July 1981

Report No. STAN-CS-81-867

ADAM - An Ada based Language for Multi-Processing

by

D. C. Luckham, H. J. Larsen,

D. R. Stevenson, F. W. von Henke

sponsored by

Defense Advanced Research Projects Agency

Department of Computer Science

Stanford University
Stanford, CA 94305

ADAM
An Ada based Language for Multi-processing

D. C. Luckham, H. J, Larsen, D. R. Stevenson, F. W. von Henks

July 1081

Abstract:

Adam is an experimental language derived from Ada. it was developed to facilitate study of issues in Ada implementation. The two primary objectives which motivated the development of Adam were: to program supervisory packages for multitask scheduling, and to formulate algorithms for compilation of Ada tasking.

Adam is a subset of the sequential program constructs of Ada combined with a set of parallel processing constructs which are lower level than Ada tasking. in addition, Adam places strong restrictions on sharing of global objects between processes. Import declarations and propagate declarations ate included.

A compiler has been implemented in Maclisp on a DEC PDP-10. It produces assembly code for a PDP- 10. It supports separate compilation, generics, exceptions, end parallel pt ocesses,

Algorithms translating Ada tasking into Adam parallel processing have been developed and implemented. An experimental compiler for most of the final Ada language design, including task types and task rendezvous constructs, based on the Adam compiler, is presently available on PDP-10's. This compiler uses a procedure call implementation of task rendezvous, but will be used to develop and study alternate implementations.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 003-80-C-0150.

CONTENTS

1. INTRODUCTION 4
   1.1 Notation **and Conventions** 6

2. OVERVIEW OF Adam 7
   2.1 Types for Scheduling 7
   2.2 Exceptions 7
   2.3 Modules 8
   2.4 Processes 8
   2.6 Program Units, Visibility and Imports 9
   2.6 Adam Runtime Environment 11

3. TYPES AND DECLARATIONS 12
   3.1 Object Declarations 12
   3.2 Types and Subtypes 13
   3.3 Predefined Types 13
   3.4 Types for Scheduling 13
   3.6 Machine Dependent Types 15

4. STATEMENTS 16
   4.1 Reserve Statement 16
   4.2 Initiate Statement 16

6. SUBPROGRAMS 17
   6.1 Propagate Declarations 18
   6.2 interrupt Procedures 19

6. MODULES 19
   6.1 Schedulers 20
   6.2 Devices 21
   6.3 Scheduled Modules 21
   6.4 Instantiation 23
   6.6 Examples 23

7. PROCESSES 28
   7.1 Channels Declarations and Use Clauses 29
   7.2 Initiation of Processes 31
   7.3 Termination of Processes 31
   7.4 Instantiation of Processes 32

8. VISIBILITY RULES and IMPORTS 33
   8.1 Visible Declarations 33
   8.2 Declarations Requiring Importation 33
   8.3 Imports 33

9. CONCLUSIONS 36
9.1 Writing Supervisors 36
8.2 Transporting the Adam Compiler **and Runtime Environment** 37
9.3 Translating Ada Multitasking 37

REFERENCES 40

APPENDICES 42
A Syntax 42
B Reserved Words **61**
C List of Pragmas **61**
D Predefined Attributes 62
E Predefined Exceptions 62
F A Standard Supervisor 63
G Ada Multitasking Translation **Example** 63
H Compiler **Commands** 68

1. INTRODUCTION.

Adam is an experimental multiprocessing language based on **Ada. It** consists of a **large** subset of the non-tasking constructs of the Ada language **[6],** augmented by some simple primitive constructs for scheduling and parallelism. The sequential subset includes **Ada** packages, generic units, and exceptions; the omissions **have** mainly **to do** with **numeric** types.

Adam is intended to remain as close as possible to the final Ada language design while facilitating:
(i) construction of schedulers and **runtime** supervisors for multitask programs **intended to** run on either single or multiple processor hardware,
(ii) formulation of translation algorithms for Ada tasking, and **for other** high **level parallel** constructs, and
(iii) specification of parallel programs.

The additional constructs for parallelism are lower level **than Ada tasking. These** constructs **include** (1) units, called processes, which **may execute in parallel, (2)** constructs for communication among processes, specifically scheduled modules, which **are** packages that schedule **access** to their visible procedures, **and (3) the predefined types** *Locks, Process Names,* and *Condition Variables.* Some of these constructs **are related to** concepts in Concurrent **Pascal** [1] and Modula **[13].** In every **case,** their **compilation is well** understood.

The reader might well ask why we feel it is necessary to deviate from **Ada.**

Goal **(i)** is motivated by the prediction that users of Ada tasking (or indeed any high level multiprocessing language) will need to modify **"standard" runtime** scheduling **and** supervisory packages to suit their own needs. (The reader who doubts this **should** consider, e.g., the design philosophy changes that took place between references **[1] and [2].** Certainly, if one man can go through such changes, how widely might two men, an implementor and a user, disagree on the necessary language constructs for scheduling and their implementation? There are many other publications discussing this problem, e.g., **[5]** and **[9].)** It is therefore necessary to study the structure of **runtime supervisors and to** develop languages **that** facilitate their construction.

Why not use Ada? Most supervisory packages can be written **in Ada provided the** programmer obeys a very strict discipline in coding critical regions, **and is content to** simulate low level protection with high level constructs. Some **simple additions to Ada** might help. Adam attempts to alleviate some of the burdensome discipline by introducing the *scheduled module* **and** *tesetve* constructs, **and also provides the very low level** scheduling objects, *Locks* **and Process** *Names.*

Additional problems arise in writing a supervisor in Ada if it is itself **a parallel program (as** might well be the case in a multiprocessor system, or indeed must be so if it uses interrupts). This requires coding a **"subsupervisor"** which in turn would provide scheduling for tasks in the main supervisor. But the Ada task rendezvous **semantics ([6], 9.6) implies** a very rigid scheduling algorithm. So tasks in the supervisor **have to have a special** semantics (e.g., as interrupt entry procedures, [6], 13.5.1). In Adam parallelism is expressed by the process construct, the semantics of which do not imply any particular scheduling. Also, because process names are objects in Adam, a subsupervisor may be easily coded to schedule the processes in the main supervisor. (c.f., discussion of programming process scheduling in [5].)

Goal (ii) became a concern during the study of the preliminary Ada tasking design. it very soon became clear that the informal semantics of task rendezvous given in the preliminary Ada Rationale was by no means the only viable method of implementation. indeed, translation algorithms for high level parallelism based on CSP [4] or similar concepts appear to require formulation and analysis. A promising approach is to implement these algorithms at a very high level, i.e., as translations from Ada to an existing high level parallel language. This should result in (a) precise definition of translation algorithms, (e.g., as input/output relations between Ada source and programs in the target language), and (b) the possibility of formalizing correctness of the tasking translation algorithms.

To do this, the semantics of parallelism in the target language has to be already well understood. Existing languages with clearly defined parallelism included Concurrent Pascal and Moduia. So the parallelism in Adam is closely related to processes in these two languages.

Goal (iii) is concerned with the accurate description of parallel programs. How should a programmer document (by comments, formal specifications, or whatever) the intended behaviour of an Ada program, especially one containing packages or tasks? What descriptive facilities should the language provide, what restrictions should be enforced? This is an area of research that is relevant to such immediate questions as how to teach Ada and how to develop standards of documentation for Ada. (Some of these questions are being studied separately, e.g., in [8], [10] or [11].) There are also many possible longer term consequences; these might Include techniques for formal verification and automatic generation of programs from their specifications.

One immediate reason for considering goal (iii) in the Adam design is to ensure that the translations of Ada tasking conform to a structure and discipline which makes them easy to specify and anaiyse. Since the translation algorithms are themselves defined in terms of input/output relations between Ada and Adam, this clearly affects the formulation and study of their correctness.

The crucial question is exactly how much weight to give goal (iii) in the design decisions, particularly when it appears to conflict with the goal of useability. In Adam we chose a form of parallel processes such that all interactions between processes can be deduced from their declarations and instances. To do this requires changing the visibility rules of Ada. In Ada these rules permit undeclared use of global objects, e.g. between tasks, in packages, or in exception handlers, which makes any attempt at precise documentation difficult. Their generality (or permissiveness) is a pitfall to the uninitiated and an area where good programming practice should be developed and taught. Here, Adam simply enforces restrictions that prevent processes from communicating (i.e., influencing each others computations) in arbitrary ways.

The discipline enforced by Adam includes: (a) restricting the visibility rules of Ada, (b) limiting the interactions among processes and requiring that such interaction be explicitly declared in process specifications, (c) requiring import declarations to specify the use **of** global units in modules, and (d) requiring propagate declarations **to** specify **the** exceptions propagated by **a** unit.

The Adam restrictions can In fact be followed in Ada by disciplined programming; but **the** programmer will have to invent **his** own commentary to specify what he is doing, **and his** own methods of checking that he does it. On the issue of useability, the value of these restrictions in terms of whether they help or hinder current programming techniques remains to be studied.

We can report the following progress:

An Adam compiler has been running on DEC PDP-10 computers since June 1980. This compiler is implemented in Maciisp and generates PDP-10 assembly **language. The** compiler provides a small set of commands permitting users to manipulate library files **for** separate compilation. **Runtime** supervisor packages have been written in Adam, compiled, and now form part of our Ada **runtime** environment (an example is given in Appendix F). So **far, these supeyisors** schedule processes on a single CPU; multiprocessor supervisors **have** not yet been **constructed. A** discussion of experience in transporting **the compiler** and environment is given in section Q, and a short description of the compiler facilities is **in** Appendix H.

Three algorithms for translating Ada task types and tasking constructs into **the lower level** Adam processing have been defined; an example is given in Appendix **G, a** description of the algorithms is given in **[12],** and a detailed report is forthcoming. One of these translation algorithms has been implemented in the compiler **(as a** subfunction **of the static** semantic checking). Many examples of Ada tasking programs have been compiled **and run** using separately compiled i/O and supervisor packages; the compiler **has been used in** teaching courses on **Ada** programming. Implementation of the other **translation algorithms is** in progress, and experiments comparing their **runtime** characteristics **are** planned.

1.1 Notation and Conventions.

This report is addressed to readers who already have some familiarity with **the Ada** reference manual **[6].** The notation and formatting conventions of **Ada are adopted with a** few changes. This report describes those constructs of Adam that **are not in Ada.** Constructs common to both languages are only listed. Description **of constructs is by an informal general format and** examples; BNF syntax is given in Appendix A.

in both general formats and BNF the following notation is used.

Square brackets, **[ ], indicate** an optional construct. Curly **brackets, { }, indicate zero or** more repetitions of **a** construct.

Terminology. Modules **and** processes are called *units.* Nongeneric **declarations are often called** actual. Variables and actual modules are called *objects.* **Elements declared in the**

specification - or visible - part of a module are said to be *exported* by the module. Elements declared in the body of a module are said to be encapsulated by the module.

2, OVERVIEW OF Adam.

Adam consists of a large subset of the non-tasking constructs of the Ada language [6], augmented by some simple primitive constructs for scheduling and paraiieiism. This section presents a brief overview of the scheduling and parallel features together with the rationale for them. Syntax and examples are given in later sections.

2.1 Types for Scheduling.

The type differences between Ada and Adam are described in Section 3. The type facilities in Adam are not as rich as Ada, the general philosophy being not to include facilities that are not essential to studying the construction of multiprocess programs.

However, Adam includes some new types that are important in writing schedulers and process supervisors.

(i) *Locks.* Variables of type Lock provide a low-level facility for programming critical regions.

(ii) Process *Names.* Processnames provide a means of refering to the process (or thread of control) currently executing an instruction. This facility is important in programming scheduling of shared variables (or any interaction between processes), supervision of resources, and message-passing operations.

(iii) *Conditions.* Condition variables provide a fifo queue of process names.

2.2 Exceptions.

Exception handling and propagation is the same as Ada except that there is no direct propagation *of* exceptions between processes. in addition, ail exceptions propagated from units must be declared in the specification part of the unit by means of propagate declarations (this can be practiced in Ada using the comment facility - see [8]).

Propagate declarations ensure that a calling unit will not receive any "surprise" errors. Their use permits compiletime checking (a) that the propagate declarations are consistent with the set of exceptions raised but not handled in the unit body, and (b) that exceptions are not propagated outside of the scope of their declaration, a somewhat ambiguous situation permitted in Ada, The use of propagate declarations in a method of specifying programs with exceptions is described in [10].

2.3 Modules,

Modules in Adam correspond to packages in Ada. They provide facilities for encapsulation or abstraction. in addition Adam provides special modules:

(i) Device *modules:* Devices are intended for interfacing with hardware. Devices may contain machine coded operations and interrupts.

(ii) *Schedulers:* Schedulers are used to encapsulate scheduling and synchronization operations.

(iii) *Scheduled modules* (and devices): units for commmunication between processes. A scheduled module is simply a module containing a scheduler; its visible operations are associated with scheduler operations by scheduling declarations. Details are in Section 6.

Generic modules-are declared and instantiated as in Ada.

2.4 Processes.

Processes are program units which may be initiated and run in parallel. The major difference from Ada tasks is that the constructs for communication between processes are essentially lower level than the Ada Rendezvous, and are more restricted.

Processes communicate by operating on scheduled modules. These modules are called communication channels and are declared by means of a channels declaration in the specification part of process declarations. The *Monitor* construct of Concurrent Pascal [1] and the *mailbox* concept of Gypsy [3] are examples of scheduled modules.

There are no means of communication among processes, other than channels; in particular, processes do not import values (see 2.5), may not reference external values, and do not have exports.

Execution of a process is begun by means of the initiate statement. Multiple initiations of a process are permitted; each initiation results in a new copy of the process. initiation is the only operation on processes. A process terminates when it reaches the end of its body. A scope may be left only when all dependent processes have terminated (see 7.3).

Processes may be generic (but may have only type and in parameters). *A generic* process may be Instantiated to an actual process.

The choice of communication constructs in Adam has several consequences and implications. First, the Ada rendezvous constructs, including entry procedures and accept and select statements, are omitted. Second, all communication among processes in a system can be determined from the specification parts of the process declarations. (in

Ada the bodies of consumer tasks must be examined to determine which service tasks they communicate with.) It Is expected that this will be of some advantage in developing specifications for multiprocess systems. Third, the Adam programmer has to construct schedulers which determine interaction with the **runtime** supervisor in **a** multiprocess system. This may be burdensome for high level parallel processing applications, but should be close to the kind of programming required in constructing embedded computer **systems.** For high level multiprocessing it is expected that standard library communication modules with underlying supervision will be available in a separately compiled units. Fourth, **Ada** multitasking systems can be translated into Adam multiprocessing **[12].** Different translation algorithms exist, each having advantages in **runtime** efficiency depending on the system to be translated, the facilities exported by the **runtime** supervisor, **and the** hardware.

2.6 Program Units, Visibility and imports,

The visibility rules of Adam differ from those of Ada. The motivation for the difference is to ensure that all interactions between processes are deducible from the process declarations. There are two changes from Ada. First, the visibility of outside objects within **a** process, (i.e., the Adam version of a task) is restricted to be exactly the scheduled modules declared as communication channels of the process. No other outside objects may be mentioned within a process. Second, all outside objects mentioned inside **a** module must be declared as *imports* of the module. As a consequence of these **two** changes, all objects by means of which processes may be able to influence **each others,** computations can be enumerated by taking the transitive closure **of** the **channels, their** imports, imports of imports, and so on.

it remains to motivate the definition of *object.* The elements out of which **Adam programs are** constructed are declarations (of types and subtypes, variables, **constants,** subprograms, modules, and processes) and statements.

The elements that **may be** used to communicate are those that have accessible values or states that may vary during a computation, **i.e.,** variables and actual ***modules.*** **These we** define to be *objects.*

Elements that are purely definitional in nature and do not have states **are type,** subtype and generic unit declarations. *(Note:* Adam visibility rules for generic units ensure **that they are** purely definitional.) These cannot be used to communicate when shared between processes, and their visibility is exactly as in Ada.

Processes have states but their states are not accessible by **other units. The only action** that can be performed on a process is initiation. The only manner **in** which **an** external unit may effect the state of an initiated process is to operate on **a** channel shared with the process. So the visibility of processes is also unrestricted,

Finally, the treatment of subprograms results from **a** compromise with useability considerations. A subprogram declaration is definitional unless there are global objects. We could require imports declarations on subprograms, in which case their visibility **need not** be restricted. However, the introduction of modules into a programming language appears to change the role of subprograms from basic unit (as in Pascal) to small subunit of a module. The unencapsuiated subprogram becomes a rarity, But in its new role as building block of modules, it is normal for a subprogram to import the local data of a module body. So imports declarations would then be part of most subprograms. But the imports declarations on subprograms internal to a module body are invisible outside the body and no longer contribute to the specification of outer systems of processes. So instead of requiring imports declarations on subprograms, we have restricted the visibility of subprograms (which is what seems to be happening naturally anyway).

We may now summarize the visibility rules of Adam for units.

Declarations that are always visible within a unit if they are visible at the point of declaration of that unit are:
(i) type declarations (including the constants of an enumeration type),
(ii) constants,
(iii) generic unit declarations,
(iv) processes,
(v) exception declarations.
(vi) predefined system modules (e.g., process supervisor - 2.6.1).

Declarations that are not visible unless explicitly imported into **a unit:**
(i) variable declarations,
(ii) nongeneric module declarations (including instantiations of generic modules).

Note: Thus, outside unencapsulated subprograms are never visible inside units.

Channels
The channels declaration of a process permits an actual scheduled module to be visible inside a process body. This is the only kind of external object that can be made visible inside a process.

Imports
import declarations can appear only in nongeneric modules. Such declarations permit a list of outside objects to be visible inside a module specification or a module body. if a module declaration mentions external objects, those objects must be declared as imports.
in the case of an instantiation of a generic module, those generic parameters that are objects are imports. A separate imports declaration is omitted. (See Section 8.3.)
Note: A generic module may not have imports.

Notes:

1. in defining those objects whose importation must be declared, there are two possible approaches. Make every identifier an object, as in Euclid. This is very simple but leads to lengthy and irrelevant imports lists containing many "objects" that cannot possibly be used to communicate between parallel computations. Alternatively, define as objects exactly those elements that a process may be able to use to influence another's computation. This requires more complex visibility rules and forces the programmer to think, but leads to more relevant lists of imports. We take the second approach,

2. Construction of the transitive closure of channels lists and imports lists is easily automated and may prove useful in checking for some common errors,

3. Ada with clauses function as imports declarations between separately compiled units.

4. Essentially, the stricter visibility rules enforce a discipline in Adam that can be practiced in Ada.

2.6 Adam Runtime Environment

2.6.1 Process Supervisor.

One of the design goals of Adam is to provide a language for writing process scheduling, often called ***supervision*** here. Consequently, the semantics of the basic multiprocessing constructs of Adam do not assume calls on an underlying supervisor.

However, it is important to define a minimal expected interface of operations to be provided by most supervisors. This facilitates programming scheduling of processes (since scheduling often involves supervisor calls) and substitution of new supervisors into multiprocess programs.

The Adam compiler assumes the presence of a predefined scheduled module, ***supervisor***, that implements a set of visible standard procedures for activating and suspending processes:

```
procedure SUSPEND;
procedure ACTIVATE (P : PROCESSNAME);
procedure SWITCH (P : PROCESSNAME);
procedure START    (D : INIT_DATA);
procedure FI NI SH;
```

(The types PROCESSNAME and INIT_DATA (process initialization data) are discussed in Section 3.6.) The supervisor procedure START is called when an initiate statement is executed; it sets up the proper entries in the supervisor tables and activates the process. When a process has reached the end of its body, the supervisor procedure FINISH is called. A call to SUSPEND results in the suspension of the calling process and (normally) the running of another process in its place. The procedure ACT I VATE reactivates a process after suspension. The procedure SWITCH causes a context switch from the calling process to process P, i.e. suspends the calling process and activates P. Examples of these supervisor procedures are given in Appendix F.

We assume that any Adam environment contains a predefined supervisor (cf. Appendix F). Calls to START and FIN1 SH are generated by the compiler; these procedures have to be present in every supervisor. The other standard supervisor procedures are most often called directly by scheduling procedures in user programs. However, when processes are nested the compiler may also have to generate calls to SUSPEND **and** ACTIVATE, **e.g. for** synchronizing termination of outer and inner process,

The interface of standard supervisor procedures is ail the compiler **knows about** supervision. it is thus easy to substitute a user-written supervisor **that conforms with the** interface for the standard one and have it used in the compilation **of** multiprocess programs. A pragma, SUPERVISOR, notifies the compiler to substitute **ceils to procedures of the** same name from **a** new module for **calls to the standard supervisor procedures. For instance, the pragma,**

pragma SUPERVISOR (M);

where M is **a** module name, **will** result in the replacement of all calls to ACTIVATE **by calls to** M.ACTIVATE in the compilation, and similarly for the other expected procedures **in the supervisor** interface. (Obviously, the supervisor pragma **has to appear in the program text before any** calls to supervisor procedures are to be compiled.)

A user-supplied supervisor may be more sophisticated and implement **additional** procedures. For instance, the supervisor presented in Appendix **F supports** multiprocessing on a one-processor installation; a supervisor for programs intended **to run** on multiprocessor hardware must deal with additional problems like possible time **races. On the other hand,** in some applications, e.g., where **processors are dedicated to single processes,** the user-supplied supervisor may be trivial **and provide only the** START **and** FINISH procedures.

2.6.2 Input-Output.

The Adam runtime environment also includes a predefined module **for input and output to** files **and** terminals. The module provides an implementation **of the standard package for I/O as defined in Section 14 of** the Ada Reference **Manual [6].**

3. TYPES AND DECLARATIONS.

3.1 Object **Declarations.**

In Adam objects are declared as in Ada:

```
identifier-list : [constant] type [:= expression];
```

3.2 Types and Subtypes.

The type structure of Adam is derived from that of Ada with the following **major differences** and restrictions. Range constraints must be static. There is only **one Integer type. Float, and** fixed types **are not** implemented. There are no type conversions.

For the syntax of type definitions refer to Appendix A.

3.3 Predefined Types.

The predefined type identifiers of Adam include the following subset **of** the predefined **Ada language environment:**

```
type INTEGER is implementation-defined;
type NATURAL is INTEGER range 1..INTEGER'LAST;
type BOOLEAN is (FALSE, TRUE);

type CHARACTER is (NUL,...,'a',...,'~');
type STRING    is array (NATURAL range <>) of CHARACTER;
```

3.4 Types for Scheduling,

In addition to the standard Ada types listed above, the following new **types are** implemented in Adam. Variables of these types **are** intended **to facilitate the writing of** scheduling and synchronization,

3.4.1 Locks,

Variables of type LOCK are used to implement primitive mutual exclusion. **For purposes of** description, locks **may** be thought of as being in one of two states, **ON and** OFF. **There are three** procedures that may be applied to locks :

TEST-SET (L: **in out** LOCK; B: out BOOLEAN)	- gains exclusive **access to L; if L is** OFF changes L to ON and sets B to TRUE, else **sets B to** FALSE.
SET (L : **in out** LOCK)	- busy waits until L is OFF, then **gains exclusive access to** L and changes the state to ON.
RESET (L : **out** LOCK)	- gains exclusive **access to** L, **then changes state of L** from ON to OFF.

These are the only operations that may be applied to locks.

Example 3-2: The <body> is protected by L from simultaneous **execution.**

```
L :  LOCK;           -- variable L is declared to be a lock with Initial state OFF.
begin
      SET (L);        -- busy wait to gain access to L and <body>.
      <body>
      RESET(L);       -- reset L so next user may gain access.
end;
```

3.4.2 Processnames,

Values of type PROCESSNAME are created by the initiate statement. **These values are** distinct. This is the only **way** that processnames can be created. The only permissable operations are assignment, equality, and selection using MYNAME which returns **the name of** the thread of control executing the call to MYNAME.

The type PROCESSNAME **also** includes a constant, **null, which is not associated with any process.**

3.4.3 Conditions.

Values of type CONDITION are a FIFO queue **of processnames. Variables of type** CONDITION are called condition variables.

The operations on condition variables are as follows. Ail of these operations **are indivisible** (e.g., a possible implementation of indivisibility is to protect **operations on each variable of type** condition by disabling interrupts and locking the operations).

Selectors and constructors:

```
function EMPTY (CV : CONDITION) return BOOLEAN
    -- returns TRUE If the queue of CV is empty; initial value is TRUE.

procedure INSERT (CV : in out CONDITION; P : PROCESSNAME)
    -- inserts P on the queue of CV;
    -- raises CONDITION-QUEUE-FULL exception If the queue of CV is full.

procedure REMOVE (CV : in out CONDITION; P : out PROCESSNAME)
    -- removes the first processname from the queue of CV and returns it as the
    -- value Of P;
    -- raises CONDITION-QUEUE-EMPTY except/on if the queue of CV Is empty.
```

Example 3-3: Two synchronization operations coded using condition variables.

The following two procedures are typical operations used to implement schedulers **for** modules in a multiprocessor environment. They include **both** decisions to queue **(or** dequeue) processes and calls to the process supervisor. They **are, in** turn, **protected by** locks.

```
procedure WAITFOR(CV : in out CONDITION; CVL : in out LOCK) is
                        -- tests some condition followed by a queuing operation and a
                        -- supervisor call if the test is FALSE.
                        -- CVL should be a unique lock protecting all operations on CV.
begin
   SET (CVL);
   if <some-condition> then
      INSERT (CV,MYNAME 0 );
      RESET (CVL);
      SUSPEND;
   else
      RESET (CVL);
   end if;
end;

procedure SIGNAL (CV : in out CONDITION; CVL : in out LOCK) is
                        -- removes the first processname from the queue CV (If nonempty) and
                        -- actlvates lt. CVL is a unique lock protectlng all operations on CV.
   P :  PROCESSNAME;
begin
   SET (CVL);
   if <some-condition> and not EMPTY (CV) then
      REMOVE(CV,P);
      RESET (CVL);
      ACT1 VATE (P);
   else
      RESET (CVL);
   end if;
end;
```

Another example of use of condition variables is given in Section 7.6, Example 7-4.

3.6 Machine Dependent Types.

The initiate statement interfaces with the runtime supervisor by passing a record of information about the initiated process. The structure of this record is implementation dependent. For our PDP-1 0 implementation it has the form:

```
type INIT_DATA is
   record
      PNAME      : PROCESSNAME;
      CODESTART  : ADDRESS;
      STKSTART   : ADDRESS;
      PRIORITY   : PRIORITY;
   end record;
```

4. STATEMENTS.

The statement syntax of Adam is taken from that of **Ada** with some minor **differences and** additions. All of the sequential statements of **Ada are provided.**

Since the multitasking in Adam differs from that of Ada there **are no delay, abort, select, accept, or terminate statements in Adam.**

4.1 Reserve Statement.

The reserve statement is used to reserve a scheduled module. It allows a process to perform a sequence of operations on the module without any intervening operations by another process on the same module, It is intended for use when the number of operations is determinable only **at runtime.**

The form of a reserve statement is:

```
reserve scheduled-module-name do
     statement-list
end reserve;
```

Example 4- 7: Printing a file of arbitrary length.

```
reserve LPT,DRIVER do
     loop
          ...                           -- get next line of file
          LPT,DRIVER. PRINT (...);      -- print it on the line printer
     end loop;
end reserve;
```

The **compilation of reserve statements** using the REQUEST **and** RELEASE **operations of the module's scheduler is described** in Section 6.3.2.

4.2 initiate Statement.

The **initiate statement is used to cause a process to begin its execution. The general form of the statement is:**

```
initiate 1 ist-of-process-names;
```

A process may be initiated any number of times, with each initiation causing a new copy of the process to begin execution,

6. SUBPROGRAMS.

A subprogram is either a procedure or function **as in Ada. Adam includes the generic** facility of Ada.

The main differences from Ada are required declarations in the specification part of a subprogram declaration. The new **required** declarations specify exceptions, **and scheduling.**

(1) If the subprogram is part of a **scheduled module,** it may be linked to procedures **of a** scheduler by a scheduling **declaration - see Section 6.3.1.**

(2) All exceptions that may be propagated by the subprogram **must be** named **in a** propagate declaration. Such a declaration must be within **the scope of the exception.**

The form for a subprogram declaration is:

```
subprogram-declaration ::=
     subprogram-header;
   | generic-subprogram-declaration        -- as in Ada
   | generic-subprogram-instantiation      -- as in Ada

subprogram-header ::=
     function designator [formal-part) return subtype-indication
   | procedure identifier [formal-part]
   | interrupt identifier called from number
```

The form **for a subprogram body is:**

```
subprogram-body ::=
    subprogram-header is
        specification-part
        declarative-part
    begin
        statement-list
    [exception
        (exception-handler)]
    end [designator];

specification-part ::=
    [propagate-declaration]
    [scheduling (scheduling-item,   scheduling-item);]
```

where:

propagate declaration - propagate list of exception names;
scheduling-item is either a **procedure call or else null; (see section 6.3.1).**

6.1 Propagate Declarations.

Propagate declarations specify which of the declared exceptions **may** be raised **and** propagated by a subprogram. The others clause may be used in one **propagate declaration** to refer to all unnamed exceptions that may be propagated.

Propagate declarations **may** be annotated, and thus may be used to specify not only **the** exceptions that **may be** propagated, but also the conditions under which **propagation of** exception8 will occur.

Example 5- 7 : Subprogram specifications with propagate **declarations with annotation.**

In the procedure SEARCH, when propagation of the exception, NOT-FOUND, occurs, it is specified that the key, X, is not in array A. This illustrates **a use of exceptions to break** the normal output specification of a procedure into cases.

```
type NARRAY is array (1 . . N) of INTEGER;
. . . . .
NOT_FOUND: exception;

procedure SEARCH (N, X : INTEGER; A : NARRAY; I : out INTEGER) is
    propagate NOT-FOUND; -- 1 <= J <= N => X /= A(J);
            -- annotation states a property of parameter values when propagation occurs.
    -- exit 1 <= I <= N and X = A(I);
            -- exit comment specifies parameter values on normal exit from Search.
begin
            . . . .
    raise NOT_FOUND;
            . . . .
end;
```

5.2 Interrupt Procedures.

Interrupts **are special** procedures that are called directly from **the hardware. Interrupt** procedures can occur only in device modules (cf. Section 6.2). Interrupt **procedures may** not be generic **and may** not have parameters; they **may have** global variables.

6, MODULES.

Module declarations in Adam are the same as for Ada packages except:
(i) Global objects in generic module declarations are not permitted.
(ii) Declaration of global objects imported into nongeneric modules is required.
(iii) Declaration of exceptions that may be propagated is required.

(iv) Scheduling declarations are used in a scheduled module body **to associate visible** operations with scheduling operations.
(v) Different kind8 of modules can be defined:

module	- basic module declaration,
device	- bodies of devices may contain machine **code and interrupts,**
scheduler	- provide8 procedure8 for scheduling,
scheduled module	- **a** module containing a scheduler; this is **the program unit for** communication between processes.

Separate compilation of module specification and body is Supported. **The general format for module declarations is:**

```
[generic generic parameter 1 ist]
module M is
    [Imports (imports list); ]
    exception declarations         -- declaration of those exceptions propagated
                                   -- by the procedures and functions if they
                                   -- are not already declared;
    type declarations              -- including private types,
    procedure specifications       -- the visible procedures and functions are
    function specifications        -- called the module operations,
    module specifications

  [private . . . ]                 -- private part
end [M];
  . . .
module body M is
    [imports (imports list); ]     -- encapsulated imports, see Section 8
    type declarations
    variable declarations
    other declarations             -- must include the bodies of all operations
                                   -- and modules specified In the visible part
end [Ml ;
```

Note:

An imports list is **a** list of variables and actual **modules.** Each import must be **visible** at **the** point of declaration. It is then imported or made visible inside **the declaration. Each** imported variable has a mode **in,** out, or in out), and each imported module **has the nature module). See** Section **8.**

Example 6-7: Adam version of the visible part of the ON-STACKS **example from the Ada** Reference Manual **[6], p.** *12-5.* Some annotations are included.

```
generic
    SIZE : INTEGER;
    type ELEM is private;
  --entry SIZE > 0;                -- assertion on gener lc parameters.
module ON-STACKS is
```

```
type STACK is private;
OVERFLOW, UNDERFLOW : exception;  -- visibility of exceptions is the same as
                                  -- the module declaration.
procedure PUSH (S: in out STACK; E: in ELEM);
   propagate OVERFLOW; -- full (S);

procedure POP (S: in out STACK; E: out ELEM);
   propagate UNDERFLOW; -- empty (S);

-- comments specifying visible operations PUSH and POP, and full and empty.

private
     type STACK is
       . . .
end ON_STACKS;
```

Note; The formal annotation of modules is currently a topic of research. A discussion of annotation of Ada package8 can be found in [11].

6.1 Schedulers.

Schedulers are intended to encapsulate both the synchronization and protection for scheduled modules shared between processes. Schedulers implement (a) the scheduling procedure8 for entry to, and exit from module operations, (b) procedure8 REQUEST and RELEASE (for the `reserve` statement), and (c) procedures for synchronization between module operations. Scheduler procedure declarations follow the normal format for procedure8 (Section 5), except they may not contain scheduling declarations; REQUEST and RELEASE do not have parameters.

A nongeneric declaration of a scheduler must be in the body of a scheduled module. Conversely, a scheduled module must contain exactly one scheduler. Generic schedulers may appear in any declarative part. An instance of a scheduler may be declared only in the body of a scheduled module.

Example 6-2: A common format for scheduler declaration8 within scheduled modules is:

```
scheduled module M is
   . . .                          -- specification part of M,
end M;

scheduled module body M is
   . . . . .                      -- local variables of M,
  scheduler SCHED is              -- schedule; for M.
     imports (list of local variables of M);
     procedure REQUEST;           -- REQUEST and RELEASE are procedures
     procedure RELEASE;           -- used for scheduling reserve (Sec. 6.3.2).
     procedure ENTER (...); -- ENTER, LEAVE, DELAY, . . . are other
     procedure LEAVE ( . . . ); -- scheduling procedures exported by SCHED
     procedure DELAY ( . . . );
  end SCHED;
```

```
    . . . . .                       -- body of M with scheduling declarations
                                    -- (see Section 6.3.1) for each of the visible
                                    -- operations of M.
                                    -- body of SCHED.
  end M;
```

6.2 Devices.

Device modules are the only program units that may contain machine code and interrupt procedures. They are intended to encapsulate the machine-dependent parts of a system. Devices may be generic if they do not contain interrupts.

6.2.1 Machine code.

Machine code is inserted into a program through the use of an aggregate as in Ada. (An example is given in Appendix F.) Unlike Ada, both machine code and Adam statements may appear in the same subprogram.

6.2.2 Interrupts,

Interrupt procedures are declared by:

```
   interrupt P called from number is
     . . .                          -- procedure body as in 6.3
```

Notes:
1. Interrupt procedures may not be generic.
2. Interrupts may not have parameters, but may reference global variables. See example 6-5, Section 6.5.
3. A separately compiled device which has an interrupt procedure in its body must contain a subprogram header for that procedure in its private part.

Remark: Specifications for interrupt procedures are certainly inadequate. For example, it would be useful to be able to name a procedure that must be called before the interrupt can be enabled and which will resume whenever the interrupt is disabled.

6.3 Scheduled Modules,

A scheduled module is a module whose visible operations are scheduled by a scheduler local to its body. Scheduling of operations is declared by scheduling declarations. If a scheduled module is named in a reserve statment, then the scheduler for the module must provide REQUEST and RELEASE procedures.

6.3.1 Scheduling Declarations,

Let **p be** an operation of a scheduled module, A scheduling **declaration for p has the format,**

```
procedure P (parameter list) is
    . . .
    [scheduling (scheduling list); ]
    . . .
```

where
scheduling list has the form: **p1(L1), p2(L2)**
and each pi is either **a** visible procedure of the local scheduler or else is **null.**
p1 - scheduling before entry to p,
p2 - scheduling on exit from p,
Li - parameter lists.

The effect of a scheduling declaration for a procedure P in a module with **a** scheduler S is **that the body of** P, **is compiled as**

```
s. pl (L1);          -- omitted if p 1 (L1) is null.
<body of P>
s. p2 (L2);          -- omitted if p2(L2) Is null.
```

Notes:

1. Within the body of a scheduled module with scheduler, S, calls to a scheduler procedure, **p,** are stated in the usual format, **S.p,** unless they **are** in the scope of a **"use S"** clause.
2. Scheduling declarations permit specification of **"before"** and **"after"** scheduling. **The** set of scheduling declarations specifies explicitly the scheduling **of entrance to and** exit from the boundary of the scheduled module.
3. Internal synchronization between module operations cannot **be declared by this** mechanism. For this, one must still use calls to scheduler procedures.

6.3.2 Reserve Statements.

Let M be a scheduled module with scheduler S.

```
reserve M do statement-list end reserve;
```

is compiled **as**

```
S. REQUEST;
statement-list
S. RELEASE;
```

if REQUEST **and** RELEASE are not supplied by the scheduler, attempts **to compile reserve statements for the module** will result in an error message.

6.3.3 Exceptions in Scheduled Modules.

if an exception which is unhandled reaches the outer level of a scheduled **operation the operation% exit procedure is** run before the exception **is propagated out of the operation.**

6.4 Instantiation.

Instances of generic modules are declared as in Ada:

```
module M is new N(L);
```

where N is a generic module and L is a list of actual generic **parameters. Similarly for scheduler and scheduled module.** Each new instance of a generic scheduled **module or** device has a new instance of the local scheduler.

6.6 Examples.

Example 6-3: Buffer module. A buffer is a typical example of **a** scheduled module. **We give** first a very simple version; the example is presented **in two stages, first the top level** structure showing the scheduling, then the implementation of the scheduler.

```
generic
    BOUND :   INTEGER;
scheduler BUFFER,SCHED is
    procedure FOO,1;                  -- f lrst scheduler operation,
    procedure FOO,2;                  -- second scheduler operation,
    procedure FOO,3;                  -- third scheduler operation.
end BUFFER,SCHED;

generic
    type ITEM is private;
    SIZE :   INTEGER;
scheduled module BUFFER is
    procedure READ (X: out ITEM);
    procedure WRITE (Y: in ITEM);
end BUFFER;

scheduled module body BUFFER is
        . . .                              -- declaration of local variables of BUFFER,
   scheduler 'SCHED is new BUFFER,SCHED (SIZE) ;  -- declaration of scheduler,

   procedure READ (X : out ITEM) is
       scheduling (FOO_1, FOO_3);      -- scheduling for READ - see below,
       . . . .

   procedure WRITE (Y : in ITEM) is
       scheduling (FOO_2, FOO_3) ;     -- scheduling for WRITE - see below,
       . . . . .
```

```
end BUFFER;

scheduled module BIG-BUFFER is new BUFFER (CHARACTER, 120);
                                      -- declaration of an instance of BUFFER.
```

The declaration of BIG-BUFFER will result in the **declaration of a new scheduler that is an instance of** BUFFER,SCHED. The scheduler for BIG-BUFFER **is not named, but conceptually its declaration is:**

```
scheduler BIG,BUFFER,SCHED is new BUFFER,SCHED (120);
```

The effect of the scheduling declarations in BUFFER **is that calls to** BIG-BUFFER **will be compiled as,**

```
BIG,BUFFER,SCHED.FOO,1;          BIG,BUFFER,SCHED.FOO,2;
BIG_BUFFER.READ (X);             BIG,BUFFER.WRITE (Y);
BIG,BUFFER,SCHED.FOO,3;          BIG,BUFFER,SCHED.FOO,3;
```

Example 6-4: **Implementation of** BUFFER,SCHED.

```
scheduler body BUFFER,SCHED is

   PROTECT : LOCK;                        -- local variables of scheduler
   COUNT   : INTEGER range 0 . . BOUND := 0;
   INUSE   : BOOLEAN := FALSE;
   READQ   : CONDITION;                   -- queue for readers
   WRITEQ  : CONDITION;                   -- queue for writers

   procedure FOO,1 is                     -- schedules entry to READ,
   begin
      SET (PROTECT) ;                     -- wait to gain exclusive access,
      if COUNT = 0 or INUSE then          -- BUFFER is empty or in use,
         INSERT (READQ, MYNAME 0) ;       -- place reader on queue,
         RESET (PROTECT);                 -- release BUFFER,SCHED (note 2 below),
         SUSPEND;                         -- supervisor ceil to suspend caller,
      else
         INUSE:= TRUE;                    -- prepare to enter free BUFFER,
      end if;
      COUNT := COUNT - 1;                 -- reduce no. of items in BUFFER,
      RESET (PROTECT) ;                   -- release BUFFER,SCHED.
   end FOO,1;

   procedure FOO,2 is                     -- schedules entry to WRITE,
   begin
```

```
        SET (PROTECT);                -- wait to gain exclusive access,
        if COUNT = BOUND or INUSE then -- BUFFER is full or in use,
           INSERT (WRITEQ, MYNAME()); -- place writer on queue,
           RESET (PROTECT);           --release BUFFER,SCHED (see note 2 below)
           SUSPEND;                   -- supervisor call to suspend caller,
        else
           INUSE := TRUE;             -- prepare to enter free BUFFER,
        end if;
        COUNT := COUNT + 1;           -- increase no. of items In BUFFER,
        RESET (PROTECT);              --release BUFFER,SCHED.
    end FOO,2;

    procedure FOO,3 is                -- schedules exit from READ end WRITE,
       P: PROCESSNAME;
    begin
        SET (PROTECT);                -- wait to gain exclusive access,
        if COUNT > 0 and not EMPTY (READQ) then
           REMOVE (READQ, P);
           ACTIVATE (P);              -- supervisor call to activate a reader
        elseif COUNT < BOUND and not EMPTY (WRITEQ) then
           REMOVE (WRITEQ, P);
           ACTIVATE (P);              -- supervisor cell to activate a wr iter
        else
           INUSE := FALSE;            -- else BUFFER is free
           RESET (PROTECT) ;
        end if;
    end FOO,3;

begin
   RESET (PROTECT);
end BUFFER,SCHED;
```

Notes: .

1. All procedures of BUFFER,SCHED are protected by the same lock, PROTECT. **Only one** -thread of control, P say, can have access to BUFFER,SCHED at **any time. Processes** busy wait to enter BUFFER,SCHED. The implementation of waiting in SET is **not** required to be fair, and this could cause a process to be starved.
2. Each BUFFER,SCHED procedure calls the supervisor. PROTECT is reset before calls **to** SUSPEND. In **a** multiprocessor system this means an ACTIVATE (P) might be executed (by -another thread of control) in FOO,3 before SUSPEND is executed **by p itself in** FOO,1 or FOO,2. This will not cause blocking only if the supervisor **can** remember **en** ACTIVATE that arrives ahead of the matching SUSPEND.

An alternative design of supervisor calls is to permit locks as parameters of ACTIVATE **end** SUSPEND, and require these procedures to reset the lock.

3. Operations of the scheduler, BUFFER,SCHED, and BUFFER may execute simultaneously. However the very simple scheduling in BUFFER,SCHED makes BUFFER a critical region also. It is a simple exercise to change BUFFER,SCHED **so that Read and Write** operations may execute simultaneously in BUFFER.

Example 6-5: Simple device module using interrupts.

The following example demonstrates the use of interrupts and scheduling in device modules, The device module Line-Out is to be used for sending a line of output to a device, such as a line printer, which is initiated by receipt of the first character of the line and which will generate an interrupt when it is ready to accept each succeeding character, A user of the device does. a call to the procedure Send. If the device is already in use, the caller will be put on a wait queue and suspended. The body of the Send procedure performs the initial output to the device and then suspends the calling process via a call to the scheduler procedure Await. The interrupt procedure within the device module performs the output of the remaining characters in the line and activates the calling process upon completion of the IO. Upon leaving the module, the awakened caller checks if other processes have been suspended awaiting access to the device and activates the first process on the wait queue.

```
scheduled device LINE-OUT is
    LINE-LENGTH : constant INTEGER := 80;

    subtype CHAR-POSITION is INTEGER range 1 .. LINE-LENGTH;
    type    LINE is array ( 1 .. LINE-LENGTH ) of CHARACTER;
    type    RESULT is ( OK, ERR );

    procedure SEND (L : in LINE; R : out RESULT) ;

end;

scheduled device body LINE-OUT is

    subtype IO-RESULT is INTEGER range 1 .. 2;--1 for error,2 for ok

    LINE-STORE     : LINE;
    CURRENT-CHAR   : CHAR-POSITION := 1;
    DEVICE,STATUS  : IO-RESULT;

    scheduler LINE,SCHED is
        procedure ENTER;
        procedure AWAIT;
        procedure LEAVE;
        procedure LEAVE-INTERRUPT;
    end LINE,SCHED;


    procedure INITIALIZE-DEVICE (C: in CHARACTER; IR :out IO-RESULT ) is
        machine code . . .        -- machine code procedure to send a
                                  -- character to the device; see appendix F.
    end INITIALIZE-DEVICE;

    procedure TERMINATE,DEVICE is
        machine code . . . -- machine code to tell device to stop interrupting
    end TERMINATE,DEVICE;

    procedure SEND (L : in LINE; R : out RESULT) is
        scheduling (ENTER, LEAVE) ;
  , begin
        LINE-STORE := L;
        CURRENT-CHAR := 1;
```

```
    INITIALIZE-DEVICE (LINE_STORE (CURRENT-CHAR), DEVICE,STATUS) ;
    if DEVICE-STATUS = 1 then
        R := ERR;
    else
        LINE_SCHED.AWAIT;
        if DEVICE-STATUS = 1 then
            R := ERR;
        else
            R := OK;
        end if;
    end if;
end SEND;

interrupt OUT-CHAR called from 0016 is
    scheduling (null, LEAVE-INTERRUPT);
begin
    CURRENT-CHAR := CURRENT-CHAR + 1;
    INITIALIZE-DEVICE (LINE_STORE(CURRENT_CHAR), DEVICE-STATUS);
    if DEVICE-STATUS = 1 or CURRENT-CHAR = LINE-LENGTH then
        TERMINATE-DEVICE;
    end if;
end OUT-CHAR;

--the scheduler procedures insure mutual exclusion on the send procedure
--and provide synchronization between SEND and the interrupt

scheduler body LI NE,SCHED is
    imports (CURRENT-CHAR, DEVICE-STATUS : in );

    SCHED,LOCK  : LOCK;
    BUSY        : BOOLEAN := FALSE;
    WAIT_QUEUE  : CONDITION;
    USER        : PROCESSNAME;

    procedure ENTER is
    begin
        SET (SCHED,LOCK);
        if BUSY then
            INSERT (WAIT-QUEUE, MYNAME());
            RESET (SCHED_LOCK);
            SUSPEND;
        else
            BUSY := TRUE;
            RESET (SCHED,LOCK);
        end if;
    end ENTER;

    procedure AWAIT is
    begin
        SET (SCHED,LOCK) ;
        USER := MYNAMEO;
        SUSPEND;
        RESET (SCHED,LOCK) ;
    end AWAIT;
```

```
        procedure LEAVE is
            NEXT : PROCESSNAME;
        begin
            SET (SCHED,LOCK) ;
            if not EMPTY (WAIT_QUEUE) then
                REMOVE (WAIT-QUEUE, NEXT) ;
                ACTIVATE (NEXT) ;
            else
                BUSY := FALSE;
            end if;
            RESET (SCHED,LOCK) ;
        end LEAVE;

        procedure LEAVE-INTERRUPT is
        begin
            SET (SCHED,LOCK);
            if DEVICE-STATUS = 1 or CURRENT-CHAR = LINE-LENGTH then
                ACTIVATE (USER);
            end if;
            RESET (SCHED,LOCK);
        end LEAVE-INTERRUPT;

    begin
        RESET (SCHED,LOCK);
    end LINE,SCHED;

end LINE-OUT;
```

7. PROCESSES.

Processes are program units which may be initiated and **run in parallel. Processes** communicate by operating on scheduled modules. These modules **are called communication** channels and are declared in the specification part of process **declarations. Channels are** the only means of communication among processes. **Processes may not** import **objects and** objects **declared** within processes may not be imported by other units. **Processes may be** generic (but may have only **type** and **in** parameters). **Channel parameters (scheduled modules) may also** be **generic.**

The general format for a process declaration is:

```
[generic
    generic process parameter list]
process p [is
    channels channels list;
end [p] ];
```

A process body has the form:

```
process body p is
        declarative part
begin
        statement-list
end [p];
```

where:

(a) generic process parameter list has the form,
list of generic **type** and **in** parameters; **channels** generic channels-list;

(b) generic-channels_ list is a list of declarations of the form,-

m **is** n (L) **[restricted** (operations list) I

- where n is a generic scheduled module, L is a list **each member of which is in the** preceding list of generic **type** and in parameters. n(L) must **be an instance of n** obtained by replacing the generic formal parameters of n by generic formal type and in parameters of process p. Any actual module substituted **for m in an instance of p must** be an instance of n with the same parameters as those substituted **for corresponding members** of L. (See Examples 7-2, 7-3, and Section 7.4.)

(c) channels list has members of the form

m **[restricted** (operations list)]

- where m is an actual scheduled module.

(d) operations list - a list of visible operations of **a** module.

Notes:

1. Optional **clauses of the form, restricted (operations-list), in *a* channels declaration** restrict the operations on the channel which can be performed by the process.
2. Processes are an encapsulation unit; they **have some important differences from** modules:
   (a) channels declarations provide the only form of importation.
   (b) Processes cannot propagate exceptions.
   (c) generic parameters can only be type or in parameters.
3. A generic process declaration may have actual **channel parameters (perhaps in addition to generic channel parameters),**

7.1 Channels Declarations and Use Clauses.

The channels declaration may also include a use clause containing some of the names **of** the scheduled modules in the channels list. This avoids duplication **of channels and use declarations.**

Examples.

Example 7-7 **: Nongeneric Process.**

```
type BLOCK is . . .
type LINE is . . .
scheduled device LPT is
     procedure WRITE-LINE (L : LINE);

end LPT;' ..

scheduled module DISKFILE is
     procedure READ-BLOCK (B : out BLOCK);
          . . .
end DISKFILE;

process FI LE,PRI NT is
     channels use LPT, DISKFILE;
end FILE-PRINT;


process body FILE-PRINT is
     type LINE-STORE is array(1..C) of LINE;
     procedure BLOCK-TO-LINES (B : BLOCK; A : out LINE-STORE) is
          . . .                         -- transfers a block to a line store.
     end BLOCK_TO_LI NES;

     BUF :   LINE-STORE;
     BLOC : BLOCK;

begin
     READ-BLOCK (BLOC);                  -- read into BLOC from DISKFILE.
     BLOCK-TO-LINES (BLOC, BUF);         -- transfer BLOC to BUF.
     reserve LPT do                      -- tesetve LPT
          for i in 1 . . C loop
               WRITE-LINE (BUF(i));   -- write onto LPT
          end loop;
     end reserve;
end FILE-PRINT;
```

Example 7-2: Generic process with generic **channels.**

```
generic
     type T is private;
     SIZE : INTEGER;
scheduled module BUFFER is
     procedure READ (X: out T);
     procedure WRITE (Y: in T);
          . . .
end BUFFER; .

generic
     type ITEM is private;
```

```
      LENGTH : INTEGER;
      channels A is BUFFER (ITEM, LENGTH) restricted (READ),
                 B is BUFFER (ITEM, LENGTH) restricted (WRITE);
process TRANSFER;
                                    -- instances of TRANSFER must have channels
                                    -- that ate instances of BUFFER with the same
                                    -- pair of actual genetic parameters.
process body TRANSFER is
     C : ITEM;
begin
   loop
      A. READ (C);
      B. WRITE (C);
   end loop;
end TRANSFER;
```

7.2 Initiation of Processes.

Processes are initiated by the initiate statement:

initiate process-list;

where process-list is **a** list of previously declared actual processes.

A process may be initiated more than once; each time a new activation of execution of the process body occurs. Previously initiated instances of a process **ate not effected by later** initiations, except in **as** much **as** the instances share **channels.**

Initiate statements compile as **a** sequence of calls to the supervisor procedure, START (one call for each **process** in process list).

7.3 Termination of Processes.

The visibility and declaration rules of Adam establish **a** similar dependency **relation for processes as** exists for tasks in Ada **[6,** pg. **9-5].** Processes depend on subprograms, blocks, or processes within which they are initiated. Termination of a process occurs **when** the process execution reaches the end of its body and all dependent processes, **if any, have** terminated. Termination of a process compiles as a call to the supervisor **procedure** FINISH.

The dependency relationship **of** a process to a subprogram, block, or **process** imposes significant requirements on the techniques used to implement scope exit in **Adam. Each** unit which may have dependent processes must have an **associated** count or list of its nonterminated dependent **proceses,** in order to detect satisfaction of the exit condition. Only when the count reaches zero, or the list is empty, may the subprogram **return, the** block be left, or the **process** terminate. However, the scope exit problem in Adam is much less complicated and requires less **runtime** mechanism than is needed for scope exit in

Ada. Because Adam has no analogue of the terminate alternative of Ada and processes do not have visible operations, the only manner in which a process can terminate is by reaching the end of its body, either normally or by means of an exception. Once the end of its body has been reached, no further activity occurs in the process. The semantics of terminate in Ada requires that a dependent task inform the unit on which it depends when it selects a terminate alternative. This information could be exchanged either by decrementing a count or removing an object from a list as in Adam. However, in Ada the dependent task may have to change its terminate vote because of a call to an entry in the select statement containing the terminate. This communication of state information among scopes and their dependent tasks must be carefully implemented to insure absence of race and deadlock conditions and requires a considerably more sophisitcated task supervisor.

7.4 Instantiation of Processes.

instances of generic processes are created as follows:

process P **is new** Q (L; channels M);

where

L is a list of actual compiletime generic parameters,
M is a list of actual channels.

Rules for matching actual and formal generic parameters in instantiation of a generic process extend the Ada rules for generic instantiation [6,12.3]. Scheduled modules match generic formal channel parameters. The actual channel must be an instance of the formal channel obtained by replacing its generic formal parameters by actual generic parameters of the process instance. Thus, above, each member of M must be an instance of the corresponding generic channel with the members of L indicated in the declaration of Q.

Example 7-3. We continue with the previous example: correct and incorrect instantiations of the process, Transfer.

```
scheduled module BUF1 is new BUFFER (APPLES, 120);
scheduled module BUF2 is new BUFFER (APPLES, 120);
scheduled module BUF3 is new BUFFER (ORANGES, 120);

process T1 is new TRANSFER (APPLES, 120; channels BUF1,BUF2);
            -- This is a proper instance of TRANSFER,

process T2 is new TRANSFER (APPLES, 120; channels BUF1,BUF3);
            -- This is an improper instance of TRANSFER, since the declaration of
            -- TRANSFER requires that the genetic parameters in the declaration of
            -- BUF3 be the same as those in the declaration of T2, namely, APPLES, 720.
```

8, VISIBILITY RULES and IMPORTS,

The visibility rules of Adam are those of Ada with the following **restrictions placed on the** visibility of subprograms **and** objects within units. **Outer declarations of objects are not** visible within modules unless explicitly imported by an **imports declaration. Outer declarations of** objects **are not visible within processes, except those scheduled modules declared by a channels declaration. There are no restrictions on the visibility of objects** within subprograms. **Outer subprograms that are not exported from a module are not** visible in units.

These restrictions ensure **that all objects shared among processes can be enumerated** from channels and imports declarations. The correctness **of these declarations can be checked.**

8.1 Visible Declarations.

Declarations that **are always** visible within the body *of* **a** unit **if they are visible at the** point of declaration of that unit are the following:
(i) type declarations (including the constants of an enumeration type),
(ii) constants,
(iii) generic unit declarations,
(iv) exception declarations.
(v) processes,
(vi) predefined system modules (e.g., Supervisor).

8.2 Declarations Requiring Importation.

The following declarations must be imported in order to be visible within a module:
(i) variable declarations,
(ii) nongeneric module declarations (including instantiations of generic modules).

8.3 Imports.

Declarations requiring importation can be explicitly imported **into a nongenetic module** specification or body by **an imports** declaration of the **form:**

imports (imports list);

where

```
imports list ::= import-item {; import-item}
import-item ::= [use] identifier-list : import-kind
import-kind ::= in | out | in out | module
```

Imports are declared as part of the specification or body of a nongeneric module (see

formats, Section 6). Any external object that is mentioned **syntactically in** a **module must be** declared in its imports.

Imports declared in the specification of a module are **also** imported **into the body;** repetition of the imports declaration in the body is not required. **However, a module body may** have additional imports that are not declared in the module **specification.**

Note: Generic module declarations may not have imports.

8.3.1 Redundant Imports Declarations

Some parts of Ada and Adam declarations already perform **the** function **of declaring imports. In these cases a separate** imports declaration is unnecessary:

1. Objects listed in a with clause are imports.

2. When a generic module is instantiated, those generic parameters **that are objects are** imports of the **actual** instantiation.

8.3.2 Using Imports.

To avoid duplication **of Imports** and use lists, **a use declaration may appear within an imports list (see syntax above).**

Examples.

Example **8-1:** Imports **and** using imports.

```
module A is                               -- declaration of actual module A
    procedure  P ( . . . );
          . .
end A;'

module body B is
    imports (A: module) ;                 --A is now visible In body of B.

        procedure  C  ( . . . ) is
        begin
                A.P [illegible] .);
        end C;
end B;
```

```
module body D is
     imports (use A: module) ;          --A is both visible and used in body of D.

          procedure E ( . . . ) is
          begin
               . .
               P ( : , .);              -- call to A.P
               . .
          end E;'
end D;
```

Example 8-2: Visibility of generics and instances.

```
generic . . .
module STACK is
     procedure PUSH (. . .);
     . . .
end STACK;

module body CATALOGUE is
     . . .
     module GLOBAL_STACK is new STACK ( . . .);
                              -- generic STACK is visible in the body of CATALOGUE.
     . . .
     module SYMBOL_TABLE is
          imports (use GLOBAL-STACK: module) ;
                              -- nongenetic GLOBAL-STACK must be imported
          module LOCAL-STACK is new STACK ( . . .);
                              -- generic STACK is visible.
          . . .
          PUSH ( . . .);      - - GLOBAL_STACK.PUSH ( . . .).
          LOCAL_STACK.PUSH ( . . .);
          . . .
     end SYMBOL_TABLE;
     . . .
end CATALOGUE;
```

8.3.3 Encapsulated Imports.

Modules may be encapsulated by an enclosing module and exported by **the outer module.** In this **case, the declaration** within the body of the encapsulating module **may have** imports of local objects that **are** not visible in the exported specifications. These **are called** *encapsulated* imports. Modules **may** be used to encapsulate **and** hide imports from outside users. Thus careful modularization should result in only **essential objects appearing in** imports lists.

Example 8-3: Encapsulation of imports. The FILE TEMPLATE module body has two imports local to DISKFILES. These imports are implementation details **encapsulated in the body of** DISKFILES and not declared in the specification of FILE-TEMPLATE.

```
module DISKFILE is              -- DISKFILE exports FILE_TEMPLATE.

    scheduled module FILE-TEMPLATE Is
              . . .             -- no imports are declared In specification
    end FILE_TEMPLATE;          -- of FILE-TEMPLATE.
      . . .
end DISKFILE;

module body DISKFILE is         -- body of FILE-TEMPLATE is local to DISKFILE.

    module FILE_MANAGER is      -- FILE-MANAGER Is encapsulated by DISKFILE.
              . . .
    end FILE_MANAGER;

    device DISK is              -- DISK is encapsulated by DISKFILE.
              . . .
    end DISK;

    module body FILE,TEMPLATE is
        imports (use FILE_MANAGER, DISK : module) ;
          ..                    -- both imports are local to DISKFILE.
              . . .
    end FILE_TEMPLATE;

end DISKFILE;
```

9. CONCLUSIONS.

Although our use of Adam in pursuing the goals which motivated **development of the** language is not complete, our experience has led us to several conclusions which will **be** useful in our continued experimentation with Adam **and Ada. We hope these results will assist** others who **are** involved in similar efforts **and may be considering similar techniques.**

0.1 Writing Supervisors.

A supervisor for scheduling processes on **a** single POP-10 **processor has been written in** Adam, compiled, and is part of the runtime environment. It is used by Adam **programs with** processes and Ada programs with tasks. Two versions are currently in use; **one interfaces** with the SU-AI WAITS operating system and the other is for use with **TOPS-20. Appendix F** gives a simplified version of the WAITS supervisor.

Supervisors so far constructed fit very naturally into the structure of **a** scheduled **device** module encapsulating both interrupts and machine code. First, **it turns out that the standard** procedures, ACTIVATE (P : PROCESSNAME), . . . , **require only the simple "before** and **after"** protection provided by Adam scheduling declarations. Second, the encapsulation of protection in a separate scheduler subunit and the use of scheduling declarations improves the structure of the supervisor; the scheduling is much **easier to** understand than if each subprogram body contains protection, scheduling and computation all mixed together. This is true even in cases where the scheduler operations are trivial (e.g., simply disabling and enabling interrupts). For example, in Appendix F two of the standard supervisor procedures have null scheduling on exit. If these declarations were absent the reader might well consider the omission of enabling interrupts on exit from these procedures to be an error. Thirdly, it has been possible so far to encapsulate all machine language procedures (for switching contexts on the CPU and enabling **and** disabling interrupts) in a single subdevice such **as** CPU in Appendix F. (in fact, the two versions of the supervisor for WAITS and TOPS-20 differ only in the CPU subdevice.) In addition, process names have provided an adequate means of **refering to** threads of control in user-defined process scheduling and in interaction between such scheduling **and** the supervisor.

Further development is being undertaken to determine the extent to which the scheduled device structure suffices as a paradigm **for** more sophisticated supervisors and other varieties of resource scheduling systems. A first step in this work is to extend the current existing **supervisor** to provide facilities necessary for implementation of the Ada terminate alternative, the abort statement, and the FAILURE exception. We also intend to develop supervisors for more varied and complicated applications, including **large system** control, realtime interrupt driven processing, device handling, and multiple CPU **interfacing.** By considering a broad class of supervisor implementations, we **hope to demonstrate the** utiiity of Adam constructs or, perhaps, to discover generalizations of the Adam notions which will better meet the requirements for scheduling programs.

9.2 Transporting the Adam Compiler and Runtime Environment.

Our experience in transporting the Adam compiler to different operating systems **has** demonstrated the utility of providing a simple module interface between the compiler **and** the **runtime** system. The only changes to the compiler which were required in moving from **thg** WAITS to TOPS-20 systems involved changes in file naming **for i/O and interrupt** initialization code. Ail other operating system differences were absorbed by the supervisor written in Adam and, **as** mentioned above, the changes needed **in the supervisor were** confined to a single local device.

9.3 Translating Ada Multitasking.

in extending the Adam compiler to support Ada multitasking, we **have** used Adam **as an** intermediate target language. This use results essentially in **a** translation of Ada tasking into the lower level multiprocessing facilities of Adam. The development of this **translation** proceeded in two stages. First, for each Ada tasking construct we defined Adam text corresponding to the semantics of the construct and utilizing the scheduling **features of** Adam. Thus, the translation algorithm is defined by a mapping between **Ada source and**

Adam target programs; an example of this mapping is given in Appendix **G.** The second step in the translation was to augment the semantic processing phase of the compiler with MACLISP routines which replace abstract syntax tree nodes for Ada multitasking constructs with the corresponding Adam trees. Thus, though the algorithms for translating Ada multitasking were defined by correspondences between Ada and Adam program text, the implementation is by tree surgery on the parsed form of Ada programs. **The actual text** translation could be constructed by applying a pretty printer to the **Adam syntax tree after** completion of the compiler's semantic phase.

Our experience in using Adam as a target language for implementing **Ada multitasking has** led **us** to **several** conclusions, which are given below.

Q.3.1 Advantages of Using a High Level Target Language

The main advantages were reliability and ease of implementation; others include clarity **and** understandability of the translator.

We found that using Adam to specify the translation of Ada multitasking was an excellent technique for quickly producing **a** reliable implementation. Errors in the definition **of** Ada/Adam correspondence were easily identified and corrected by analyzing the proposed Adam text; we are certain that these errors would not have been so easily detected if it we had decided to hand generate and analyze sample assembly code. **In** fact, we were able to give quite short and convincing informal proofs that the Adam translations were semantically equivalent to the corresponding constructs of Ada. (Because adequate proof methods for the semantic correctness of parallel programs have not *yet* been developed, use of informal arguments is the best step towards verification of tasking **translators that** can be expected at the moment.)

Starting with the compiler for the parallel constructs of Adam, we implemented the tree transformations required for **a** substantial subset of Ada multitasking, including task types, rendezvous, select, conditional entry call, and a simplified abort statement (no abortion of dependent tasks) in **a** short time. Less than two man months were expended in writing **and** debugging the MACLISP code which was added to the compiler. Relatively few difficulties arose during the translation implementation and those that did were generally **resolved at** the level of the Adam text. The transforming operations basically emit a **subtree** of an Adam program, utilizing the context of the Ada abstract syntax tree. When **a** problem **was** encountered in the implementation, it was possible to identify the Adam source which was needed in the tree, to compile and dump the syntax tree for that Adam program, **and then** simply to write the (correct) constructor for that tree. We anticipate **that the advantages of** using Adam in our initial implementation of Ada tasking will **carry over to the alternative** implementations currently underway.

9.3.2 Using Ada as a Target for Translating Ada

One might ask whether the same advantages of using a high level **language for the target** of the multitasking translation would not have accrued if a pure subset of Ada **had been** used. For those parts of the translation specification which describe **only sequential** operations, Ada would certainly suffice, since sequential Adam is **sequential Ada. However,** one would still have to deal with describing concurrency by using **a restricted target** subset of **Ada. The target, of** course, could not utilize any form of rendezvous. One could. restrict the usage of tasks to only those with no' entries, eliminate select **and accept** statements, and have ail inter-task communication be carried out by operations **on shared** data structures (packages) for which one had explicitly written the exclusion **and** synchronization mechanisms. However, in such an Ada system, the problems **of interfacing** to a supervisor and of naming tasks would still be present. Furthermore, **the** discipline of scheduling the visible operations of the shared data structures would **have** to be carefully followed. in such a system in Ada, it would not be possible to determine from **their visible** parts which packages were shared by tasks (and **hence had scheduling) and which** packages were not. Clearly, solution of these problems was a primary factor in developing many of the constructs of Adam, and, hence, specifying the multitasking **translation is much more easy and clear in Adam.**

9.3.3 Multiprocessors and Optimization

We believe that use of an intermediate Adam transformation to implement Ada multitasking will expedite research into multiprocessor implementations and optimization **techniques.** The translation algorithms we currently use separate ail scheduling operations **and** rendezvous code from the thread of control of a task. By such a breakdown it will be easier to identify computations which may be truly carried out in **parallel and to isolate the** critical sections and synchronizations required for multiprocessor environments. Also, it is relatively easy to identify, as an optimization, those tasks which **may be compiled as static rather** than active structures.

9.3.4 Disadvantages of the Translation Technique

Thg use of Adam in developing the tasking translation algorithms **has had some drawbacks.** First, strong typing was occasionally very annoying and cumbersome. For **example, in a** message passing implementation of rendezvous, it is necessary **to declare complicated** variant record types in which the variant parts are lists **of the parameters of the task** entries. in a directly compiled impiementation the manipulation **of the parameters could be handled** without the type definitions. A second difficulty encountered **was due to some** mismatch between the Adam constructs and the desired translated semantics. **For** example, the task type construct for tasks with entries is not readily translatable into Adam. The actual impiementation of task types required modifying the code **generator of** our Adam compiler to permit creation of copies of scheduled module local **data** (almost like including a module type). Most of the mismatch problems occurred because much **of the** Adam design was based on Preliminary Ada and was maintained although **the tasking was** siginificantiy revised in final Ada.

9.3.5 Future Research

Our experience thus far has not been sufficient to allow us to draw conclusions about a number of questions associated with the multitasking translation. We have not been able to evaluate the compilation overhead and implemention efficiency of translating the internal form as compared to directly checking semantics and generating assembly code for the tasking constructs. Our experiments with modifying and replacing supervisors have been limited, and have not yet included supervisors and tasking for multiprocessor systems. The use of the multitasking translation for specification and verification of Ada task systems, including proof of equivalence of an Ada program and its corresponding Adam program, has not progressed much beyond the concept stage. These questions, and many other related issues, are the objects of ongoing research.

Acknowledgment:

The authors wish to thank Anthony Garagaro for many helpful comments on an earlier draft of this paper.

REFERENCES.

[1] Brinch Hansen, Per, "Architecture of Concurrent Progams," Prentice Hail.
[2] Brinch Hansen, Per, "The Design of Edison", University of Southern California, Comp. Science Dept. report, Sept. 1980.
[3] Good,D.I., et al, Gypsy Manual, University of Texas Report.
[4] Hoare, C.A.R., @Communicating Sequential Processes'@ Comm. ACM 21, 1978, pp. 666-677.
[5] Holden, J., and Wand, I.C., "An Assessment of Modula", Software Practice and Experience, Vol. 10, pp. 593-621, 1980.
[6] Ichbiah, J., et al., "Reference Manual for the ADA language", proposed standard document, U.S. Dept. of Defense, July 1980. see also: "Preliminary Ada Reference Manual and Rationale," ACM Sigplan Notices vol. 14.6, June 1979.
[7] Karp, R. A., "Proving concurrent systems correct", Computer Science Department Report No. STAN-(X-79-783, Stanford University, Nov. 1979.
[8] Krieg Bruckner, B., and Luckham, DC., "ANNA : Towards a language for annotating Ada programs", Proceedings of the ACM Sigplan Symposium on the Ada Programming Language, ACM Sigpian Notices, vol. 15, No. 11, November 1980.
[9] Lohr, K-P., "Beyond Concurrent Pascal", Proc. Sixth ACM Symposium on Operating System Principles, Nov. 1977, pp.1 73-180.
[10] Luckham, D.C., and Polak, W., "Ada Exceptions: Specification and proof techniques," Stanford University Computer Science Dept. Program Verification Group Report 16; also TOPLAS April 1980.
[11] Luckham, DC., and Poiak, W., "A Practical Method of Documenting Ada Programs with Packages", Proceedings of the ACM Sigpian Symposium on the Ada Programming Language, ACM Sigpian Notices, vol. 16, No. 11, November 1980.

[12] Stevenson, D.R., "Algorithms for Translating **Ada Multitasking", Proceedings of the** ACM Sigplan Symposium on the Ada Programming **Language, ACM Sigpian Notices, vol. 15,** No. 11, November **1980.**
[13] Wirth, N., **"Modula - 2",** ETH report, Zurich, March 1980.

APPENDIX A : SYNTAX.

Notation:

Reserved words are in boldface. Square brackets, **[ ],** indicate **an optional construct.** Curly brackets, (**},** indicate zero or more repetitions of a construct. To indicate one or more repetitions we use **{ }+.** The bar, **|,** is used to indicate alternation in the **right-hand-**side of a production. **Any** nonterminal of the form xxx-name is **syntactically the same as** name.

compilation ::= (compilation-unit)

compilation-unit ::=
    context-specification module-specification;
  | context-specification generic-module-declaration
  | context-specification module-body
  | context-specification subprogram-body

context-specification ::= (with-clause [use-clause])

with-clause ::= **with** unit-name {, unit-name);

DECLARATIONS.

declarative-part ::=
    (declarative-item) {representation-specification) (program-component}

declarative-item ::= declaration | use-clause

program-component ::= body | module-declaration

body ::= subprogram-body | module-body | process-body

declaration ::=
    object-declaration | type-declaration
  | subtype-declaration | subprogram-declaration
  | module-declaration | process-declaration
  | exception-declaration | renaming-declaration

use-clause ::= **use** module-name {, module-name);

object-declaration ::=
    identifier-list : [constant] subtype-indication [:= expression];
  | identifier-list : [constant] array-type-definition [:= expression3

ident if ier-list ::= identifier {, identifier)

exception-declaration ::= identifier-list : **exception**

renaming-declaration ::=
identifier : type-mark **renames** name;
| identifier : **exception renames** name;
| module-nature identifier **renames** name;

TYPES.

type-declaration ::=
type identifier **is** type-definition;
| incomplete-type-declaration;

type-definition ::=
enumeration-type-definition | array-type-definition
| record-type-definition | access-type-definition
| private-type-definition

subtype-declaration ::=
subtype identifier **is** subtype-indication;

subtype-indication ::= type-mark [constraint]

type-mark ::= type-name | subtype-name

constraint ::= range-constraint | index-constraint

range-constraint ::= **range** range
range ::= simple-expression .. simple-expression

index-constraint ::=(discrete-range {, discrete-range))
discrete-range ::= [type-mark **range**] range

enumeration-type-definition ::= (enumeration-literal {, enumeration-literal))

enumeration-literal ::= identifier | character-literal

chararacter-literal ::=' character '

record-type-definition ::=
record
component-list
end record

component -list ::=
  (component-declaration}+

component-declaration ::=
  identifier-list : subtype-indication [:= expression];
  | identifier-list : array-type-definition [:= expression)

array-type-definition ::=
  array (index {, index)) of subtype-indication
  | array index-constraint of subtype-indication

index ::= type-mark range <>

access-type-definition ::= access subtype-indication

incomplete-type-declaration ::= type identifier ;

private-type-definition ::= [limited] private

representation-specification ::= address-specification

address-specification ::=
  for identifier use at expression;

EXPRESSIONS.

expression-list ::= expression {, expression)

expression ::= relation (logical-operator relation)

relation ::= simple-expression (relational-operator simple-expression)

simple-expression ::= [unary-operator] term (adding-operator term)

term ::= factor (multiplying-operator factor)

factor ::= primary { primary)**

primary ::= literal | aggregate | name | function-call | (expression) | allocator

logical-operator ::= and | or | xor | and then | or else

relational-operator ::= = | /= | < | <= | > | >=

adding-operator ::= + | -

multiplying-operator ::= * | / | mod

unary-operator ::= + | - | not

literal ::= number | enumeration-literal | character-string | null

aggregate ::=
[type-mark'] (component-association {, component-association))

component-association ::= [choice {| choice} =>] expression

choice ::= simple-expression | range | others

allocator ::= new type-mark [(expression)]| new type-mark aggregate

function-call ::= function-name (expression-list)| function-name ()

name ::= identifier | indexed-component | selected-component | attribute

indexed-component ::= name (expression-list)

selected-component ::= name.identifier | name.all

attribute ::= name'identifier

identifier ::= letter { letter | digit | underscore }

number ::= digit (digit)

character-string ::= "(character)"

STATEMENTS.

statement-list ::= statement (statement)

statement ::= (label) simple-statement | (label) compound-statement

simple-statement ::=
assignment-statement | exit-statement
| return-statement | goto-statement
| raise-statement | procedure-call

```
   |   code-statement
   |   initiate-statement         | null;

compound-statement ::=
         if-statement             | case-statement
   | loop-statement               | block
   |   reserve-statement
label ::= <<identifier>>


assignment-statement   ::= name := expression;

exit-statement ::=  exit [identifier]  [when  expression];

return-statement::=   return [expression];

goto-statement   ::=  goto identifier;

raise-statement   ::=  raise [exception-name];

procedure-call ::=   procedure-name [( expression-list )];

code-statement ::= aggregate

initiate-statement ::=   initiate process-name {, process-name};

if-statement ::=
        if expression then
              statement-list
        {elsif expression then
              statement-list }
        [else
              statement-list ]
        end if;


loop-statement   ::= [identifier:] [iteration-clause] basic-loop

iteration-clause ::=
        while expression
   |    for identifier in [reverse] range

basic-loop ::=
        loop
              statement-list
        end loop;

case-statement ::=
        case expression is
           { when  choice {| choice) => statement-list }+
        end case;


block ::=
```

```
        [identifier:]
        [declare
              declarative-part]
        begin
              statement-list
        [exception
              (exception-handler) ]
        end;

exception-handler ::=
        when exception-choice {| exception-choice) =>
              statement-list

exception-choice ::= exception-name | others

reserve-statement ::=
        reserve module-name do
              statement-list
        end reserve;
```

SUBPROGRAMS.

```
subprogram-declaration ::=
        subprogram-header;
   | generic-subprogram-declaration
   | generic-subprogram-instantiation

subprogram-body ::=
        subprogram-header is
              specification-part
              declarative-part
        begin
              statement-list
        [exception
              (exception-handler)]
      end [designator];

subprogram-header ::=
        function designator [formal-part] return subtype-indication
   | procedure identifier [formal-part]
   | interrupt identifier called from number

designator ::= identifier | character-string
```

```
formal-part ::=
      (parameter-declaration {; parameter-declaration} )

parameter-declaration ::=
      identifier-list : mode subtype-indication

mode ::= [in] | out | in out


specification-part ::=
      [propagate-declaration]
      [scheduling (scheduling-item, scheduling-item);]


imports-list ::= imports (import-item {; import-item});

import-item ::= [use] identifier-list : import-kind

import-kind ::= in | out | in out | module


propagate-declaration ::= propagate identifier-list;

scheduling-item ::= identifier [(expression-list)] | null


generic-subprogram-declaration ::= generic-part subprogram-header;
generic-part ::= generic (generic-formal-parameter)

generic-formal-parameter ::=
      parameter-declaration;
   |  with subprogram-header;
   |  type identifier is private-type-definition;


generic-subprogram-instantiation ::=
      procedure identifier is generic-instantiation;
   |  function designator is generic-instantiation;

generic-instantiation ::= new identifier [( expression-list )]
```

MODULES.

```
module-declaration ::=
      module-specification;
   | generic-module-declaration
   | generic-module-instantiation
```

```
module-specification ::=
        module-nature identifier is
                [import-list]
                (declarative-item)
        [private
                (declarative-item}
                {representation-specification}]
        end [identifier]


generic-module-declaration ::= generic-part module-specification;

generic-module-instantiation ::=
        module-nature identifier is generic-instantiation;


module-body ::=
        module-nature body identifier is
                [import-list]
                declarative-part
        [begin
                statement-list
        [exception
                (exception-handler)]]
        end [identifier];

module-nature ::=
        scheduler | device | module | scheduled device
    | scheduled module
```

PROCESSES.

```
process-declaration ::=
        process-specification
  | generic-process-declaration
 | generic-process-instantiation;


process-specification ::=
        process identifier [is
                [channels actual-channel {, actual-channel};]
        end];

generic-process-declaration ::= process-generic-part process-specification

generic-process-instantiation ::=
        process identifier is new identifier [( instantiation-actuals )];


process-body ::=
        process body identifier is
```

```
        declarative-part
    begin
        statement-list
    [exception
        {exception-handler}]
    end [identifier];


actual-channel ::= identifier [restricted ( identifier-list )]


process-generic-part ::=
    generic [ {process-generic-parameter}
        [channels generic-channel {, generic-channel};] ]

process-generic-parameter ::=
    identifier-list : [in] type-mark;
  | type identifier is private-type-definition;

generic-channel ::=
    identifier is module-name [(expression-list)]
        [restricted ( identifier-list )]


instantiation-actuals ::=
    expression-list
  | channels-list
  | expression-list; channels-list

channels-list ::= channels identifier-list
```

APPENDIX B: RESERVED WORDS.

and	for	new	reserve
array	from	not	restricted
at	function	null	return
			reverse
begin	generic	of	
body	got0	or	scheduled
		others	scheduler
called	if	out	scheduling
case	imports		subtype
channels	in	pragma	
constant	initiate	private	then
	interrupt	procedure	to
declare	is	process	type
device		program	
do	limit	propagate	use
	loop		
else		raise	when
eisif	mod	range	while
end	module	record	with
exit		renames	
exception		requires	xor

APPENDIX C: LIST OF PRAGMAS,

pragma INCLUDE (file name)
pragma PRIORITY (number between 0 and 10)
pragma SUPERVISOR (module name) - explained in Section 2.8
pragma MAIN - specify **main program**

APPENDIX D: PREDEFINED ATTRIBUTES.

Attributes of any object:

ADDRESS	X 'ADDRESS returns an integer corresponding to the location of the first storage cell of X.

Attributes of any type or subtype:

SIZE	T'SIZE gives the number of storage units used to represent the type.

Attributes of any scalar type or subtype:

FIRST	T'FIRST returns the minimum value in the range of T.
LAST	T'LAST returns the maximum value in the range of T.

Attributes of any discrete type or subtype T:

POS(X)	Returns an integer which is the ordinal position of X in the type T.
VAL(I)	Returns the enumeration value occupying the Ith position in T.

Attributes of any array object or array type with specified bounds:

FIRST --	Returns the value in the first index which is the lower bound of that index.
FIRST(i)	Same as FIRST for the ith index.
LAST	Upper bound for the first index.
LAST(i)	Same as LAST for the ith index.
LENGTH	Number of elements in the first dimension.
LENGTH(i)	Same as LENGTH for the ith dimension.

APPENDIX Et PREDEFINE0 EXCEPTIONS,

Exception Name	When Raised
CONSTRAINT-ERROR	When exceeding the declared range of a variable, or when an index value is outside the range specified for an array, or when dereferencing an access variable that has the value null
STORAGE-ERROR	When all free storage in the heap is used up.
CONDITION-QUEUE-FULL	When attempting to insert into a full condition queue
CONDITION_QUEUE_EMPTY	When attempting to remove from an empty condition queue

APPENDIX F: A STANDARD SUPERVISOR.

This appendix presents a simplified version of the process supervisor that we have been using in our implementation of Adam on SAIL WAITS. Processes are given a fixed size time slice and are preempted if they exceed their time slice. Scheduling within a priority level is round-robin. The WAITS operating system provides timer interrupts to implement the timing. The priority of a process may be specified by a pragma.

```
scheduled device SUPERVISOR is
    type INIT_DATA is limited private;

    subtype TICK-COUNT is INTEGER range 0 . . INTEGER'LAST;
    subtype PRIORITY      is INTEGER range 0 . . 10;
    subtype ADDRESS       is INTEGER range 0 . . 2 ** 18 - 1;
    procedure SUSPEND;
    procedure ACTIVATE (P : PROCESSNAME);
    procedure SWITCH (P : PROCESSNAME);
    procedure START (D : INIT_DATA);
    procedure FINISH;
    procedure DELAY-FOR (I : TICK-COUNT);

private
    type INIT_DATA is
        record
            PNAME       : PROCESSNAME;
            CODESTART : ADDRESS:
            STKSTART   : ADDRESS;
            PRIORITY    : PRIORITY;
        end record;

    interrupt TIMER-INTERRUPT called from 0;

end SUPERVISOR;


with DEC10_INSTRUCTIONS;
scheduled device body SUPERVISOR is

-- DEC10_INSTRUCTIONS is a package which defines the formats for insetting
-- machine code.

    MAX-PROCESSES :  constant INTEGER := 40;
    subtype PT,INDEX is INTEGER range 0 , , MAX-PROCESSES;

    NO-PROCESS :  constant PT,INDEX := 0;

    type PROCESS-STATUS is (RUN, READY, BLOCKED);

    type QHEADER is
        record
            FIRST :  PT,INDEX;
            LAST   : PT,INDEX;
        end record;
```

```
    type READYQS is
        array (PRIORITY) of QHEADER;

    type REGISTER-SET is
        array (0 . . 15) of INTEGER;

    type PROCESS-DATA is
        record
            NAME            : PROCESSNAME;
            STATUS          : PROCESS-STATUS;
            PC              : ADDRESS;
            REG             : REGISTER-SET;
            PRIORITY        : PRIORITY;
            DELAY-TIME      : TICK-COUNT;
            NEXT, PRIOR : PT_INDEX;
        end record;

    PT : array (PT,INDEX range 1 . . MAX-PROCESSES) of PROCESS-DATA;

    MAIN-PROGRAM : constant PT,INDEX : = 1;
    RUNNING : PT,INDEX : = MAIN_PROGRAM;- table index of the currently running
                                          -- process
    FREE      : PT,INDEX : = 2;

    READYQ : READYQS;
    DELAYQ : QHEADER;

    BLOCKED-COUNT : INTEGER : = 0;        -- count of number of blocked processes

    TICK-LENGTH : constant INTEGER : = 6;        -- = 6/60 of a second
    TIME-SLICE   : constant TICK-COUNT : = 10;-- = 70 * TICK_LENGTH = 7 second

    RUNNERS-TICKS : TICK-COUNT := 0;
    SP   : constant INTEGER : = 14;        -- register which points to stack frame
    TOP : constant INTEGER : = 15;

    MAIN-PRIORITY : PRIORITY;             -- priority of the main program

    for MAIN-PRIORITY use at "PRITY↑"; --this teptesentatlon specification
                                       --is known at link time

scheduler S is
    procedure ENTER;
    procedure LEAVE;
    pragma I NLI NE (ENTER, LEAVE) ;
end S;


module Q is

    procedure INSERT (Q : in out QHEADER; i : PT,INDEX);
    procedure REMOVE (Q : in out QHEADER; i : in out PT,INDEX);
```

```
        procedure DELETE (Q : in out QHEADER; i : PT,INDEX);
        function EMPTY (Q : QHEADER) return BOOLEAN;
        pragma I NLI NE (EMPTY) ;

        procedure INSERT (A : in out READYQS; i : PT,INDEX);
        procedure REMOVE (A : in out READYQS;    i : in out PT,INDEX);
        procedure DELETE (A : in out READYQS; i : PT,INDEX);
        function EMPTY (A : READYQS) return BOOLEAN;
        function FIRST (A : READYQS) return PT,INDEX;
        pragma INLINE (EMPTY, DELETE, INSERT);
    end Q;


    device CPU is
      -- machine dependent code
        procedure IDLE;
        procedure START-PROCESS (D : PROCESS-DATA) ;
        procedure SAVE-CONTEXT (D : in out PROCESS-DATA);
        procedure DISABLE;
        procedure ENABLE;
        pragma I NLI NE (ENABLE, DI SABLE) ;

        procedure STARTUP        (D : PROCESS-DATA);
        procedure SAVE-STATE (D : in out PROCESS-DATA);
    end CPU;


    function NAME-TO-INDEX (P : PROCESSNAME) return PT,INDEX is
        -- convert a processname into a table index
    begin
        if P /= null then
            for i in PT_INDEX'FIRST + 1 . . PT_INDEX'LAST loop
                if PT(i). NAME = P then
                    return i;
                end if;
            end loop;
        end if;
        return NO-PROCESS;
    end NAME-TO-INDEX;


    procedure UNBLOCK (i : PT,INDEX) is
        PCB : PROCESS-DATA renames PT(i);
    begin
        if PCB. DELAY_TIME > 0 then
            Q. DELETE (DELAYQ, i);
            PCB. DELAY_TIME := 0 ;
        end if;
    end;


    procedure DO-SUSPEND is
    begin
```

```
        CPU. SAVE-CONTEXT (PT(RUNNING));
        PT(RUNNING).STATUS := BLOCKED;
        BLOCKED-COUNT := BLOCKED-COUNT + 1;

        if Q.EMPTY (READYQ) then
            CPU.IDLE;
        else
            Q.REMOVE (READYQ, RUNNING);
            PT(RUNNING).STATUS := RUN;
            CPU.START_PROCESS (PT(RUNNING));
        end if;
    end DO-SUSPEND;


    procedure START (D : INIT_DATA) is
        scheduling (ENTER, LEAVE);
        i : PT,INDEX;
    begin
        if FREE = NO-PROCESS then
            raise STORAGE-ERROR;
        else
            i := FREE;    FREE := PT(FREE).NEXT;

            PT(i) := (NAME          => D.PNAME,
                      STATUS        => READY,
                      PC            => D.CODESTART,
                      REG           => (SP => D.STKSTART, others => 0),
                      PRIORITY      => D.PRIORITY,
                      DELAY-TIME    => 0,
                      NEXT I PRIOR  => NO-PROCESS);

            Q.INSERT (READYQ, i);
        end if;
    end START;


    procedure FINISH is
        scheduling (ENTER, null);
    begin
        PT(RUNNING).NEXT := FREE;     FREE := RUNNING;
        PT(RUNNING).NAME := null;
        if Q.EMPTY (READYQ) then
            CPUJDLE;
        else
            Q.REMOVE (READYQ, RUNNING);
            PT(RUNNING).STATUS := RUN;
            CPU.START_PROCESS (PT(RUNNING));
        end if;
    end FINISH;


    procedure ACTIVATE (P : PROCESSNAME) is
        scheduling (ENTER, LEAVE);
```

```
        i : constant PT_INDEX := NAME-TO-INDEX (P);
    begin
        if i /= NO-PROCESS and then PT(i).STATUS = BLOCKED then
            UNBLOCK (i);
            BLOCKED-COUNT := BLOCKED-COUNT + 1;
            Q. INSERT (READYQ, i);
            PT(i).STATUS := READY;
        end if;
    end ACTIVATE;


    procedure SUSPEND Is
        scheduling (ENTER, null);
    begin
        DO-SUSPEND;
    end SUSPEND;


    procedure SWITCH (P : PROCESSNAME) is
        scheduling (ENTER, LEAVE);
        i : constant PT,INDEX := NAME-TO-INDEX (P);

        procedure DO-SAVE (D: in out PROCESS-DATA) is
        begin
            CPU.SAVE,CONTEXT (D);
        end;
    begin
        If i /= NO-PROCESS and then PT(i).STATUS /= RUN then
            DO-SAVE (PT(RUNNING));
            PT (RUNNING). STATUS := BLOCKED;
            RUNNING := i;
            if PT(i).STATUS = READY then
                Q.DELETE (READYQ, i);
                BLOCKED-COUNT := BLOCKED-COUNT + 1;
            else
                UNBLOCK (i);
            end if;
            PT(i). STATUS := RUN;
            CPU.START,PROCESS (PT(i));
        end if;
    end SWITCH;


    procedure DELAY-FOR (I : TICK-COUNT) is
        scheduling (ENTER, LEAVE);
    begin
        if I > 0 then
            Q.INSERT (DELAYQ, RUNNING);
            PT(RUNNING).DELAY_TIME := I;
            DO-SUSPEND;
        end if;
    end;
```

```
module body Q is

    function EMPTY (Q : QHEADER) return Boolean is
    begin
        return Q. FIRST = KO,PROCESS;
    end;

    procedure INSERT (Q : in out QHEADER; i : PT,INDEX) is
    begin
        PT (i). PRIOR : = NO-PROCESS;
        if Q.FIRST = NO-PROCESS then
            Q. FIRST : = i;  PT(i). NEXT : = NO-PROCESS;
        else
            PT (Q. LAST). PRIOR : = i;   PT (i). NEXT := Q. LAST;
        end if;
        Q. LAST : = i;
    end;

    procedure DELETE (Q : in out QHEADER; i : PT_INDEX) is
        PCB : PROCESS-DATA renames PT(i);
    begin
        if i = Q.FIRST then
            Q.FIRST : = PCB.PRIOR;
            if Q.FIRST /= NO-PROCESS then
                PT(Q. FIRST).NEXT: = NO-PROCESS;
            end if;
        elsif i = Q.LAST then
            Q.LAST : = PCB.NEXT;
            if Q.LAST /= KO,PROCESS then
                PT(Q. LAST).PRIOR: = NO-PROCESS;
            end if;
        else
            PT (PCB. PRIOR). NEXT : = PCB, NEXT;
            PT (PCB. NEXT). PRIOR : = PCB, PRIOR;
        end if;
    end;

    procedure REMOVE (Q : in out QHEADER;  i : in out PT_INDEX) is
    begin
        i : = Q. FIRST;
        Q.FIRST : = PT (i). PRIOR;
        if Q. FIRST /= NO-PROCESS then
            PT(Q. FIRST).NEXT: = NO-PROCESS;
        end if;
    end;

    procedure DELETE (A : in out READYQS; i : PT,INDEX) is
    begin
        Q. DELETE (READYQ (PT (i). PRIORITY), i);
    end;
```

```
    procedure INSERT (A : in out READYQS; i : PT,INDEX) is
    begin
        Q. INSERT (READYQ(PT(i).PRIORITY), i);
    end;


    function EMPTY (A : READYQS) return BOOLEAN is
    begin
        return Q.FIRST (A) = NO-PROCESS;
    end;


    procedure REMOVE (A : in out READYQS; i: in out PT,INDEX) is
    begin
        for j in reverse PRIORITY'FIRST . . PRIORITY'LAST loop
            if not Q.EMPTY (READYQ(j)) then
                Q.REMOVE (READYQ(j), i);
                exit;
            end if;
        end loop;
    end;


    function FIRST (A : READYQS) return PT,INDEX is
    begin
        for j in reverse PRIORITY'FIRST . . PRIORITY'LAST loop
            if not Q.EMPTY (READYQ(j)) then
                return READYQ(j).FIRST;
            end if;
        end loop;
        return NO-PROCESS;
    end;
end Q;


device body CPU is
    use DEC10_INSTRUCTIONS;
    use REG I STERS;


    procedure DISABLE is
    begin
        UUO'(INTMSK, (LITERAL, 0));
    end;


    procedure ENABLE is
    begin
        UUO'(INTMSK, (LITERAL, -1));
    end;
```

```
procedure STOP is
begin
    FAIL_INSERTION'(TEXT => "EXIT");
end;

procedure SLEEP is
    -- this procedure is called when there ate only blocked processes
begin
    loop
        ENABLE;
        R1 := 1;
        UUO' (SLEEP, 1, (QUOTED, ""));          -- sleep for 1 second
        DISABLE;
        if not Q.EMPTY(READYQ) then
            Q.REMOVE (READYQ, RUNNING);
            PT(RUNNING).STATUS := RUN;
            CPU.START,PROCESS (PT(RUNNING));
        end if;
    end loop;
end;

procedure IDLE is
    -- this procedure is called when there ate no ready processes to run
begin
    RUNNING := NO-PROCESS;

    if BLOCKED-COUNT = 0 then
        STOP;
    else
        SLEEP;
    end if;
end;

procedure SAVE-STATE (D : in out PROCESS-DATA) is
    -- this procedure is called when a process Is preempted during a timer
    -- interrupt
    SAIL,REG,SAVE : REGISTER-SET;
    SAIL-PC-SAVE : INTEGER;
    for SAIL,REG,SAVE use at 16;
    for SAIL-PC-SAVE use at 87;
begin
    D.REG := SAIL_REG_SAVE;
    DEC10'(HRRZ, 1, (ADDRESS, SAIL-PC-SAVE));
    D.PC := R1: ;
end;

procedure STARTUP (D : PROCESS-DATA) is
begin
    CPU.DISABLE;
    UUO' (OP => DEBREAK);
    CPU.START_PROCESS (D);
end;
```

```
procedure START-PROCESS (D : PROCESS-DATA) is
begin
    RUNNERS-TICKS := 0;
    RO := D.PC;
    DEClO' (MOVEM, 0, (LABEL, "XFER", 1));
    DEC10'(MOVSI, 0, (ADDRESS, INTEGER'(D.REG)));
    DEClO' (BLT, 0, (REG,15));
    UUO' (INTDEJ, (LABEL, "XFER",0));
    FAIL_LABEL'("XFER", (VALUE, -1));
    DIRECTIVE'(BLOCK, (VALUE, 1));
end;


procedure SAVE-CONTEXT (D : in out PROCESS-DATA) is
    -- this procedure must be called 3 dynamic links away from the
    -- stack frame of the caller of the kernel
begin
    DEC 10 '(HLRZ, 1, (I NDEX, 0, 14) ); -- get caller's saved return address
    DEClO' (HLRZ, 1, (INDEX, 0, 1)); -- by following dynamic links
    DEClO' (HRRZ, 0, (INDEX, 0, 1));
    D.PC := RO;
    DEClO' (HLRZ, 0, (INDEX, 0, 1)); -- get caller's dynamic link
    D.REG(SP):= RO;
    D. REG (TOP) := R1-1;            -- save caller's top of stack pointer
end;
end CPU;


procedure DO,WAKEUPS is
    i : PT_I NDEX := DELAYQ, FIRST;    j : PT_INDEX;
begin
    while i /= NO-PROCESS loop
        declare PCB : PROCESS-DATA renames PT (i) ; begin

            PCB.DELAY,TIME := PCB.DELAY,TIME - 1;
            if PCB. DELAY-TIME <= 0 then
                j := i; i := PCB.PRIOR;
                Q.DELETE (DELAYQ, j);
                Q.INSERT (READYQ, j);
                BLOCKED-COUNT := BLOCKED-TIME - 1;
            else
                i := PCB. PRIOR;
            end if;
        end;
    end loop;
end;


interrupt TIMER-INTERRUPT called from 0 is
    i : PT_I NDEX;
begin
    DO,WAKEUPS;

    if RUNNING /= NO-PROCESS then
```

```
            RUNNERS-TICKS := RUNNERS-TICKS + 1;
            if not Q.EMPTY(READYQ) and then
                 (RUNNERS-TICKS > TIME-SLICE or
                 PT(Q.FIRST(READYQ)).PRIORITY > PT(RUNNING) ● PRIORITY) then
                 CPU.SAVE,STATE (PT(RUNNING));
                 PT(RUNNING).STATUS := READY;
                 Q.REMOVE(READYQ, i);
                 Q.INSERT(READYQ, RUNNING);
                 RUNNING := i;
                 PT(i).STATUS := RUN;
                 CPU.STARTUP (PT(i));
            end if;
       end if;
end TIMER-INTERRUPT;


scheduler body S is

     procedure ENTER is
     begin
          CPU.DISABLE;
     end;

     procedure LEAVE is
     begin
          CPU.ENABLE;
     end;
end S;


begin
     PT(MAIN_PROGRAM) := (NAME          => PROCESSNAME'(1),
                          STATUS        => RUN,
                          PC            => 0,
                          REG           => (others => 0),
                          PRIORITY      => MAIN-PRIORITY,
                          DELAY-TIME    => 0,
                          NEXT I PRIOR  => NO-PROCESS);

     for i in PRIORITY loop
          READYQ(i).FIRST := NO-PROCESS;
     end loop;

     DELAYQ.FIRST := NO-PROCESS;

     for i in PT,INDEX'FIRST + 2 . . PT,INDEX'LAST - 1 loop
          PT(i).NEXT := i + 1;
          PT(i).NAME := null;
     end loop;
     PT(PT_INDEX'LAST).NEXT := NO-PROCESS;
     PT(PT_INDEX'LAST).NAME := null;
     CPU.ENABLE;
     UUO'(CLKINT, (VALUE, TICK-LENGTH));            -- enable timer Interrupts
end SUPERVISOR;
```

APPENDIX G: ADA MULTITASKING TRANSLATION EXAMPLE.

This appendix presents an example of techniques used in translating the multitasking constructs of Ada into Adam. Various algorithms for such translation are being developed and are described in detail in [9].

Tasking in Ada provides a very general, expressive, and elegant means of designing parallel systems. However, because of their generality, the high level tasking constructs of Ada pose a significant challenge for language implementers. Much concern has been expressed about the efficiency and even possibility of implementing the full multitasking capabilities of Ada. The multiprocessing constructs of Adam, on the other hand, are much lower level than those of Ada and create no major compilation difficulty. Hence, by developing implementations of Ada tasking in Adam the problem a Ada multitasking may be readily identified and studied. Automation of the algorithms will permit testing and comparing performance of implementations which use different execution, scheduling, and resource allocation schemes. Also, the algorithms may be used with the existing Adam compiler to produce a two-step compiler for Ada tasking.

The essential step of the translation algorithms is to transform the components of an Ada multitasking system into corresponding elements of an Adam system. Any Ada task which does not have visible entries is transformed into an Adam process. Ada tasks with entries, which we term "service tasks", are translated into both a process and a scheduled module in Adam. This division of the service task into two parts separates the truly independent thread of control of the task from the synchronization and inter-task communication functions of the task.

The example below presents the general form of translation for a very simple Ada task system, a buffer and two user tasks. In Ada, such a system might appear as follows:

```
task CHARACTER-BUFFER Is
     entry PUT-CHAR (C : In CHARACTER);
     entry GET-CHAR (C : out CHARACTER);
end  CHARACTER-BUFFER;

task body CHARACTER-BUFFER is
     MAX        : constant INTEGER := 200;

     subtype BUFFER-POINTER is INTEGER range 0 . . MAX;

     BUFFER   : array (1 . .  MAX) of CHARACTER;
     IN,PTR   : BUFFER-POINTER := 1;
     OUT,PTR :  BUFFER-POINTER := 0;
begin
     loop
          select
               when IN,PTR /= OUT,PTR =>
                    accept PUT-CHAR (C : In CHARACTER) do
                         BUFFER(IN_PTR) := C ;
                    end PUT-CHAR;
                    IN,PTR :=  IN,PTR mod MAX + 1;
```

```
                    if OUT,PTR = 0 then
                        OUT,PTR := 1;
                    end if;
            or
                when OUT,PTR =/ 0 =>
                    accept GET-CHAR (C : out CHARACTER) do
                        C := BUFFER (OUT,PTR) ;
                    end GET-CHAR;
                    OUT,PTR := OUT,PTR mod MAX + 1;
                    if OUT,PTR = IN,PTR then
                        OUT,PTR := 0; IN,PTR := 1;
                    end if;
            end select;
        end loop;
    end CHARACTER-BUFFER;

    task PRODUCER;                          -- the body of PRODUCER contains calls to
                                            -- CHARACTER-BUFFER. PUT

    task CONSUMER;                          -- the body of CONSUMER contains calls to
                                            -- CHARACTER-BUFFER. GET
```

One algorithm used for translation of Ada tasking uses procedure **call to implement the user** task/service task rendezvous. In this scheme, the calling task **executes the body of the accept** and awakens the service task **at** completion of **the** rendezvous **to perform** scheduling and internal actions.

```
    scheduled module CHARACTER-BUFFER is
        procedure PUT-CHAR (C : in CHARACTER);
        procedure GET-CHAR (C : out CHARACTER);
        procedure NEW-PROCESS-ENTRY;        -- this procedure corresponds to the separate
                                            -- thread of control of the service task
    end CHARACTER-BUFFER;

    scheduled module body CHARACTER-BUFFER is
        MAX       : constant INTEGER := 200;

        subtype BUFFER-POINTER is INTEGER range 0 .. MAX;

        BUFFER   : array (1 .. MAX) of CHARACTER;
        IN_PTR   : BUFFER-POINTER := 1;
        OUT,PTR  : BUFFER-POINTER := 0;

        type ENTRY-NAME is (PUT-CHAR, GET-CHAR) ;
        subtype SYNCHRONIZATION-LEVEL Is INTEGER range 1 .. 3;
        SL : SYNCHRONIZATION-LEVEL;         -- this variable Is used to track which accept
                                            -- or select statement Is being executed

        scheduler BUFFER,SCHED is
            imports (IN,PTR, OUT,PTR : in; SL : in out);
            procedure ENTER (E : in ENTRY-NAME);
            procedure COMMON-EXIT;
            procedure AWAIT;
        end BUFFER,SCHED;
```

```
procedure PUT-CHAR (C : in CHARACTER) is
     scheduling (ENTER (PUT-CHAR), COMMON-EXIT);
begin
     BUFFER(IN_PTR):= C ;          -- executed by the ceiling process
end PUT-CHAR;

procedure GET-CHAR (C : out CHARACTER) is
     scheduling (ENTER (GET-CHAR), COMMON_EXIT);
begin
     C := BUFFER(OUT_PTR);         -- executed by the calling process
end GET-CHAR;

procedure NEW-PROCESS-ENTRY is
begin
     loop                          -- executed by the thread of control of
                                   -- the buffer
          SL := 1;
          BUFFER_SCHED.AWAIT;      -- schedule entry calls and suspend
                                   -- until entry call is complete
          case SL is               -- current value of SL determines
                                   -- which call was accepted
               when 2 => IN,PTR := IN,PTR mod MAX + 1;
                              if OUT,PTR = 0 then
                                   OUT,PTR := 1;
                              end if;
               when 3 => OUT,PTR := OUT,PTR + 1;
                              if OUT,PTR = IN,PTR then
                                   OUT,PTR := 0; IN_PTR := 1;
                              end if;
               when others => null;
          end case;
     end loop;
end NEW-PROCESS-ENTRY;

-- NOTE: the bodies of the visible procedures above contain the translation of
-- the Ada source statements; the scheduler procedures below contain the
-- implementation of scheduling and mutual exclusion fat entry calls which would
-- be provided by the compiler in an implementation of Ada.

scheduler body BUFFER,SCHED is
     PROTECTION  : LOCK;           -- mutual exclusion in module scheduling
     BUSY        : BOOLEAN := TRUE;-- whether module is in use
     ENTRY-OPEN  : array (ENTRY-NAME) of BOOLEAN: -- which entries open
     ENTRY-Q     : array (ENTRY-NAME) of CONDITION;-- queues for names
                                             -- of calling processes
     BUFFER-NAME : PROCESSNAME;-- internal name for thread of control
                              -- of the buffet

     procedure ENTER (E : in ENTRY-NAME) is
     begin
          SET (PROTECTION);
          if BUSY or else not ENTRY-OPEN(E) then     -- module is in use
```

```
                INSERT (ENTRY-Q (E), MYNAME 0);  -- or guard is false
                RESET (PROTECTION);              -- so calling process
                SUSPEND;                         -- suspends itself
            else
                BUSY : = TRUE;              -- call is accepted so set module
                RESET (PROTECTION);         -- in use
            end if;
            case E is
                when PUT-CHAR =>
                    SL : = 2;         -- Put call is being accepted
                when GET-CHAR =>
                    SL : = 3;         -- Get call is being accepted
            end case;
        end ENTER;

        procedure COMMON-EXIT is
        begin
            ACTIVATE (BUFFER_NAME);-- Activate thread of control of buffet
        end COMMON,EX IT;

        procedure AWAIT is
            NEXT : PROCESSNAME;
        begin
            SET (PROTECTION);        --wait for protection on scheduling
            BUFFER-NAME : = MYNAME 0; --setup internal name for buff et
            ENTRY-OPEN : = (IN,PTR /= OUT,PTR, OUT,PTR /= 0) ;
            BUSY : = FALSE;          --anticipate module not busy
            for E In ENTRY_NAME'FIRST . . ENTRY_NAME'LAST loop
                if ENTRY-OPEN(E) and then not EMPTY (ENTRY_Q(E)) then
                    case E is
                        when PUT-CHAR =>
                            SL : = 2;     -- Put call is being accepted
                        when GET-CHAR =>
                            SL : = 3;     -- Get call is being accepted
                    end case;
                    REMOVE (ENTRY-Q (E), NEXT) ; --remove next caller from queue
                    BUSY : = TRUE;   --set module is busy
                    ACTI VATE (NEXT) ; -- and activate
                    exit;
                end if;
            end loop;
            RESET (PROTECTION);      -- release scheduling protect/on
            SUSPEND;                 -- and suspend
        end AWAIT;

    end BUFFER,SCHED;

end CHARACTER-BUFFER;


process NEW-PROCESS Is               -- this process is the separate thread
```

```
                                       -- Of control Of the buffet
    channels CHARACTER-BUFFER;
end NEW-PROCESS;

process body NEW-PROCESS is
begin
    CHARACTER_BUFFER.NEW_PROCESS_ENTRY;
end NEW-PROCESS;


process PRODUCER is
    channels CHARACTER-BUFFER;         -- the body of PRODUCER contains calls to
                                       -- CHARACTER_BUFFER.PUT_CHAR
end PRODUCER;

process CONSUMER is
    channels CHARACTER-BUFFER;         -- the body of CONSUMER contains calls to
                                       -- CHARACTER_BUFFER.GET_CHAR
end CONSUMER;
```

Note that the scheduling used for calls to the CHARACTER-BUFFER will accept PUT's before GET's whenever the Buffer is not full, This selection scheme is consistent with the specification of Ada, which only requires that the choice among open alternatives be "performed arbitrarily? In general, however, identification of an optimal selection scheme depends on the global semantics of a program, so it is not possible to make such an identification in the syntax directed translation used with the Adam compiler. The method of selection implemented in Ada to Adam translation utilizes a pseudorandom number generator to make a choice among the open aternatives. Thus, the general implementation of select is random, which is also consistent with the Ada requirement for arbitrariness.

APPENDIX H: COMPILER COMMANDS,

This appendix briefly describes a compiler we have implemementd for Adam. The compiler is written in Maclisp, runs on a PDP-10, and produces POP-10 assembly language code. The compiler is interactive. It accepts commands from the terminal user to compile files, manipulate libraries, etc. It has three phases: a parser which constructs an abstract syntax tree, a phase which does static semantic checking, and a code generation phase. The parser is constructed by an SLR parser generation system.

The compiler supports the Ada separate compilation facility. A compilation consists of a library file and a set of compilation units. A compilation unit can be a module specification, module body, subprogram body, or module body subunit. The main program is designated by having a subprogram body compilation unit with the name MAIN, or by having a subprogram body compilation unit which has the pragma MAIN in its outermost declarative part. Compilation units can refer to units already in the library and the new units in a compilation will be added to the library or replace old units with the same name in the library. The compiler has commands to create libraries, open and close libraries, list the table of contents of libraries, etc.

1. Compiler Commands.

The compiler is invoked by typing:

```
r adam
```

The compiler prints a prompt and waits for a command to be typed.

A command to the compiler consists of a command name, or a command name followed by a list of arguments. All commands are terminated by a semicolon. Arguments are separated by commas. After a command has been executed another prompt is printed and another command can be typed.

When the compiler is initially invoked, all file operations will be defaulted to the job's current directory. The defaults for various kinds of file operations can be changed by the following commands.

Alias <dir>;	- will change the default directory for all operations to <dir>.
Input <dir>;	- will change the default directory for source program Input to <dir>.
Output <dir >;	- will change the default directory for any output which the compiler generates to <dir>.
Libdir <dir>;	- wili change the default directory for library related i/o to <dir >.

where <dir> has the form <directory-name> on TOPS-20 or p, pn on SAIL WAITS.

Quit; - terminates the compiler.
Help; - prints a list of the available commands.

2. Compiling a Source File.

Before any source file is compiled a library must be opened. See section on libraries for how to do this.

To compile a source file, which is in the file, name.ada, the command is:

Compile name;

This has the following effect. The file is opened and the program is parsed. If there is a syntax error the compiler returns to the command level. If there are no syntax errors the static semantic checking is done. if there are semantic errors, messages will be Written to the terminal and also to a file called name.err. The compilation units in the source file will be inserted into the open library, If there are no semantic errors code generation is done. The compiler generates a file of Fail source code called name.fai.

To run a main program which is in the currently open library, the command is:

Execute main-program-name;

This command tests for completion of the the compilation tree rooted at the main program. If it is complete a do file is created with commands to link and run the program, the compiler terminates.

The command:

Compile;

will recompile the most recently compiled file from the current compilation session.

3. Separate compilation and libraries.

A library consists of a set of compilation units. The information recorded for a compilation unit in a library includes: Its name, the kind of unit it is, the name of the source file it came from, the name of the file to which code was generated for it, the time and date it was compiled, its with requirements of other units in the library, and a copy of the abstract syntax tree for the unit. For non-generic units an abstract syntax tree for the specification part is kept. For generic units the abstract **syntax tree for the entire unit is retained.**

All library operations during **a** compilation session are done with **respect to the currently open** library.

Commands for libraries.

To **create a** new library the command is:

```
Create name;        -- this creates a file name.lib
                    -- in the default library directory
```

To open an existing library the command is:

```
Open name;          -- looks for name.lib in the default library
                    -- directory and opens it if found
```

To close the currently open library the command is:

```
Close;
```

A quit command will also close any open library.

To copy **a** library unit from some library to the currently open **library the command is:**

```
Copy 1 ibraryname. uni tname;
```

Example:

```
Open myli b;        -- open an existing library
copy io. tty,io;    -- copy a tty i/o module called tty_io from the
                    -- library io to the library mylib
```

The following commands **can** be applied to an open library.

```
Dir;                -- list the table of contents of the library
Tdir;               -- list times when units were compiled
Fdir;               -- list the files from which units came and were compiled into
Wdir;               -- list the 'with' requirements of the units in the library

Remove uni tname;   -- remove the named unit from the library.
```

An example of a command sequence to create a library of utility modules:

```
. r adam
...                 -- compiler prints signon message and a prompt
-> create util;     -- create and open a new library named util.lib
-> compile util;    -- compiles a file of units from the source
                    -- file util.ada
                    -- generates code to a file named util.fal
Adding STACK to the library -- complier comments
Adding RAT-NUMBERS to the library
```

```
-> close;             -- close the library
-> quit;              -- terminate the compiler
```

At a later compilation session one can type:

```
...
-> open util;         -- open an existing library
-> dir;               -- see what's in it

module STACK . . .
module RAT-NUMBERS . . .

-> compile foo;       -- compile some file that uses units in the library
-> open mylib;        -- open some other library, closes util
-> compile bar;       -- compile some other file
-> quit;              -- close any open library and terminate
```