RENDZVOUS 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
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
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 mechanisms, 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

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.

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 [MPL]. We proceed then to define temporal proof principles which are analogous to the ones developed in [MPL] 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
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 T₁,...,Tₘ. Let all the variables declared in all of the tasks be \( \vec{y} = (y₁,...,yₙ) \) with \( y₁ \) ranging over \( D₁ \cup \{ \text{undefined} \} \). A state in the execution of P has the form:

\[
s = \langle (T₁\text{-location})\wedge (T₂\text{-location})\wedge \ldots \wedge (Tₘ\text{-location}) ; n₁,...,nₙ \rangle
\]

where \( n₁ \in D₁ \cup \{ \text{undefined} \} \) is the current value of the variable \( y₁ \) in state \( s \). Each \( T_i\text{-location} \), \( i = 1,...,m \) is a description of the location of the task \( T_i \) in its program (task body). It has the general form: \( T_i \) \text{ 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 \( A \) if \( T_i \) has terminated.

We define a succession relation, written \( s \rightarrow s' \), 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:
\[
\langle \ldots (T_i \text{ at } \vec{y} = f(\vec{y}); S \wedge \ldots; \vec{n}) \rightarrow \ldots (T_i \text{ at } S \wedge \ldots; f(\vec{n})) \rangle
\]

For convenience we use simultaneous assignments to all of \( y₁,...,yₙ \).

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 the assignment statement and the value of \( f(\vec{n}) \) is assigned to the variables \( \vec{y} \).

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

If Transition
\[
\langle \ldots (T_i \text{ if } p(\vec{y}) \text{ then } S_1 \text{ else } S_2 \text{ and } S \wedge \ldots; \vec{n}) \rightarrow \ldots (T_i \text{ at } S_1 \wedge S \wedge \ldots; \vec{n}) \rangle
\]

Provided \( p(\vec{n}) = \text{true} \).

Similarly the 'else' clause may be taken:
\[
\langle \ldots (T_i \text{ if } p(\vec{y}) \text{ then } S_1 \text{ else } S_2 \text{ and } S \wedge \ldots; \vec{n}) \rightarrow \ldots (T_i \text{ at } S_2 \wedge S \wedge \ldots; \vec{n}) \rangle
\]

Provided \( p(\vec{n}) = \text{false} \).

Loop Transition
\[
\langle \ldots (T_i \text{ while } c(\vec{y}) \text{ do } B \wedge S \wedge \ldots; \vec{n}) \rightarrow \ldots (T_i \text{ at } B; \text{ while } c(\vec{y}) \text{ do } B \wedge S \wedge \ldots; \vec{n}) \rangle
\]

Provided \( c(\vec{n}) = \text{true} \).

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

\[
\langle \ldots (T_i \text{ while } c(\vec{y}) \text{ do } B \wedge S \wedge \ldots; \vec{n}) \rightarrow \ldots (T_i \text{ at } S \wedge \ldots; \vec{n}) \rangle
\]

Provided \( c(\vec{n}) = \text{false} \).

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:
Provided no task is currently in front of an accept statement for \( \alpha \) or a selective wait with an open alternative of accepting \( \alpha \).

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

```
<... (T_j at select \( e(\bar{u}, \bar{v}) \); S_j) \ldots ; \bar{\pi} >
```

```
<... (T_1 at \( S'_1 \); \( S''_1 \)) \ldots ; \bar{\pi} >
```

A special transition allows termination of the complete program.

```
Termination Transition:
< \( \bigwedge _i (T_i \at \alpha) \) \land \( \bigwedge _j (T_j \at select... or terminate...) \) \land \bar{\pi} >
```

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:

\( s_0 = (T_1 \at P_1 \ldots (T_m \at P_m) \underdefined > ) \)

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 \( \sigma \), are related by the succession relation defined above

\( s_i \rightarrow s_{i+1} \)
A legal computation is any suffix of an initial computation.

The notion of a legal computation enables us to allow the behavior of a concurrent program starting at an arbitrary observation instant, not necessarily terminal. We are only interested in maximal computations, that is, computations which cannot be extended. Such computations are either infinite or are finite and end in a state  s which is terminal, i.e., having no possible successor  s' such that  s→s'.

A. Focus 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 is eventually finished the execution of one instruction and immediately the execution of the next one. It is not helpful only by message instructions. To model the same behavior by interleaving execution we introduce the notion of Justice (LSA).

Task  T is said to move during a transition  s→s' if the location description of  T in  s is different from its location description in  s'.

Given a state  s and an entry  e we denote by  COUNT(e) the number of tasks currently waiting at  e as an entry call for  e. A task  T is said to be enabled in a state  s if one of the following conditions is met:

1.  T is in front of a local statement, i.e., assignment, if or loop statement.
2.  T is in front of a selective wait statement with an open alternative accepting the entry  e while  COUNT(e) > 0.
3.  T is in front of an end statement.
4.  T is in front of a conditional entry call of a selective wait containing an 'else' clause.

Intuitively, a task is enabled if it is in front of an instruction whose eventual termination depends 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 half of the tasks. However, it does not guarantee the receipt of honouring different calls for the same entry in their order of arrival. We therefore stipulate also the requirement of fairness.

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

At first sight 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  q, ... , q be all the entries accepted (and called) in the program. We augment our states by r queues, denoted by  q, ... , q respectively. Thus, a state will now have the form:

s = (σ₁→σ₁→σ₁→σ₁)...

where  x₁,...,x are the current values of the queue variables  q,...,q. Each  X 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,...,X.

In addition we add the following transition:

Quering Transition

<s₁₁ σ₁ → σ₁ → σ₁→ σ₁>...

<s₁₁ σ₁ → σ₁ → σ₁→ σ₁>...

Provided  T₁ ∈ X₁

This transition corresponds to the step of adding the task  T₁ to the end of the queue  q₁ 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  T₁ is waiting at an entry call on an entry  e₁,  T₁ is at the head of the  q₁ queue and a task  T₂ is ready to accept a call for entry  e₁.

Rendezvous Transition

<s₁₁ σ₁ → σ₁ → σ₁→ σ₁>...

<s₁₁ σ₁ → σ₁ → σ₁→ σ₁>...

This corresponds to the initiation of a rendezvous between  T₁ and  T₂ which is the first task on the queue  q₁. The transition also removes  T₁ from  q₁. Similar rules apply to the more general case that  T₁ is at a conditional entry call and  q₁ is...
A legal computation is any suffix of an initial-
ized 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 $s_k \rightarrow s'$.

**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 concurre-
cy. In real concurrency every task will eventually
finish the execution of one instruction and
invariably start the execution of the next one. It
can be held up only by communication instructions.

To model the same behavior by interleaving execu-
tions we introduce the notion of justice [LPS].

A task $T_i$ is said to move during a transition
$s \rightarrow s'$, 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
$e \text{COUNT}(s)$ 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 $e \text{COUNT}(s) > 0$.

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 accep-
ted, a selection of the particular calling task has
to be performed by the task potentially accepting
this entry call.

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

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 $\sigma$ 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 $\sigma$.

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 ap-
propriate 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 $q_1, \ldots, q_r$
be all the entries accepted (and called) in the
program. We augment our states by $r$ queues, de-
noted by $q_1, \ldots, q_r$ respectively. Thus, a state
will have now the form:

$s = \langle T_1, location \rangle \ldots \langle T_n, location \rangle ; n_1, \ldots, n_r ; X_1, \ldots, X_r >$

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

All the transitions considered above remain the
same with the additional requirement that they re-
tain the current values of the queue variables
$X_1, \ldots, X_r$.

In addition we add the following transition:

**Queueing Transition**

\[
\begin{align*}
&\langle T_1, at \ e_1 \langle u; v \rangle ; \ldots \rangle ; n_1, \ldots, n_r ; X_1, \ldots, X_r > \rightarrow \\
&\langle T_1, at \ e_1 \langle u; v \rangle ; \ldots \rangle ; n_1, \ldots, n_r ; X_1, \ldots, X_r, e_1 \rangle \ldots \rangle \\
&\text{Provided } T_i \in X_i.
\end{align*}
\]

This transition corresponds to the step of ad-
 ding the task $T_i$ to the end of the queue $q_i$
provided it is not already there.

The rendezvous transitions have to be modified
so that the first task on the queue will be accep-
ted. We will only present the simplest case where
a task $T_i$ is waiting at an entry call on an entry
$e_i$, $T_i$ is at the head of the $q_i$ queue and a

**Rendezvous Transition**

\[
\begin{align*}
&\langle T_1, at \ e_1 \langle u; v \rangle ; S_j \rangle \ldots \rangle \\
&\langle T_j, at \ accept \ e_1 \langle \bar{f}; \bar{g}; \bar{i}; \bar{c} \rangle \rangle \rangle ; B \end{align*}
\]

\[
\begin{align*}
&\text{end } e_1 ; S_j \rangle \ldots \rangle ; n_1, \ldots, n_r ; X_1, \ldots, X_r, e_1 \rangle \ldots \rangle \\
&\text{Provided } T_i \in X_i.
\end{align*}
\]

This corresponds to the injection of a rendezvous
between $T_i$ and $T_i$, which is the first task on
the queue $q_i$. The transition also removes $T_i$ from
$q_i$. Similar rules apply to the more general case
that $T_i$ is at a conditional entry call and $q_i$ 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 cannot 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 $s_k \rightarrow s'$.

**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 [EDS].

A task $T_i$ is said to move during a transition $s \rightarrow s'$, 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 $\delta(COUNT(s))$ 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 $\delta(COUNT(s)) > 0$.
- (c) $T_i$ is in front of an end $e$ statement.
- (d) $T_i$ is in front of a conditional entry call of 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 $\sigma$ is defined to be just if it is either finite or every task which is continuously enabled from a certain point on in $\sigma$, moves infinitely many times in $\sigma$.

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 $\sigma$ 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 $\sigma$.

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 $q_1, \ldots, q_r$ be all the queues 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:

$$ s = \langle (T_1 - location) \ldots (T_n - location); \overline{\eta}_1, \ldots, \overline{\eta}_n; x_1; \ldots; x_r > $$

where $x_1, \ldots, x_r$ are the current values of the queue variables $q_1, \ldots, q_r$. 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 $x_1, \ldots, x_r$.

In addition we add the following transition:

**Queuing Transition**

$$... (T_i \text{ at } e_1 (u; v) \ldots \overline{\eta} ; x_1; \ldots; x_r) \Rightarrow ... (T_i \text{ at } e_1 (u; v) \ldots \overline{\eta} ; x_1' ; \ldots ; (x_i' ; T_i) ; \ldots ; x_r) \Rightarrow $$

Provided $T_i \in X_i$.

This transition corresponds to the step of adding the task $T_i$ to the end of the queue $q_i$ 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 $e_i$. We will denote $T_i$ as the head of the queue $q_i$ and a task $T_j$ as ready to accept a call for entry $e_i$.

**Rendezvous Transition**

$$... (T_i \text{ at } e_1 (u; v) ; S_j) \Rightarrow $$

$$ (T_j \text{ at } \overline{\eta} ; F \in ; U \text{ out} ; B ; e_j ; S_j) \Rightarrow $$

$$... (T_i \text{ at rendezvous } e_i ; S_j) \Rightarrow $$

$$... (T_j \text{ at } F; u; B ; V ; \overline{\eta} ; e_j ; S_j) \Rightarrow $$

$$... (T_i \text{ at } \overline{\eta} ; x_1, x_1', \ldots, x_r) \Rightarrow $$

This corresponds to the initiation of a rendezvous between $T_i$ and $T_j$ which is the first task on the queue $q_i$. The transition also removes $T_i$ from $q_i$. Similar rules apply to the more general case that $T_i$ is at a conditional entry call and $q_i$ is
currently empty, or when \( T_i \) is at a selective wait and selects to accept an entry call for \( e_k \).

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

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

**Theorem**

Let \( P \) be an ACF program which does not refer explicitly to any \( \varepsilon \) 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.

**Proof (Sketch)**

Let \( \sigma \) be a computation under the explicit queuing model. Each state in \( \sigma \) has the form

\[
s = \langle (T_i \text{-location}) ; \bar{\pi} ; X \rangle
\]

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

\[
s' = \langle (T_i \text{-location}) ; \bar{\pi} \rangle
\]

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

\[
s' + s' = \langle (T_i \text{-location}) ; \bar{\pi} ; \bar{\alpha} \rangle
\]

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

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

\[
s' = \langle (T_i \text{-location}) ; \bar{\pi} ; \bar{\alpha} \rangle
\]

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

In addition we make the following two modifications in \( \sigma' \):

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

\[
<(T_i \text{ at rendezvous } e_k) \ldots (T_j \text{ at } B_i \ldots) ; \bar{\pi} \rangle ; X_1 \ldots \bar{\alpha} \ldots X_k >
\]

by the pair of transitions as follows:

\[
<(T_i \text{ at } e_k) \ldots (T_j \text{ at } E_k ; B_i \ldots) ; \bar{\pi} \rangle ; X_1 \ldots \bar{\alpha} \ldots X_k > \quad \text{(queuing step)}
\]

\[
<(T_i \text{ at } e_k) \ldots (T_j \text{ at } accept e_k ; B_i \ldots) ; \bar{\pi} \rangle ; X_1 \ldots \bar{\alpha} \ldots X_k > \quad \text{(rendezvous transition)}
\]

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

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

\[
<(T_i \text{ at } e_k) \ldots \bar{\pi} ; X_1 \ldots \bar{\alpha} \ldots X_k >
\]

All components \( X_k \) 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 \( e_k \).

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 accommodate explicit queuing as well.

**Proof Theory**

We use temporal logic in order to describe properties of admissible computations of an ACF program. 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: \( \square \) (always), \( \langle\langle \rangle\rangle \) (sometimes), \( \vee \) (next) and \( U \) (until). We refer the interested reader to [MPL] 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 [MPL].

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

For example consider the case that $\tau$ is a transition of $T_k$ from the location $V_l = f(y); S_l$ to the location $S_l$. Let $\psi = \psi(s_1, \ldots, s_m; y_1, \ldots, y_n)$ where $s_i, 1 \leq i \leq m$ are the location variables describing the current location of the tasks $T_k, i = 1, \ldots, m$ respectively. Then $\tau$ leads from $\psi$ to $\psi'$ iff the following implication is valid:

$$\psi(s_1, \ldots, s_m; y_1, \ldots, y_n) \Rightarrow \psi'(s_1, \ldots, f(y))$$

Similarly, for the case that $\tau$ is a conditional statement we have that $\tau$ leads from $\psi$ to $\psi'$ iff:

$$\psi(s_1, \ldots, s_m; y_1, \ldots, y_n) \Rightarrow$$

$$\begin{cases} \text{if } p(y) \text{ then } \psi(s_1, s_2, \ldots, s_m, y) \\ \text{else } \psi(s_1, [s_1, s_2, \ldots, s_m, y] \text{ if } p(y) \text{ then } \psi(s_1, [s_1, s_2, \ldots, s_m, y] \text{ else } \psi(s_1, s_2, \ldots, s_m, y) \end{cases}$$

A transition $\tau$ is said to be related to task $T_k$ if it is either a local statement in the task $T_k$, or a joint transition which involves $T_k$ and one of its active participants.

We say that a task $T_k$ leads from $\psi$ to $\psi'$ if all transitions $\tau$ related to $T_k$ lead from $\psi$ to $\psi'$. The complete program $P$ is said to lead from $\psi$ to $\psi'$ if each of its tasks $T_1, \ldots, T_m$ leads from $\psi$ to $\psi'$.

We are ready now to formulate several proof principles which are used to derive temporal properties of ACP 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: (INV)

Let $\psi$ be a state property.

$$\vdash \psi(P_1, \ldots, P_m; y)$$

$$\vdash \psi$$

This rule states that if $\psi$ is such that it holds initially for the initial state where each task $T_1$ is at the beginning of its program $P_1$. Also it is assumed that every transition in $P$ preserves $\psi$. Then we may conclude that $\psi$ is invariantly true for all admissible computations. The following two rules are useful for establishing liveness properties.

Let $\psi, \psi'$ be two state properties and $T_k$ one of the tasks.

The fairness rule: (FAIR)

1. $\vdash P$ leads from $\psi$ to $\psi'$
2. $\vdash T_k$ leads from $\psi$ to $\psi$
3. $\psi \Rightarrow (\psi \lor \text{Enabled}(T_k))$

$$\vdash \psi \Rightarrow \psi'$$

This rule states that if every transition in $P$ leads from $\psi$ to $\psi'$, every transition in $T_k$ leads from $\psi$ to $\psi$ and $\psi$ implies that either $\psi$ is already true or that $T_k$ is enabled, then $\psi$ is guaranteed to eventually happen if $\psi$ will continuously hold until then. Assume that we have an admissible computation whose first satisfies $\psi$. By the first premise $\psi$ will hold continuously until $\psi$ is realized, if ever. By the third premise the continuous holding of $\psi$ implies that $\psi$ will happen or that $T_k$ is continuously enabled. By fairness $T_k$ must be eventually moved which by the second premise must produce $\psi$ 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 $\psi$ to $\psi'$
2. $\vdash T_k$ leads from $\psi$ to $\psi$
3. $\psi \Rightarrow T_k \text{ at } e_1(y); S$
4. $\psi \Rightarrow \psi \land \psi'$

$$\vdash \psi \Rightarrow \psi'$$

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

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:

$$\vdash \bigwedge_{i=0}^{3} (T_i \text{ at } P_i) \Rightarrow \bigwedge_{i=0}^{3} (T_i \text{ at } 'A')$$

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

Lemma A

$$\vdash (T_1 \text{ at } P_1) \Rightarrow (n = 0)$$

To prove this we will attempt to prove:

1. $\vdash T_1 \text{ at } e_1(1, a) \Rightarrow (n = 0 \lor (T_0 \text{ after accept } c_1) \land (T_1 \text{ at } \text{ rendezvous } e_2))$

Note the abbreviation of $T_0 \text{ after accept } c_1$ standing for the more detailed description. This will be a conclusion of the fairness rule by taking
\[ \varphi_1: T_1 \text{ at } e_1(1,a) \quad \text{and} \]
\[ \psi_1: n = 0 \lor (T_0 \text{ after accept } e_1) \land \\
(T_1 \text{ at } \text{rendezvous } e_1). \]

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

- \( T_1 \text{ leads from } \varphi_1 \text{ to } \psi_1 \).

Obviously any transition in \( P \) which does not involve \( T_1 \text{ at } e_1(1,a) \).

- \( T_1 \text{ leads from } \psi_1 \text{ to } \psi_1 \).

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 \( \psi_1 \).

- \( \psi_1 \Rightarrow T_1 \text{ at } e_1(1,a) \) - Obvious.

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

- \( \psi_1 \Rightarrow \lnot ( \psi_1 \lor T_0 \text{ after accept } e_1) \).

**Lemma B** \( \vdash \psi_1 \Rightarrow \lnot ( \psi_1 \lor (T_0 \text{ at select})) \).

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.

**Lemma C** \( \vdash (T_0 \text{ at select } \land n = u) \Rightarrow \lnot (n = 0 \lor \\
T_0 \text{ after accept } e_1 \lor (T_0 \text{ at select } \land n < u)). \)

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 \( e_1 \) is accepted which inevitably decrements the value of \( n \). By applying induction on the value of \( u \) to Lemma C we obtain

- \( (T_0 \text{ at select } \land n = u) \Rightarrow \\
\lnot (n = 0 \lor T_0 \text{ after accept } e_1) \).

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

To proceed from \( n = 0 \) to total termination is straightforward.

**Conclusions and Discussions**

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 synchroniza-

...
"T 4 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 compatibility 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 c1, ..., cn, one for each task. These will count the number of operations, measured in some basic units, performed by the task Ti since the last time-step transition. They are reset to zero on each time-step transition, and incremented whenever task Ti 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 ci such that ci < 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


task $T_0$ is
  entry $e_1 (id_1 : in INTEGER ; ret_1 : out BOOLEAN)$ ;
  entry $e_2 (id_2 : in INTEGER ; ret_2 : out BOOLEAN)$
end $T_0$

task body $T_0$ is
  n : INTEGER := 1
begin
  loop
    select
      accept $e_1 (id_1 ; ret_1)$;
      if $id_1 = 1$ then $n := 0$
      elsif $n > 0$ then $n := n+1$
      ret_1 := $(n>0)$
      end $e_1$
    or
    accept $e_2 (id_2 ; ret_2)$;
    if $n > 0$ then $n := n-1$
    ret_2 := $(n>0)$
    end $e_2$
    or
    terminate
    end select
  end loop
end $T_0$

task body $T_1$ is
  a : BOOLEAN ;
begin $e_1 (1,a)$ and $T_1$

task body $T_2$ is
  b : BOOLEAN := true ;
begin while b do
  loop $e_1 (2,b)$ and loop
end $T_2$

task body $T_3$ is
  c : BOOLEAN := true
begin while c do
  loop $e_2 (3,c)$ and loop
end $T_3$

FIGURE 1