish the execution of one instruction and inevitably start the execution of the next one. It can be held up only by communication instructions. To model the same behavior by interleaving executions we introduce the notion of justice [LPS].

A task T_i is said to move during a transition $s \to s'$, if the location description of T_i in s is different from its location description in s'. Given a state s and an entry e we denote by e'COUNT(s) the number of tasks currently waiting on an entry call for e. A task T_i is said to be enabled in a state s if one of the following conditions is met:

a) T_i is in front of a local statement, i.e. assignment, if or loop statement.

b) T_i is in front of a selective wait statement with an open alternative accepting the entry e while e'COUNT(s) > 0.

c) T_i is in front of an end e statement.

d) T_i is in front of a conditional entry call or a selective wait containing an 'else' clause.

Intuitively, a task is enabled if it is in front of an instruction whose eventual termination depends only on the task itself. In particular, a task waiting in front of an entry call is not considered enabled. This is because for the call to be accepted, a selection of the particular calling task has to be performed by the task potentially accepting this entry call.

A computation σ is defined to be just if it is either finite or every task which is continuously enabled from a certain point on in σ, moves infinitely many times in σ.

This captures the notion of eventual movement in each of the tasks. However, it does not guarantee the requirement of honouring different calls for the same entry in their order of arrival. We therefore stipulate also the requirement of fairness.

An execution sequence σ is defined to be fair if no process T_i may wait forever on an entry-call for the entry e while infinitely many entry calls for e are accepted in σ.

At first appearance this concept seems weaker than the first-in-first-out discipline required in the reference manual.

In the section below we will show that under appropriate restrictions the requirement of fairness is equivalent to the discipline of accepting calling tasks in the order of their arrival.

We therefore define admissible computations to be all legal computations which are both just and fair.

Fairness vs. Explicit Queues

In the reference manual it is stated that queues are maintained in order to ensure that entry calls are honoured in the order of their arrival. All tasks issuing an entry call for a particular entry are queued on a separate queue dedicated to that entry. Then when a task selects to accept an entry call, the task being first on the queue for that entry is accepted first.

It is straightforward to incorporate the explicit queuing mechanism into our semantics. Let $e_1,\dots e_r$ be all the entries accepted (and called) in the program. We augment our states by r queues, denoted by $q_1,\dots,q_r$ respectively. Thus, a state will have now the form:

$$s = \langle (T_1\text{-location}) \wedge \dots \wedge (T_m\text{-location});\ \eta_1,\dots,\eta_n;\ X_1,\dots,X_r \rangle$$

where $X_1,\dots,X_r$ are the current values of the queue variables $q_1,\dots,q_r$. Each X_i is a (possibly empty) list of tasks.

All the transitions considered above remain the same with the additional requirement that they retain the current values of the queue variables $X_1,\dots,X_r$.

In addition we add the following transition:

Queuing Transition

$$\langle \dots (T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v})\dots) \wedge \dots;\ \bar{\eta};\ X_1,\dots,X_r \rangle \to$$

$$\langle \dots (T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v})\dots) \wedge \dots;\ \bar{\eta};\ X_1,\dots(X_\ell \cdot T_i),\dots X_r \rangle$$

Provided $T_i \notin X_\ell$.

This transition corresponds to the step of adding the task T_i to the end of the queue q_ℓ provided it is not already there.

The rendezvous transitions have to be modified so that the first task on the queue will be accepted. We will only present the simplest case where a task T_i is waiting at an entry call on an entry e_ℓ, T_i is at the head of the q_ℓ queue and a task T_j is ready to accept a call for entry e_ℓ.

Rendezvous Transition

$$\langle \dots (T_i \ \underline{at}\ e_\ell(\bar{u};\bar{v})\ ;\ S_i) \wedge \dots$$
$$(T_j \ \underline{at}\ \underline{accept}\ e_\ell\ (\bar{f}:\underline{in}\ ;\ \bar{g}:\underline{out})\ ;\ B$$
$$\underline{end}\ e_\ell\ ;\ S_j) \wedge \dots;\ \bar{\eta};\ X_1,\dots(T_i \cdot X_\ell),\dots X_r \rangle$$
$$\to$$
$$\langle \dots (T_i \ \underline{at}\ \underline{rendezvous}\ e_\ell\ ;\ S_i) \wedge \dots$$
$$(T_j \ \underline{at}\ \bar{f}:=\bar{u}\ ;\ B\ ;\ \bar{v}:=\bar{g}\ ;\ \underline{end}\ e_\ell\ ;\ S_j) \wedge \dots;$$
$$\bar{\eta};\ X_1,\dots,X_\ell,\dots,X_r \rangle$$

This corresponds to the initiation of a rendezvous between T_j and T_i which is the first task on the queue q_ℓ. The transition also removes T_i from q_ℓ. Similar rules apply to the more general cases that T_i is at a conditional entry call and q_ℓ is

currently empty, or when T_j is at a selective wait and selects to accept an entry call for e_ℓ.

We refer to this extended model of computation as the <u>explicit queuing model</u>. In defining admissible computations for this model we only require legality and justice since fairness is implemented by the explicit queuing mechanissm.

Next we will show that under very general conditions our restricted model requiring both justice and fairness is equivalent to the explicit queuing model.

<u>Theorem</u>

Let P be an ACF program which does not refer explicitly to any e'COUNT attribute. Then the class of admissible computations of P is equivalent to the class of admissible computations of P under the explicit queuing model.

<u>Proof</u> (Sketch)

Let σ be a computation under the explicit queuing model. Each state in σ has the form

$$s = \langle \bigwedge_i (T_i\text{-location}) \, ; \overline{\eta} \, ; \overline{\chi} \rangle$$

We construct from σ a computation σ' which is admissible under the fairness requirement by replacing each state such as s above by

$$s' = \langle \bigwedge_i (T_i\text{-location} \, ; \overline{\eta} \rangle$$

This replacement consists simply of omitting the $\overline{\chi}$ component from all states. In addition we have to delete from σ' all transitions corresponding to queuing steps. Since such steps only change the $\overline{\chi}$ component in a state s, they give rise in σ' to trivial transitions of the form

$$s' \rightarrow s' .$$

To see that σ' is a fair computation consider any task T_i waiting in front of an entry call for the entry e_ℓ. This situation is also duplicated in σ. By justice, it will eventually be placed in q_ℓ. If there are infinitely many calls accepted for e_ℓ, each moving T_i one position closer to the top of q_ℓ, eventually T_i will be accepted. This fact is certainly copied into σ' as well. Thus, a task waiting for e_ℓ while infinitely many calls for e_ℓ are accepted will eventually be served.

Let now σ stand for an admissible computation under the fairness requirement. States in σ have the form $s = \bigwedge_i (T_i\text{-location}) \, ; \overline{\eta} \rangle$. We construct a corresponding σ' by first replacing each state such as s above by:

$$s' = \langle \bigwedge_i (T_i\text{-location}) \, ; \overline{\eta} \, ; \Lambda, \ldots, \Lambda \rangle .$$

That is, we uniformly add to each state a list of empty queues.

In addition we make the following two modifications in σ'.

a) We replace each rendezvous transition currently having the form: (for simplicity we omit parameters)

$$\langle \ldots (T_i \ \underline{at} \ e_\ell \ldots) \wedge \ldots \ (T_j \ \underline{at} \ \underline{accept} \ e_\ell \, ; B \, ; \ldots) \wedge \ldots ; \overline{\eta} ;$$
$$X_1, \ldots, \Lambda, \ldots, X_r \rangle \rightarrow$$

$$\langle \ldots (T_i \ \underline{at} \ \underline{rendezvous} \ e_\ell \ldots) \wedge \ldots \ (T_j \ \underline{at} \ B ; \ldots) \wedge \ldots ; \overline{\eta} ;$$
$$X_1, \ldots, \Lambda, \ldots, X_r \rangle$$

by the pair of transitions as follows:

$$\langle \ldots (T_i \ \underline{at} \ e_\ell) \wedge \ldots \ (T_j \ \underline{at} \ \underline{accept} \ e_\ell \, ; B ; \ldots) \wedge \ldots ; \ \overline{\eta} ;$$
$$X_1, \ldots, \Lambda, \ldots, X_r \rangle \rightarrow \text{(queuing step)}$$

$$\langle \ldots (T_i \ \underline{at} \ e_\ell) \wedge \ldots \ (T_j \ \underline{at} \ \underline{accept} \ e_\ell \, ; B ; \ldots) \wedge \ldots ; \overline{\eta} ;$$
$$X_1, \ldots (T_i), \ldots, X_r \rangle \rightarrow \text{(rendezvous transition)}$$

$$\langle \ldots (T_i \ \underline{at} \ \underline{rendezvous} \ e_\ell \ldots) \wedge \ldots \ (T_j \ \underline{at} \ B ; \ldots) \wedge \ldots ; \overline{\eta} ;$$
$$X_1, \ldots, \Lambda, \ldots, X_r \rangle$$

This pair of transitions places T_i on the queue, which is assumed to be empty, just one step before T_j accepts. It certainly satisfies the requirement that under the explicit queuing model only tasks which are at the head of the q_ℓ queue are accepted. It also defers the act of queuing to the last moment possible.

b) If σ' contains a task T_i which is waiting in front of an e_ℓ call at a state s such that no calls on e_ℓ are accepted beyond s, then obviously T_i is stuck at that position forever. We insert anywhere following the state s the queuing transition.

$$\langle \ldots (T_i \ \underline{at} \ e_\ell \ldots) \wedge \ldots \ ; \overline{\eta} \, ; X_1, \ldots X_\ell, \ldots X_r \rangle \rightarrow$$

$$\langle \ldots (T_i \ \underline{at} \ e_\ell \ldots) \wedge \ldots \ ; \overline{\eta} \, ; X_1, \ldots (X_\ell \cdot T_i), \ldots X_r \rangle$$

All components X_ℓ following this transition should be modified accordingly. Thus, with stuck tasks, we defer their being queued to the point beyond which there are no more calls accepted for the entry e_ℓ.

This transformation will construct an admissible explicit queuing computation out of every fair admissible computation.

Supported by this theorem we will proceed to study ACF without explicit queuing mechanisms. We use instead the concept of admissible computations, being fair and just legal computations.

However, as shown above, our treatment is easily extendable to accomodate explicit queuing as well.

<u>Proof Theory</u>

We use temporal logic in order to describe properties of admissible computations of an ACF program P. In describing state properties we use predicates over the program variables $y_1, \ldots, y_n$ and the task location descriptors. State properties are then combined into temporal formulas using the temporal operators: $\Box$ (always), $\Diamond$ (sometimes), $\bigcirc$ (next) and $\mathcal{U}$ (until). We refer the interested reader to [MP1] for an introduction to temporal logic and its usage for proving properties of programs. The proof system that we would outline here is based on the basic approach presented in [MP3].

Let τ be any of the transitions presented above in the semantic definition of computations. We observe that in a given program P there are only finitely many transitions corresponding to each of the statements in any of the tasks. Joint transitions such as rendezvous correspond to a pair of matching statements in two different tasks, but there are only finitely many of them.

We say that a transition τ leads from φ to ψ, where φ and ψ are state properties, if for every pair of states s and s' such that $s \xrightarrow{\tau} s'$ it follows that $\varphi(s) \supset \psi(s')$ holds. This implies that if φ was true before the transition then ψ will hold after the transition. For every type of transition τ it is possible to write a formula involving the program and location variables and the predicates φ and ψ which will be valid iff τ leads from φ to ψ.

For example consider the case that τ is a transition of T_i from the location $\bar{y}:=f(\bar{y});S_i$ to the location S_i. Let $\varphi=\varphi(\pi_1,..\pi_m;y_1,..y_n)$, $\psi=\psi(\pi_1,..\pi_m;y_1,..y_n)$ where π_i, $i=1,..m$ are the location variables describing the current location of the tasks T_i, $i=1,..m$ respectively. Then τ leads from φ to ψ iff the following implication is valid:

$$\varphi(\pi_1,..[\bar{y}:=f(\bar{y});S_i],..\pi_m;\bar{y}) \supset$$
$$\psi(\pi_1,..[S_i],..\pi_m;f(\bar{y}))$$

Similarly, for the case that τ is a conditional statement we have that τ leads from φ to ψ iff:

$$\varphi(\pi_1,..[\underline{\text{if}}\ p(\bar{y})\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2;S],..\pi_m;\bar{y}) \supset$$
$$\underline{\text{if}}\ p(\bar{y})\ \underline{\text{then}}\ \psi(\pi_1,..[S_1;S],..\pi_m;\bar{y})$$
$$\underline{\text{else}}\ \psi(\pi_1,..[S_2;S],..\pi_m;\bar{y})$$

A transition τ is said to be related to task T_i if it is either a local statement in the task T_i, or a joint transition which involves T_i as one of its active participants.

We say that a task T_i leads from φ to ψ if all transitions τ related to T_i lead from φ to ψ. The complete program P is said to lead from φ to ψ if each of its tasks $T_1,..T_m$ leads from φ to ψ.

We are ready now to formulate several proof principles which are used to derive temporal properties of ACF programs. We present here only some derived principles adequate for most of the needed applications. We refer the reader again to [MP3] for the more basic axioms. The principles presented here are adequate for proving invariance and liveness properties.

The Invariance Rule: (IINV)

Let φ be a state property.

$\vdash \varphi(P_1,\ldots,P_m;\bar{y})$

$\vdash$ P leads from φ to φ

$\vdash \Box\varphi$

This rule states that if φ is such that it holds initially for the initial state where each task T_i is at the beginning of its program P_i. Also it is assumed that every transition in P preserves φ. Then we may conclude that φ is invariantly true for all admissible computations.

The following two rules are useful for establishing liveness properties.

Let φ, ψ be two state properties and T_k one of the tasks.

The Justice Rule: (JUST)

1. $\vdash$ P leads from φ to $\varphi \vee \psi$
2. $\vdash T_k$ leads from φ to ψ
3. $\vdash \varphi \supset (\psi \vee \text{Enabled}(T_k))$

$\vdash \varphi \supset \varphi\,\mathcal{U}\,\psi$

This rule states that if every transition in P leads from φ to $\varphi\vee\psi$, every transition in T_k leads from φ to ψ and φ implies that either ψ is already true or that T_k is enabled, then ψ is guaranteed to eventually happen and φ will continuously hold until then. Assume that we have an admissible computation whose first state satisfies φ. By the first premise φ will hold continuously until ψ is realized, if ever. By the third premise the continuous holding of φ implies that ψ will happen or that T_k is continuously enabled. By justice T_k must be eventually moved which by the second premise must produce ψ immediately.

The following liveness rule is more specific and relies on the fairness assumption applied to tasks waiting on entry calls.

The Fairness Rule: (FAIR)

1. $\vdash$ P leads from φ to $\varphi\vee\psi$
2. $\vdash T_k$ leads from φ to ψ
3. $\vdash \varphi \supset T_k\ \underline{\text{at}}\ e(\bar{u};\bar{v});S$
4. $\vdash \varphi \supset \Diamond(\psi \vee \underline{\text{after}}\ \underline{\text{accept}}\ e)$

$\vdash \varphi \supset \varphi\,\mathcal{U}\,\psi$

The difference between this and the previous rule lies in premises 3. and 4. Premise 3. assures that while φ holds T_k is waiting in front of an entry call on the entry e. Premise 4. states that φ implies that eventually either ψ will be realized or a call for entry e will be accepted. Thus if T_k is stuck and φ maintained forever, an infinite number of e-calls would be accepted. By fairness T_k must eventually be accepted, leading to ψ.

An Example

As illustration of a proof of a liveness property we consider the program in Figure 1.

We wish to prove termination of the whole program. That is:

$$\vdash \bigwedge_{i=0}^{3} (T_i\ \underline{\text{at}}\ P_i) \supset \Diamond(\bigwedge_{i=0}^{3} T_i\ \underline{\text{at}}\ \Lambda)$$

A crucial stage in the termination of the program is given by:

Lemma A $\quad \vdash (T_1\ \underline{\text{at}}\ P_1) \supset \Diamond\ (n=0)$

To prove this we will attempt to prove

1. $\vdash T_1\ \underline{\text{at}}\ e_1(1,a) \supset \Diamond\,(n=0 \vee [(T_0\ \underline{\text{after}}\ \underline{\text{accept}}\ e_1) \wedge (T_1\ \underline{\text{at}}\ \underline{\text{rendezvous}}\ e_1)])$

Note the abbreviation of T_0 after accept e_1 standing for the more detailed description.

This will be a conclusion of the fairness rule by taking

φ_1: T_1 <u>at</u> $e_1(1,a)$ and

ψ_1: $n=0 \vee [(T_0$ <u>after</u> <u>accept</u> $e_1) \wedge$

$(T_1$ <u>at</u> <u>rendezvous</u> $e_1)]$

It only remains to establish the three premises to the FAIR rule. The first premise is:

$$\vdash P \text{ leads from } \varphi_1 \text{ to } \varphi_1 \vee \psi_1 .$$

Obviously any transition in P which does not involve T_1 leaves T_1 at $e_1(1,a)$.

$$\vdash T_1 \text{ leads from } \varphi_1 \text{ to } \psi_1$$

The only possible transition involving T_1 is the acceptance of the e_1 call of T_1 by T_0 which leads immediately to ψ_1.

$\vdash \varphi_1 \supset T_1$ <u>at</u> $e_1(1,a)$ - Obvious.

The only premise requiring further proving is the last one, namely:

$\vdash \varphi_1 \supset \Diamond(\psi_1 \vee T_0$ <u>after</u> <u>accept</u> $e_1)$

<u>Lemma B</u> $\vdash \varphi_1 \supset \Diamond(\psi_1 \vee [\varphi_1 \wedge T_0$ <u>at</u> <u>select</u>$])$

That is, given that T_1 is waiting at the e_1 entry call, then either n will be zero, the T_1 entry call will be accepted or T_0 will reach once more the location immediately in front of the select statement. This is proved by considering all the possible locations in which T_0 might currently be and using justice following its execution to the beginning of the loop.

<u>Lemma C</u> $\vdash [T_0$ <u>at</u> <u>select</u> $\wedge\ n=u] \supset \Diamond(n=0 \vee$

T_0 <u>after</u> <u>accept</u> $e_1 \vee [T_0$ <u>at</u> <u>select</u> $\wedge\ n<u])$

This states that T_0 being at the beginning of the select statement with a certain value of n, then either n will be set to zero, an e_1 entry accepted or T_0 will return to the beginning of the select with a strictly lower value of n. By considering the different entries that T_0 may choose to select, it is obvious that either e_1 is accepted or e_2 is accepted which inevitably decrements the value of n. By applying induction on the value of u to Lemma C we obtain

$\vdash [T_0$ <u>at</u> <u>select</u> $\wedge\ n=u] \supset$

$\Diamond(n=0 \vee T_0$ <u>after</u> <u>accept</u> $e_1)$

This certainly establishes the last premise for the FAIR rule and proves Lemma A.

To proceed from $n=0$ to total termination is straightforward.

<u>Conclusions and Discussions</u>

In this short paper we have outlined a proof theoretical approach to the semantic definition and verification of a fragment of the ADA language. We have concentrated in particular on the synchronization and tasking mechanism for which the literature contains much less established formal techniques than for sequential programs. We have also shown that for programs obeying some restrictions, both the semantic definition and proof principles become simpler. This has been demonstrated for programs which do not explicitly test the size of entry queues (e'COUNT). For such programs the whole concept of explicit queues which does have an implementation flavor and may appear as a strange intruder in the formal definition of a language, can be replaced by the much more liberal notion of fairness. This may hint that programs obeying these restrictions are somewhat more well-constructed in much the same way that structured programs, leading to a simpler proof theory, are considered better constructed.

In order that this preliminary investigation will not remain an academic exercise, one should seriously consider the extension of this approach to cover all of the ADA language. In trying to do so there are two types of extensions one has to make. The first type should consider many additional details that we have omitted for the sake of simplicity. Providing rules both on the operational semantic level and on the temporal level, for treating these additional features of the language may require ingenuity but is still a standard extension of the approach suggested here. This includes the sequential features of the language such as blocks, declarations, procedures, packages and data structures. The second type of feature is much more challenging since it seems to question the adequacy of temporal logic for its expression. These are all the features that relate to real time and its measurement such as the delay statements of different forms. Statements claiming that a certain block of code will be terminated within a certain number of time units since its initiation seem to be out of the scope of temporal logic which by nature is qualitative rather than quantitative. One development that this seems to call for is the extension of temporal logic into some more quantitative time logic in which such statements can be expressed.

Within the framework presented here, we would like to point to another approach which may yet be able to manage these features without having to extend the time logic. This approach is to add to the state some additional artifacts which will enable to capture quantitative time in increasing degrees of accuracy.

For example, in the simplest approximation we could describe the location of a delayed task by a state such as:

$<..(T_{i_1}$ <u>at</u> <u>delay</u> $(n_1))\wedge..(T_{i_2}$ <u>at</u> <u>delay</u> $(n_2))\wedge..>$

Then we would introduce a special time-step transition which will transform a state such as the above into

$<..(T_{i_1}$ <u>at</u> <u>delay</u> $(n_1))\wedge..(T_{i_2}$ <u>at</u> <u>delay</u> $(n_2))\wedge..>$

$\xrightarrow{\text{time-step}}$

$<..(T_{i_1}$ <u>at</u> <u>delay</u> $(n_1-1))\wedge..(T_{i_2}$ <u>at</u> <u>delay</u> $(n_2-1))\wedge..>$

and explicitly require that this transition be applied with justice and all components of the form

"T_i at delay (0)" are resolved before the next time step is taken.

Such a device will ensure a correct synchronization among all the delay statements, which for many applications is quite sufficient. On the other hand it does not assure a correct compatability between explicit delay statements and the timing of execution of other instructions such as assignment, communication, etc. The report itself does not say anything about this since it is evidently implementation dependent.

For a hint how even these requirements can to some degree be incorporated into our model, one could introduce a master clock into the state. This will be a global variable which is incremented on each time step transition. Intuitively this clock should count in "big" units, much bigger than the timing of a single instruction. In addition we could introduce instruction counters $c_1,..c_m$, one for each task. These will count the number of operations, measured in some basic units, performed by the task T_i since the last time-step transition. They are reset to zero on each time-step transition, and incremented whenever task T_i performs a transition. We may now add to our semantics the restriction that none of these counters ever exceeds 1000, say. This implies that no task performs more than a 1000 elementary operations in a "big" time slot. On the other hand we may also require that no time step transition is allowed when there exists a c_i such that $c_i < 500$. This could provide a lower bound on the rate of speeds of the different tasks.

By adding such a timing mechanism into the operational semantics itself - states and transitions, we are now assured that the temporal logic approach is still applicable and can even deal with real time analysis.

References

[A] Reference Manual for the ADA Programming Language. United States Department of Defense, July 1981.

[AFR] Apt, K.R., Francez, N., de Roever, W.P. - A Proof System for Communicating Sequential Processes, ACM Transactions on Programming Languages and Systems 3 (July 1980) pp. 359-385.

[BH] Brinch Hansen, P. - Distributed Processes: A Concurrent Programming Concept - CACM 21, 11 (November 1978) pp. 934-941.

[G] Gerth, R. - A Sound and Complete Hoare Axiomization of the ADA Rendezvous. Proc. ICALP July 1982, Aarhus, Denmark.

[H] Hoare, C.A.R. - Communicating Sequential Processes. CACM, Vol. 21, No. 8 (1978) pp. 666-677.

[L] Lamport, L. - "Sometime" is sometimes "not never". On Temporal logic of programs. 7th Symposium on Principles of Programming Languages, Jan. 1980, pp. 174-186.

[LPS] Lehman, D., Pnueli, A., Stavi, J. - Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. Proc. of the 8th Symposium on Automata Languages and Programming, Lecture Notes in Computer Science 115, Springer Verlag (July 1981), pp. 264-277.

[M] Milner, R. - A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer Verlag (1980).

[MP1] Manna, Z., Pnueli, A. - Verification of Concurrent Programs: The Temporal Framework - in the Correctness. Problem in Computer Science (R.S. Boyer, J.S. Moore eds.) International Lecture Series in Computer Science, Academic Press, London 1981.

[MP2] Manna, Z., Pnueli, A. - Verification of Concurrent Programs: Temporal Proof Principles. Proc. of the Workshop on Logics of Programs, Yorktown Heights, Springer Verlag, Notes in Computer Science (D. Kozen ed.) 1981.

[MP3] Manna, Z. and Pnueli, A. - Verification of Concurrent Programs: The Temporal Proof System. In the Proceeding of the 4th Advanced Course on Programming, Amsterdam, June 1982.

[RDKR] Roncken, M., van Diepen, N., Kramer, M., de Roever, W.P. - A Proof System for Brinch Hansen's Distributed Processes. Technical Report CS-81-5 Rijksuniversiteit Utrecht.

```
task T0 is
     entry e1 (id1 : in INTEGER ; ret1 : out BOOLEAN) ;
     entry e2 (id2 : in INTEGER ; ret2 : out BOOLEAN)
end T0

task body T0 is
   n : INTEGER : = 1  ;
begin
     loop
          select
                 accept e1 (id1 ; ret1)  ;
                        if id1 = 1 then n: = 0
                        elsif n>0 then n : = n+1
                        ret1 : = (n>0)
                 end e1
          or
                 accept e2 (id2 ; ret2)  ;
                        if n>0 then n : = n-1  ;
                        ret2 : = (n>0)
                 end e2
          or     terminate
          end    select
     end loop
end T0

task body T1 is
   a : BOOLEAN ;
begin e1 (1,a) end T1  ;

task body T2 is
   b : BOOLEAN : = true  ;
begin while b do
           loop e1 (2,b) end loop
end T2

task body T3 is
   c : BOOLEAN : = true
begin while c do
           loop e2 (3,c) end loop
end T3
```

FIGURE 1