COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305

Adam
An Ada-based
Language for Multiprocessing

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

Technical Report No. 83-240

May 1983

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contracts MDA 903-80-C-0159 and NOO-039-82-C-0250.

Adam
An Ada[1]-based
Language for Multiprocessing

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

Technical Report No. 83-240

May 1983

Abstract:

Adam is a high level language for parallel processing. It is intended for programming resource scheduling applications, in particular supervisory packages for runtime scheduling of multiprocessing systems. An important design goal was to provide support for implementation of Ada and its runtime environment. Adam has been used to implement Ada task supervision and also as a high level target language for compilation of Ada tasking.

Adam provides facilities that match the Ada sequential constructs (including subprograms, packages, exceptions, generics). In addition there are specialized module constructs for implementation of packages that may be shared between parallel processes. Adam omits the Ada real types but includes some new predefined types for scheduling. The parallel processing constructs of Adam are more primitive than Ada tasking. Strong restrictions are enforced on the ways in which parallel processes can interact.

A compiler for Adam has been implemented in MacLisp on DEC PDP-10 computers. Runtime support packages in Adam for scheduling (on a single CPU) and I/O are also provided. The compiler contains a library manipulation facility for separate compilation.

The Adam compiler has been used to build an Ada compiler for most of the July 1980 Ada language design including task types and rendezvous constructs. This was achieved by implementing algorithms translating Ada tasking into Adam parallel processing as a preprocessor to the Adam compiler. This present Ada compiler, which has been operational since December 1980, uses a procedure call implementation of tasking (due to Habermann and Nassi and to Stevenson). It can be easily modified to other implementations. Compilation of Ada tasking into a high level target language such as Adam facilitates studying questions of correctness and efficiency of various compilation algorithms, and code optimizations specific to tasking, e.g. elimination of unnecessary threads of control.

This paper gives an overview of Adam and examples of its use. Emphasis is placed on the differences from Ada. Experience using Adam to build the experimental Ada system is evaluated. Design of runtime supervisors in Adam and algorithms for translating Ada tasking to Adam processing are discussed in detail.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contracts MDA 903-80-C-01 59 and NOO-039-82-C-0250.

---

[1] Ada is a registered trademark of the U.S. DoD (AJPO).

Table of Contents

1. Introduction. 1

1.1 NOTATION AND TERMINOLOGY. 3

2. Overview of Adam. 5

2.1 TYPES AND DECLARATIONS. 5
2.2 STATEMENTS AND SUBPROGRAMS. 5
2.3 MODULES. 6
2.4 PROCESSES. 6
2.5 PROGRAM UNITS, VISIBILITY AND IMPORTS. 7
2.6 EXCEPTIONS. 8
2.7 PROCESS SUPERVISION. 8

3. Types for Scheduling. 11

3.1 LOCKS. 11
3.2 PROCESS NAMES. 11
3.3 CONDITIONS. 12

4. Modules. 15

4.1 SCHEDULERS. 16
4.2 SCHEDULED MODULES. 17
4.2.1 SCHEDULING DECLARATIONS. 17
4.2.2 RESERVE STATEMENT. 18
4.2.3 EXCEPTIONS IN SCHEDULED MODULES. 18
4.3 DEVICES. 18
4.3.1 MACHINE CODE. 19
4.3.2 INTERRUPTS. 19
4.4 EXAMPLES. 19

5. Processes. 25

5.1 DECLARATION OF PROCESSES. 25
5.2 CHANNELS DECLARATIONS AND USE CLAUSES. 26
5.3 INSTANTIATION OF GENERIC PROCESSES. 27
5.4 INITIATION OF PROCESSES. 28
5.5 TERMINATION OF PROCESSES. 28

6. Visibility Rules and Imports. 31

6.1 VISIBILITY. 31
6.2 IMPORTS. 31
6.2.1 REDUNDANT IMPORTS DECLARATIONS 32
6.2.2 EXAMPLES. 32
6.2.3 ENCAPSULATED IMPORTS. 33

7. Experience and Conclusions 35

7.1 COMPILER DEVELOPMENT 35
7.2 WRITING SUPERVISORS 36
7.3 TRANSPORTING THE ADAM COMPILER AND RUNTIME SUPERVISOR 36
7.4 PROCESS NAMES 37
7.5 TRANSLATING ADA MULTITASKING 37

7.5.1 ADVANTAGES OF USING A HIGH LEVEL TARGET LANGUAGE 38
7.5.2 USING ADA AS A TARGET FOR TRANSLATING ADA 38
7.5.3 MULTIPROCESSORS AND OPTIMIZATION 38
7.5.4 DISADVANTAGES OF THE TRANSLATION TECHNIQUE 39
7.5.5 FUTURE RESEARCH 39
ACKNOWLEDGMENTS: 39

Appendix A: A Standard Supervisor. 41

Appendix B: Ada Multitasking Translation Example. 5-I

1. INTRODUCTION.

Adam is a high level language for parallel processing. It is intended specifically for programming resource scheduling applications. It has also provided a simple and flexible methodology for two stage implementation of experimental compilers for the Ada language throughout the design changes from the preliminary 1979 and 1980 designs [9] to the 1982 ANSI standard.

Adam was designed and implemented during 1979 — 1980. An important design goal was to provide support for the implementation of Ada and its runtime environment. Adam was developed independently of the official Ada project with the objective of producing experimental Ada implementations quickly with a small investment in manpower.

Two main goals motivating the design were:

1. to provide a high level language suitable for construction of efficient runtime supervisors for parallel programs intended to run on either single or multiple processor hardware,
2. to provide a high level target language for translation algorithms for Ada tasking.

Based on these goals, the Adam design remains close to the preliminary Ada sequentia! (non tasking) design [9] but provides a more flexible lower level parallelism. The parallel constructs are closer to constructs in previous languages such as Concurrent Pascal [I] and Modula [14] so that techniques for their compilation are already well understood.

The-sequential constructs of Adam are essentially the sequential constructs of Ada augmented by constructs for scheduling parallel computations and communication between them: (1) new predefined types *Process Names, Locks,* and *Condition Variables,* and (2) special kinds of structured packages called *Scheduled Modules.* Parallel computations are represented by *Processes.* Processes are program units, similar to procedures, which may be initiated and executed in parallel. Processes may communicate only by operating on scheduled modules. The runtime scheduling of processes is explicitly controlled by the user program and is not built into the semantics of processes. The visibility rules of Adam are stricter than Ada, the intention being to make it very easy to determine from the source text how parallel computations can interact. The main ommissions from the sequential part of Ada are the real types.

In additon to the language itself, certain requirements are placed on its environment. The Adam environment is required to include a runtime supervisor module which must provide six standard interface procedures. These procedures are implementation dependent; their semantics is left open, but a "standard" interpretation is indicated. An Adam compiler pragma enabies a user to change the runtime supervisor module. (c.f., discussion of programming process scheduling in [8].) This approach to runtime supervision provides flexibility in changing supervisors and in interfacing user programs with the underlying supervisor. It also improves portability of the compiler.

Design goal (1) 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. [8] and [10].) It is therefore necessary to study the structure of runtime supervisors and to develop languages that facilitate their construction.

Simple supervisory packages can be written in Ada directly, but with difficulty. The programmer must follow a very strict discipline in coding critical regions both to ensure correctness and to distinguish between scheduling code and computation for purposes of documentation and readability. He must also be willing to simulate low level protection with high level constructs (e.g. semaphores encoded as tasks).

Adam contains facilities specifically aimed at helping the programmer cope with these problems. A simple discipline in structuring critical regions is enforced by the *scheduled module construct.* Scheduling operations are separated from the operations that need to be scheduled and the two kinds of operations are associated by a new kind of declaration called scheduling declarations. Adam also provides predefined low level types for protection and scheduling: Locks, Process Names and *Condition Variables.* The Process Name type is particularly important in constructing resource scheduling in user programs and in interfacing such schedulers with the underlying standard runtime supervisor (see discussion in Section 7.4).

If a runtime supervisory package is itself a parallel program (as might well be the case in a multiprocessor system) it cannot be written in Ada using the task rendezvous constructs (for then there would need to be a "sub-supervisor" to schedule rendezvous in the main supervisor, and changes in the compiler so that compilation of the main supervisor would interface with the sub-supervisor). In Adam it is anticiparted that supervisory packages will involve distributed and parallel processing. Therefore parallel processes in Adam have a semantics which do not imply any particular underlying scheduling. The scheduling of processes is always controlled by the user program. If a supervisor package has parallel processes, a subsupervisory kernel may be written specifically to schedule these particular processes by using their process names.

Goal (2) 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 Rationaie was by no means the only viable model for implementation. Alternative operational models were suggested in [6] and [13]. There are several advantages to giving a precise definition of these algorithms at a very high level, i.e., as translations from Ada to an existing high level parallel language in which the parallelism is lower level than Ada tasking: (a) the translation from Ada to the target language could be a preprocessor to a compiler for the target language; this should result in an easy implementation of a compiler for Ada based on a compiler for the target language. (b) It provides a capability to change the compilation algorithm quickly or to have alternative algorithms available in the compiler. (c) Precise definition of translation algorithms (e.g. as input/output relations between Ada source and programs in the target language) gives us the possibility of formulating and proving their correctness.

To achieve goal 2, the semantics of parallelism in the target language must already be well understood. Existing languages with clearly defined parallelism included Concurrent Pascal and Modula. So the parallelism in Adam is closely related to processes in these two languages.

In Adam we chose a form of parallel processes in which 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. These restrictions include: (a) limiting the kinds of global objects that may be accessed by processes, and requiring that such access to be explicitly declared in process specifications, and (b) requiring import declarations on modules to specify the use (inside the module body) of outside (global) variables and modules.

The reason for restricting visibility in the Adam design is to make possible compiletime checking for many common errors due specifically to failure in communication between parallel processes. An immediate benefit is to ensure that the translations of Ada tasking conform to a structure which makes them easy to specify and analyse. 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 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.

A compiler and runtime supervisor for Adam were implemented and running by June 1980. The compiler was implemented in Maclisp, and the supervisor in Adam. Following the ideas of goal (2), an Ada to Adam translation algorithm for Ada tasking was then implemented. The resulting two stage compiler for most of July 1980 Ada design was demonstrated at the Ada Symposium in Boston, December 1980. This compiler has been used for over two years as an instructional tool and as a basis for other research projects in Ada.

The structure of this paper is as follows. Chapter 2 gives an overview of the Adam language **and** a rationale. Chapters 3 - 5 explain in detail those constructs of Adam that differ from Ada, and give simple detailed examples of their use. Chapter 3 discusses the new primitive types for scheduling; the different kinds of modules provided by Adam are presented in Chapter *4;* Chapter 5 deals with processes. Chapter 6 explains the rules for visibility and importation of objects. Chapter 7 gives a summary of our experience in implementing Adam, and using Adam to build a compiler for preliminary Ada. Experience in transporting the compiler and runtime supervisor is described. Appendix A gives a detailed Adam source text for a simple runtime supervisor. This illustrates the structured design of a portable supervisor using Adam. Appendix B gives details of the preprocessing algorithm translating Ada tasking into Adam processing that was implemented. Other' translation algorithms that could be implemented similarly are discussed in [13].

1 .1 NOTATION AND TERMINOLOGY.

The paper assumes that readers already have some familiarity with Ada. The syntax of Adam follows the notation and formatting conventions of Ada with a few exceptions In syntax descriptions square brackets, [ ], indicate an optional construct; curly brackets, { }, indicate zero or more repetitions of a construct.

Modules and processes are called *units.* Nongeneric entities are often called *actual.* Variables and actual modules are called *objects.* Entities declared in the specification — or visible — part of a module are said to be *exported* by the module. Entities declared in the body of a module are said to be *encapsulated* by the module.

2. OVERVIEW OF ADAM.

Adam consists of almost all of the non-tasking constructs of the Ada language, July 1980 [9], augmented by some simple primitive constructs for scheduling and parallelism. This section briefly discusses the areas where Adam differs from Ada and presents a short overview of the scheduling and parallel features of Adam together with the rationale for them. Details and examples are given in later sections.

2.1 TYPES AND DECLARATIONS.

The types and declarations of Adam are essentially those of Ada. The type facilities in Adam are not. as rich as Ada, the general philosophy being to provide only a simple set of types sufficient to support construction of multiprocess programs. Accordingly, non-integer numeric types and type conversions have not been included in Adam, and range constraints must be static.

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

Process Names. Process names provide **a** means of referring to a process (or thread of control).
Conditions. Condition variables provide a FIFO queue of process names.
Locks. Variables of type Lock provide a low-level facility for programming critical regions.

Adam provides a set of operations for each of these types; for details see Section 3.

The facility of having names for processes is important in programming scheduling of shared variables (or any interaction between processes), supervision of resources, and message-passing operations. One might think that locks and conditions could as well be defined as (exported or standard) modules. However, Adam follows Ada in not treating modules as types; thus modules cannot be passed **as** parameters. Not having locks and conditions available as parameters would not permit to write scheduling code as clearly and succinctly as with lock and condition variables. Furthermore, it would make it impossible to express certain scheduling strategies that depend on critical sections being locked up across procedure calls (in particular calls to supervisor procedures; for such a situation see the notes following Example 4-5).

2.2 STATEMENTS AND SUBPROGRAMS.

The *statements* of Adam include all of the sequential statements of Ada. Since the multitasking in Adam differs from that of Ada there are no delay, abort, select, accept, or terminate statements in Adam. However, Adam provides two kinds of statements not in Ada: **reserve** for reserving **a** scheduled module (see Section 4.2.2), and **initiate** for the initiation of processes (see Section 5.4).

Adam *subprograms* are the same as in Ada (including generic subprograms), with two exceptions:

1. A subprogram that is part of a scheduled module may be linked to procedures of a scheduler by a *scheduling declaration* (see Section 4.2.1). A scheduling declaration is included in the declarative part of the procedure body.

2. Adam provides *interrupt procedures,* which are special procedures that are called directly from the hardware (see Section 4.3.2).

2.3 MODULES.

Modules in Adam correspond to packages in Ada. They provide facilities for encapsulation or abstraction Modules may be generic; generic modules are declared and instantiated as in Ada. In addition Adam provides special modules:

- *Schedulers,*
- *Scheduled modules, and*
- *Device modules.*

Schedulers are used to group and encapsulate scheduling and synchronization operations. They are intended to simplify the overall structure of scheduled modules by supporting a clear distinction between scheduling and regular computation. They also serve to enforce a discipline of stating the intended biding of scheduling operations to individual subprograms in a scheduled module by use **of** scheduling declarations (Section **4.2.1).**

Scheduled modules are the units for communication between parallel threads of control (processes) and therefore provide a facility for scheduling operations concurrently. A scheduled module is simply a module containing a scheduler; its visible operations are associated with scheduler operations by scheduling declarations (see Section **4** for Details). the actual scheduler is part of the user program (as opposed, for example, to Monitors in Concurrent Pascal which have a one-at-a-time scheduling imposed by the language semantics). In Adam, a scheduled module may be constructed so as to permit simulaneous operations. The *monitor* construct of Concurrent Pascal **[1]** and the *mailbox* concept of Gypsy [5] are particular instances of scheduled modules.

Device modules — or devices, for short — are intended for encapsulating those parts of a system that are machine-dependant; they provide the interface to operating system and hardware. Devices may contain machine-coded operations and interrupts. A device module may be a scheduled module. **A** device module often represents the interface to an actual hardware device, and the **scheduler** provides appropriate access control. Device modules are similar to devices in Modula [14].

2.4 PROCESSES.

Processes are program units which may be run in parallel. 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 There are no means **of** communication among processes other than channels; in particular, processes do not import objects, may not reference external values, and do not have exports (see Sections 2.5 and 6 for details).

Execution of a process is begun by means of the **initiate** statement (see Section 5.2). 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 Section 5.3).

Processes may be generic, but may have only generic **type** and in parameters. A generic process may be instantiated to an actual process.

In comparison with Ada, the constructs in Adam for communication between processes are lower level than the Ada Rendezvous and more restricted. Their choice has been motivated by several considerations. First, the Adam constructs can easily be implemented. Second, they enforce a disciplined and controlled style of interaction among processes. All possible communication among processes in a system can be determined from the specification parts of the process declarations; this should also aid in understanding multi-process systems, in developing specifications for them and in checking their correctness. (In Ada, for example, the bodies of consumer tasks must be examined to determine which service tasks they communicate with.) Third, Adam provides to the programmer great flexibility in organizing the scheduling of processes. Adam is not restricted to the FIFO entry queues of Ada. The language provides the means for constructing schedulers that interact smoothly with the runtime supervisor in a multiprocess system. These capabilities should be close to the kind of programming required in constructing embedded computer systems. For high level multiprocessing it can be expected that standard library schedulers and runtime supervisors are available in separately compiled units, so that the programmer is not burdened by having to construct them. Fourth, the Adam multiprocessing constructs have been designed so that Ada multitasking systems can be translated into Adam multiprocessing systems [13]. Adam thus serves as the vehicle for developing an implementation of the Ada language. (For experience with this approach see Section 7.)

2.5 PROGRAM UNITS, VISIBILITY AND IMPORTS.

The visibility rules of Adam are adopted from Ada with several modifications which restrict visibility of objects, which include variables and non-generic modules, and subprograms. First, the outside objects visible within a process are exactly the scheduled modules declared as communication channels of the process. No other outside object may be mentioned within a process. Second, all outside objects mentioned inside a non-generic module must be declared as *imports* of the module; generic modules may not contain imports declarations and therefore cannot access objects declared outside. Third, subprograms declared outside a module or process are not visible inside those units. Unlike objects, subprograms cannot be imported, either, unless as part of an encapsulating module.

The purpose of these rules is to ensure that all possible interactions between processes are deducible from the declarations of processes and communication channels. The entities that processes may use to influence each others' computations are the objects; they can be enumerated by taking the transitive closure of the channels, their imports, imports of imports, and so on.

The *objects,* i.e. the elements that may be used for communication, are defined to be those entities that have accessible values or states that may vary during a computation; these include *variables* **and** *actual modules.*

Most other entities in programs, including types, subtypes and generic units, are purely definitional in nature and do not have states, thus cannot be used as shared comminucation media. (The visibility rules of Adam are intended to ensure that generic units are purely definitional.) For these entities, 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 usability considerations. A subprogram declaration is definitional unless there are global objects. We could require imports declarations on subprograms, in which case their visibility would not need to 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 unencapsulated 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.

Details of the rules for visibility and importation in Adam are given in Section 6.

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 EXCEPTIONS.

Exception handling and propagation is the same as Ada except that there is no direct propagation of exceptions between processes. Exceptions that are raised during scheduling are treated slightly differently; for details see Section 4.2.3.

2.7 PROCESS SUPERVISION.

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 multiprocessing constructs of Adam do not imply any particular scheduling.

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

The Adam language environment is required to contain a precompiled scheduled module, *supervisor,* that implements a set of visible standard procedures for activating and suspending processes:

```
procedure START     (D : INIT_DATA);
procedure SUSPEND;
procedure ACTIVATE (P : PROCESSNAME);
procedure SWITCH (P : PROCESSNAME);
procedure DELAY-FOR;
procedure FINISH:
```

These procedures and the type IN IT-DATA (process initialization data) are implementation-dependent. Their semantics is therefore left open, and it should be emphasized that the procedures are not part of the language itself. However an intended interpretation for single CPU implementations is indicated as follows. 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 ACTIVATE 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. An Adam implementation of these supervisor procedures is given in Appendix A.

It is recommended that an Adam compiler should use only the standard supervisor procedure calls in the compilation of processes. Calls to START and FINISH are generated by the compilation of initiiation and termination of processes respectively. 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 processes.

If the interface of standard supervisor procedures is all that an Adam compiler assumes about supervision, it is 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 calls 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 A 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 START and FINISH procedures may be the only non-trivial ones in a user-supplied supervisor for that application.

3. TYPES FOR SCHEDULING.

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

3.1 LOCKS.

Variables of type LOCK are used to implement primitive mutual exclusion. Locks may be in one of two states, which may be thought of as 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 uhtil L is OFF, then gains exclusive access to L and changes the state to ON.

RESET (L : **in 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 3-1: The <body> is protected by L from simultaneous execution.

```
    L : LOCK;                    -- variable L is declared to be a loch with
                                 -- initial state 0 F F.
begin
    SET (L);                     -- busy wait to gain access to L and <body>.
    <body>
    RESET (L);                   -- reset L so next user may gain access.
end ;
```

3.2 PROCESS NAMES.

Whenever a process in initiated it is assigned a unique value of type PROCESSNAME. This is the only way that process names can be created. The only permissable operations are assignment, equality, and interrogation using the function 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.

3.3 CONDITIONS.

A value of type CONDITION is a FIFO queue of process names. Variables of type CONDITION are called *condition variables.*

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

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 process name from the queue of CV and returns it as the value of P; raises CONDITION_QUEUE_EMPTY exception if the queue of CV is empty.

Example 3-2: **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());
        RESET (CVL);
        SUSPEND;                        -- Supervisor call to suspend caller.
    else
        RESET (CVL);
    end if;
end ;

procedure SIGNAL (CV : in out CONDITION; CVL : in out LOCK) is
                                        -- removes the first process name from the
                                        -- queue CV (if nonempty) and activates it.
                                        -- CVL is a unique lock protecting all
                                        -- operations on CV.
    P : PROCESSNAME;
begin
    SET (CVL) ;
    if <some-condition> and not EMPTY (CV) then
```

```
            REMOVE (CV, P);
            RESET (CVL) ;
            ACTIVATE (P);                    -- Supervisor call to activate P.
        else
            RESET (CVL);
        end if;
    end ;
```

Another example of use of condition variables is given in Section 5.5, Example 5-4.

4. MODULES.

Module declarations in Adam are the same as for Ada packages except:

1. Global objects in generic module declarations are not permitted.

2. Declaration of global objects imported into nongeneric modules is required.

3. Scheduling declarations are used in a scheduled module body to associate visible operations with scheduling operations.

4. Different kinds of modules can be defined:

module — basic module,

device — bodies of devices may contain machine code and interrupts,

scheduler — provides procedures 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.

Modules, schedulers and scheduled modules may be generic. Generic modules are declared and instantiated as in Ada.

The general format for module declarations is:

```
[generic generic parameter list] .
 module M is
     [imports (imports list):]
      other declarations and specifications
[private . . . ]                          -- private part
 end [M];
        ...
 module body M is
     [imports (imports list);]  --  encapsulated imports, see Section 6
      other declarations        --  must include the bodies of all operations
                                --  and modules specified in the visible part
 end[M];
```

Note:

For an explanation of imports list see Section 6.2.

Example 4-1: Adam version of the visible part of the ON-STACKS example from the Ada Reference Manual [9], p. 12-5.

```
generic
     SIZE : INTEGER;
```

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

    procedure POP (S: in out STACK; E: out ELEM);

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

4.1 SCHEDULERS.

Schedulers are intended to encapsulate both the synchronization and protection for scheduled modules shared between processes. Schedulers implement *(a)* the scheduling procedures for entry to, and exit from module operations, *(b)* procedures REQUEST and RELEASE (for the **reserve** statement), and *(c)* procedures for synchronization between module operations. Scheduler procedure declarations follow the normal format for procedures, 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 4-2: A common format for scheduler declarations 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                  -- scheduler for M.
        imports (list of local variables of M);
        procedure REQUEST;              -- REQUEST and RELEASE are procedures
        procedure RELEASE;              -- used for scheduling reserve (Sec. 4.2.2),
        procedure ENTER (...);          -- procedures used to schedule entry to
                                        -- and exit from operations of M.
        procedure LEAVE (...);
        procedure DELAY (...);
    end SCHED;

                                        -- Other declarations in the body of M; this
                                        -- will contain bodies for each visible
                                        -- subprogram of M together with scheduling
                                        -- declarations using procedures of SCHED,
                                        -- and the body of SCHED
```

end M;

4.2 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** statement, then the scheduler for the module must provide REQUEST and RELEASE procedures.

When a generic scheduled module is instantiated, each new instance of the module has a new instance of the local scheduler.

4.2.1 SCHEDULING DECLARATIONS.

A scheduling declaration may be included in the declarative part of a subprogram in a scheduled module. A scheduling declaration has the following format:

scheduling (scheduling-item, scheduling-item);

where each scheduling_item is either **null** or a call to a visible procedure of the local scheduler. The first scheduling item indicates the scheduling before entry to the scheduled procedure, and the second scheduling on exit from the procedure. A scheduling_item **null** means that no scheduling action is to be executed.

A scheduling declaration for a procedure P in a scheduled module with scheduler S,

scheduling ($p(L_1), p_2(L_2)$);

has the effect that the body of P is compiled as

```
S.p1 (L1);                    -- omitted if p1(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, explicit calls to scheduler procedures are used.

4.2.2 RESERVE STATEMENT.

The reserve statement is used to reserve a scheduled module. It allows a process to perform a sequence of module operations 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 4-3: 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;
```

A **reserve** statement is compiled using the REQUEST and RELEASE operations of the module's scheduler. If the scheduler of the scheduled module is S, the **reserve** statement 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.

4.2.3 EXCEPTIONS IN SCHEDULED MODULES.

If an exception which is unhandled reaches the outer level of a scheduled operation the operation's exit procedure is executed before the exception is propagated.

If a scheduling operation propagates an exception, then a special Adam exception, SCHEDULING-ERROR, is propagated to the subunit that called the scheduled operation, i.e., scheduling errors are not handled in the scheduled operation itself.

4.3 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. Device modules may be generic if they do not contain interrupts procedures. When a generic scheduled device is instantiated, each new instance of the device has a new instance of the local scheduler.

4.3.1 MACHINE CODE.

Machine code is inserted into a program through the use of an aggregate as in Ada. (An example is given in Appendix A-see the CPU module body.) Unlike Ada, both machine code and Adam statements may appear in the same subprogram; this simplifies writing machine-dependent code.

4.3.2 INTERRUPTS.

Interrupts are special procedures that are called directly from the hardware. Interrupt procedures may occur only in device modules. Interrupt procedures may not be generic and may not have parameters; they may access global variables (see Example 4-6).

Interrupt procedures are declared by:

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

Note: A separately compiled device which has an interrupt procedure in its body must contain a subprogram header for that procedure in its private part.

4.4 EXAMPLES.

Example 4-4: 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 R-ENTER;                   -- first scheduler operation,
    procedure W-ENTER;                   -- second scheduler operation,
    procedure RW_EXIT;                   -- 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 (R-ENTER, WR_EXIT) ;
                                         -- scheduling for READ-see below,
```

```
        ...

    procedure WRITE (Y : in ITEM) is
        scheduling (W-ENTER, WR_EXIT);
                                        -- scheduling for WRITE-see below,
        .
end BUFFER;

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

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

```
scheduler CHAR-BUFFER-SCHED is new BUFFER-SCHED (120) ;
```

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

```
CHAR_BUFFER_SCHED.R_ENTER;      CHAR-BUFFER-SCHED.W-ENTER;
CHAR-BUFFER-READ (X);           CHAR_BUFFER.WRITE (Y);
CHAR_BUFFER_SCHED.WR_EXIT;      CHAR-BUFFER-SCHED.WR-EXIT;
```

Example 4-5: 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 R-ENTER 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);     -- place reader on queue,
        RESET (PROTECT);            -- release BUFFER-SCHED (note 2 below),
        SUSPEND;                    -- supervisor call to suspend caller,
    else
        INUSE := TRUE;              -- prepare to enter free BUFFER,
    end if;
    COUNT := COUNT - 1;             -- reduce number of items in BUFFER?
    RESET (PROTECT);                -- release BUFFER-SCHED.
end R-ENTER;

procedure W-ENTER 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 number of items in BUFFER,
        RESET (PROTECT);                    -- release BUFFER-SCHED.
    end W-ENTER;

    procedure RW-EXIT is
                                            -- schedules exit from READ and 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 call to activate a writer
        else
            INUSE := FALSE;                 -- else BUFFER is free
            RESET (PROTECT);
        end if;
    end RW-EXIT;

    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 RW-EXIT before SUSPEND is executed by P itself in R-ENTER or W-ENTER. This will not cause blocking only if the supervisor can remember an ACTIVATE that arrives ahead of the matching SUSPEND.

   An alternative design of supervisor calls is to permit locks as parameters of ACTIVATE and 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 4-6: 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 executes 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 90. 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    IO-RESULT is (ERR, OK);

    procedure SEND (L : in LINE; R : out IO-RESULT);
end LINE-OUT;

scheduled device body LINE-OUT is

    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> ...        -- procedure containing machine code to send
                                  as a character to the device; see Appendix A.
        ...
    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 IO-RESULT)  is
        scheduling (ENTER, LEAVE) ;
    begin
        LINE-STORE := L;
        CURRENT-CHAR := 1;
        INITIALIZE-DEVICE (LINE_STORE(CURRENT_CHAR), DEVICE-STATUS);
        if DEVICE-STATUS = OK then
            LINE_SCHED.AWAIT;
```

```
    end if;
    R := DEVICE-STATUS;-
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
                                    -- provides synchronization between SEND
                                    -- and the interrupt

scheduler body LINE-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 := MYNAME();
        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;
```

5. PROCESSES.

Processes are program units which may be run in parallel. In order to run, a process must first be initiated. 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. Processes may be generic (but may have only **type** and **in** parameters). Channel parameters (scheduled modules) may also be generic.

5.1 DECLARATION OF PROCESSES.

The genera9 format for a process declaration is:

```
[generic
     generic process parameter 1 ist]
 process p [is
     channels channel s 1 i st;
 end [p]];
```

A process body has the form:

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

where:

1. generic process parameter list has the form,

   list of generic **type** and **in** parameters;
   channels generic-channels-list:

2. generic_channels_list is a list of declarations of the form,

   m **is n(L)[restricted** (operations 1 ist)]

   where n is a generic scheduled module, L is a 9ist 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 forma9 parameters of n by generic forma9 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 5-2, 5-3, and Section 5.4.)

3. channels list has members of the form

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

   where m is an actual scheduled module.

4. 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).

5.2 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.

Example 5-1: 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 FILE-PRINT is
     channels use LPT, DISKFILE ;
end FILE-PRINT;

process body FILE-PRINT is
     type LINE-STORE is array (1 . . 40) of LINE;
     procedure BLOCK-TO-LINES (B : BLOCK: A : out LINE-STORE) is
          . . .                          -- transfers a block to a line store.
     end BLOCK_TO_LINES;

     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                      -- reserve LPT
```

```
            for i in 1 . . 40 loop .
                WRITE-LINE (BUF( i)); -- writeonto LPT
            end loop:
        end reserve ;
    end FILE-PRINT;
```

Example 5-2: 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 TRANS FER must have channels
                                         -- that are instances of BUFFER with
                                         -- the same pair of actual generic parameters.
    process body TRANSFER is
        c : ITEM;
    begin
        loop
            A.READ (C);
            B.WRITE (C);
        end loop:
    end TRANSFER;
```

5.3 INSTANTIATION OF GENERIC PROCESSES.

Instances of generic processes are created as follows:

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

where L is a list of actual generic parameters, and M is a list of actual channels. Rules for matching actual and forma9 generic parameters in instantiation of a generic process extend the Ada rules for generic instantiation [9]. Scheduled modules match generic forma9 channel parameters. The actual channel must be an instance of the forma9 channel obtained by replacing its generic forma9 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 5-3: We continue with the previous example: correct and incorrect instantiations of the process, TRANSFER.

```
scheduledmodule 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 generic parameters in the
                                -- declaration of BUF3 be the same as those
                                -- in the declaration of T2,
                                -- namely, APPLES, 120.
```

5.4 INITIATION OF PROCESSES.

Processes are initiated by the **initiate** statement:

initiate process-list ;

where process_list is a list of previously declared actual processes. Initiation of a process means that the process is activated and may then execute in parallel with any other currently active process

A process may be initiated any number of times; each initiation causes a new copy of the process to be activated and begin execution Copies of a process execute as separate threads of control and can only affect each other by operations on shared channels.

Note. **Initiate** statements could not be replaced by calls to a subprogram (e.g. in the supervisor) since processes are not available as parameters. (After initiation, but only then, a process can be identified by its process name.)

5.5 TERMINATION OF PROCESSES.

The visibility and declaration rules of Adam establish a similar dependency relation for processes as exists for tasks in Ada [9]. 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.

Note: 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 processes 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 sophisticated task supervisor.

6. VISIBILITY RULES AND IMPORTS.

The visibility rules of Adam are essentially those of Ada with additional restrictions placed on the visibility of subprograms and objects within 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. Objects that are not immediately visible within a module may be imported explicitly.

6.1 VISIBILITY.

Entities that are always visible within the body of a unit if they are visible at the point of declaration of that unit (according to Ada visibility rules) are: types (including the constants of an enumeration type and the names of record fields), constants, generic units, exceptions, processes, and predefined system modules (e.g. supervisor, see Section 2.6). Visibility of the remaining entities (subprograms and objects) is controlled by the following rules.

Externally declared *subprograms* are not visible within a module or process, unless they are declared in the visible part of a module and that module is imported into the unit. All subprograms visible at the point of declaration of a subprogram are also visible in the body of that subprogram.

Objects are variables and nongeneric modules, including instances of generic modules. Within modules, externally declared objects are not visible; they can be made visible by explicitly importing them into the module by an **imports** declaration (see Section 6.2). Within processes, externally declared objects are not visible, except those scheduled modules included in a **channels** declaration. There are no restrictions on the visibility of objects within subprograms.

6.2 IMPORTS.

An external object (i.e a variable or actual module) that is mentioned syntactically in a module must be explicitly imported by an **imports** declaration. An imports declaration is part of the specification **or** body of a nongeneric module (see Section **4).**

An **imports** declaration has 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
```

An imports list is **a** list of variables and actual modules. Each imported variable has a mode **in, out, or in out,** and each imported module has the nature **module.** To avoid duplication of **imports** and **use** lists, a **use** declaration may appear within an imports list. Each imported object must be visible at the point of the importing (module) declaration. It is then made visible inside the module. 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.

Generic module declarations may not have imports

The **channels** declaration of a process permits an actual scheduled module to be visible inside a process body, i.e. with respect to visibility a channels declaration acts as an imports declaration This is the only kind of external object that can be made visible inside a process

6.2.1 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.

6.2.2 EXAMPLES.

Example 6-1: 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 Q (...) is
    begin
        ...
        A.P (...);
        ...
    end Q;
end B;

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

    procedure R (...) is
    begin
        ...
        P (...);                             -- call to A.P
        ...
    end R;
end C;
```

Example 6-2: 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) ;
                                    -- nongeneric GLOBAL-STACK must
                                    -- be imported
        module LOCAL-STACK is new STACK ( . . . ) ;
                                    -- generic STACK is visible.
        ...
        PUSH (...);                 -- means GLOBAL_STACK.PUSH (...)
        LOCAL-STACK.PUSH (...);
        ...
    end SYMBOL_TABLE;
    ...
end CATALOGUE;
```

6.2.3 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 6-3: 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 of
                                    -- FILE-TEMPLATE
    end 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;

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

end DISKFILE;
```

7. EXPERIENCE AND CONCLUSIONS

Experience gained during the development of Adam and in using the language is summarized here. A full report on Adam is available [11].

7.1 COMPILER DEVELOPMENT

The Adam language was designed and implemented during 1979 and 1980, motivated by the goals discussed in Chapter 1. An Adam compiler has been running on DEC PDP-10 computers under the TOPS-20 and WAITS operating systems since June 1980. (Waits is the local operating system of the Stanford Artificial Intelligence Laboratory (SAIL).) This compiler is implemented in Maclisp and generates PDP-10 assembly language. The compiler provides a small set of commands permitting users to manipulate library files containing seperately compiled program units. The implementation itself, separate from language design, required less than one man year.

At the same time runtime supervisor and I/O packages were written in Adam, compiled, and formed part of our Adam environment. So far, our runtime supervisors schedule processes on a single CPU; multiprocessor supervisors have not yet been constructed (see 7.2 and Appendix A for further discussion). It should be noted that I/O packages for parallel processing need to be protected against supervisor interrupts and can be programmed naturally in Adam as scheduled devices.

Three algorithms for translating Ada task types and tasking constructs into the lower level Adam processing were defined at this time; an example is given in Appendix B, and a description of the algorithms is given in [13]. One of these translation algorithms, essentially equivalent to the method of Ada task compilation due to Habermann and Nassi [6], was then implemented in the compiler. In order not to duplicate syntax and static semantic checking already provided in the compiler, the translation was not added as a separate preprocessor, but was incorporated as a subfunction of the static semantic checking. This additional implementation required about two man months.

The resulting Ada compiler was running by September 1980. It implemented the sequentiai constructs, including packages, generics and exceptions, and most of the July 1980 Ada tasking constructs. In retrospect we feel that the use of Adam was crucial both in constructing runtime supervisors (see Chapter 1, goal 1 discussion and Sections 7.2 - 7.4) and in structuring Ada compiler development into clearly defined stages. The whole project was structured into a sequence of five clearly defined steps, and progress on each step was monitored weekly: *(i)* define a runtime supervisor interface, *(ii)* implement the Adam compiler using conventional techniques, *(iii)* construct standard environment packages in Adam, compile and test, *(iv)* define algorithms translating Ada tasking into Adam and analyse for correctness (see Section 7.5), and *(v)* implement task translation in Adam compiler. With this project structure, and starting with part-time personnel who had previous experience with Pascal-like languages, we were able to construct a useful experimental Ada compiler and environment within one year.

The compiler has proved to be an excellent instructional tool, mainly for teaching and experimenting with Ada. Many examples of Ada programs, especially tasking, have been compiled and run using separately compiled supervisor packages and I/O; the compiler has been used in teaching courses on Ada programming. More recently, the compiler has been used to support experimental projects in hardware simulation and runtime monitoring and debugging of Ada tasking [4, 7]. A compiler user manual is available [12].

The Adam runtime supervisor has been extended to provide more sophisticated interface facilities [3] so that the compiler can be extended easily to support all of the 1982 ANSI Ada tasking design

7.2 WRITING SUPERVISORS

A supervisor for scheduling processes on a single PDP-10 processor has been written in Adam and compiled. It is the standard package in the Adam environment for providing runtime scheduling for Adam programs with processes and Ada programs with tasks. Two versions are currently in use, one interfacing with TOPS-20, the other with WAITS. Appendix A gives a simplified version of the WAITS supervisor.

A more complex supervisor providing runtime support for the full Ada-82 tasking semantics was developed in Adam [3]. The compiler is currently being modified to interface with this supervisor and to compile all of Ada-82 tasking. The following remarks summarize our experience so far.

(i) All supervisors so far constructed fit very naturally into the structure of a scheduled device module encapsulating both interrupts and machine code. It turns out that the standard supervisor procedures, ACTIVATE, . . . , require only the simple "before and after" protection provided by Adam scheduling declarations. *(ii)* The encapsulation of protection in a separate scheduler subunit and the use of scheduling declarations simplifies the structure of a 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 A 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. *(iii)* 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 A. (In fact, the two versions of the supervisor for WAITS and TOPS-20 differ only in the CPU subdevice.) The Adam device construct, which allows use of machine instructions as simple statements (for instance as part of an **if** statement) greatly simplified the writing of the machine dependent parts of the supervisors. *(iv)* Finally, process names have proved adequate as a means of referring to threads of control in user-defined process scheduling and in interaction between such scheduling and the supervisor.

Further experience is required 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. Currently we are investigating the use of Ada for simulating hardware architectures. These applications require special runtime supervisory packages. The same observations mentioned above as to the naturalness of the scheduled module construct also hold true in these examples By considering a broad class of supervisor implementations, we hope to demonstrate the utility of Adam constructs or, perhaps, to discover generalizations of the Adam notions which will better meet the requirements of process scheduling.

7.3 TRANSPORTING THE ADAM COMPILER AND RUNTIME SUPERVISOR

Transporting the Adam compiler to different operating systems demonstrated the utility of providing a

simple module interface between the compiler and the runtime system. The only changes to the compiler required in moving from the WAITS to the TOPS-20 system involved changes in file naming conventions in the library manipulation facility, and interrupt initialization code in the code generator. All other operating system differences were absorbed by the supervisor written in Adam and, as mentioned above, the required changes were confined to a single local device. In the Ada-82 supervisor, interrupt initialization and timer control were also moved into this local device, further reducing modifications necessary for porting the compiler.

7.4 PROCESS NAMES

The predefined type PROCESS-NAME has proved extremely useful in all manner of scheduling and resource allocation applications of Adam. Interfacing a scheduler in the user source code with the runtime supervisor (e.g. as in examples in Chapter 6) would be impossible without this type.

Ada itself does not provide a primitive TASK-NAME type. Instead Ada relies on the access type, whereby pointers to tasks may be declared, to provide the same functionality. However, the strong typing precludes implementation of "generic" task scheduling units in Ada that are independent of the actual set of task types in a program. Examples of such units that should interface with any tasking program regardless of the task types involved are runtime monitors for detection of deadness errors [4] and supervisory packages for hardware simulation applications. In these applications a TASK-NAME type with facilities for storing and accessing the value of any task's name has to be programmed explicitly in Ada: the techniques for doing this are cumbersome (see discussion in [4]).

A facility similar to the Adam PROCESS-NAME can easily be added to Ada: a new predefined type TASK-NAME is introduced with equality and assignment permitted on values of this type; a new attribute of tasks, TASK-NAME, is introduced; for each task object, t, the attribute, t'TASK_NAME, associates a unique name with t; the function, TASK-NAME, would return the name of the executing task.

7.5 TRANSLATING ADA MULTITASKING

In extending the Adam compiler to support Ada multitasking, we used Adam as an intermediate target language. This resulted in a translation of Ada tasking into the lower level multiprocessing facilities of Adam. The development of this translation proceeded in two stages. First, we defined a mapping from each Ada tasking construct into Adam text which implements its semantics, utilizing the scheduling features of Adam. An example of this type of mapping is given in Appendix B; further details may be found in [13]. In the second step, the semantic processing phase of the compiler was augmented with Maclisp routines which replace abstract syntax tree nodes for Ada multitasking constructs with the corresponding Adam trees. Thus, although the algorithms for translating Ada multitasking were defined by correspondences between Ada and Adam program text, the actual implementation is by a transformation on the internal tree representation of Ada programs during semantic analysis.

7.5.1 ADVANTAGES OF USING A HIGH LEVEL TARGET LANGUAGE

The main advantages of the translation of Ada into Adam were reliability and ease of implementation; others included clarity and understandability of the translator.

Using Adam to specify the translation of Ada multitasking quickly produced a reliable implementation. Errors in the definition of the Ada-to-Adam mapping 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 we had hand generated and analyzed 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 Ada constructs. (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). Less than two man months were required to write and debug the Maclisp code 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 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 write out 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.

7.5.2 *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 all 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. These problems motivated many of the constructs of Adam, and, hence, specifying the multitasking translation should be easier and clearer in Adam.

7.5.3 MULTIPROCESSORS AND OPTIMIZATION

The use of an intermediate transformation into Adam to implement Ada multitasking should expedite research into multiprocessor implementations and optimization techniques. The translation

algorithms we currently use separate all scheduling operations and rendezvous code from the thread of control of a task, thus making it easier both to identify computations which may be truly executed in parallel, and to isolate the critical sections and synchronizations required for multiprocessor environments.

7.5.4 DISADVANTAGES OF THE TRANSLATION TECHNIQUE

The 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 or, alternatively, to use unchecked conversions (with potentially dangerous consequences). In a directly compiled implementation 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 implementation 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 (1979) and was maintained although the tasking was siginificantly revised in July 1980 Ada.

7.5.5 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 implementation 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. These questions are the objects of ongoing research.

ACKNOWLEDGMENTS:

The authors wish to thank Anthony Gargaro for very many helpful comments on an earlier draft of this paper and Ed Falis for comments on the revised drafts.

APPENDIX A: A STANDARD SUPERVISOR.

This appendix presents a simplified version of the standard runtime 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;
                                        -- DEC10_INSTRUCTIONS is a package
                                        -- which defines the formats for inserting
                                        -- machine code.
scheduled device body SUPERVISOR is

    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 : PI-INDEX;
            LAST  : PTJNDEX;
        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 ;
                                          -- = 10 * TICK-LENGTH = 1 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 representation specification
                                          -- is known at link time

scheduler S is
    procedure ENTER:
    procedure LEAVE ;
    pragma INLINE (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     INLINE (EMPTY);

    procedure INSERT (A : in out READYQS; i : PTJNDEX);
    procedure REMOVE (A : in out READYQS; i : in out PTJNDEX) ;
    procedure DELETE (A : in out READYQS; i : PT_INDEX);
    function   EMPTY (A : READYQS) return BOOLEAN ;
    function   FIRST (A : READYQS) return PTJNDEX;
    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     INLINE (ENABLE, DISABLE);

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


function NAME-TO-INDEX (P : PROCESSNAME) return PI-INDEX is
                                        -- convert a process name into a table index
begin
    if P /= null then
        for i in PT_INDEX'FIRST + 1 .. PTJNDEX'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 : PTJNDEX) 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 | 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
        CPU. IDLE;
    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 ( P T (i)) ;
    end if;
end SWITCH;


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


module body Q is
    . . .
end Q;


device body CPU is
    use DEC10_INSTRUCTIONS;
    use REGISTERS;

    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 are
                                            -- 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 are 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 interupt
    SAIL-REG-SAVE : REGISTER-SET;
    SAIL-PC-SAVE  : INTEGER;
    for SAIL-REG-SAVE use at 16;            -- Registers and PC are copied to these
    for SAIL-PC-SAVE use at 87;             -- locations by the WAITS OS when
begin                                       -- an interrupt occurs.
    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;
        DEC10'(MOVEM, 0, (LABEL, "XFER", 1));
        DEC10'(MOVSI, 0, (ADDRESS, INTEGER'(D.REG)));
        DEC10'(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
        DEC10'(HLRZ, 1, (INDEX, 0, 14));
                                        -- get caller's saved return address
        DEC10'(HLRZ, 1, (INDEX, 0, 1));
                                        -- by following dynamic links
        DEC10'(HRRZ, 0, (INDEX, 0, 1));
        D.PC := RO;
        DEC10'(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-INDEX := 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);
                BLOCKEDJOUNT := BLOCKED-TIME - 1;
            else
                i := PCB . PRIOR;
            end if;
        end ;
    end loop;
end ;


interrupt TIMER-INTERRUPT called from 0 is
    i : PT_INDEX;
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 TPMERJNTERRUPT;


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        => RUM,
                          PC            => 0,
                          REG           => (others => 0),
                          PRIORITY      => MAIN-PRIORITY,
                          DELAY-TIME    => 0 ,
                          NEXT | PRIOR=9 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 .. PTJNDEX'LAST - 1 loop
        PT (i).NEXT := i + 1;
        PT (i).NAME := null;
    end loop:
    PT (PTJNDEX'LAST).NEXT := NO-PROCESS;
    PT (PT_INDEX'LAST).NAME :null
    CPU.ENABLE;
    UUO'(CLKINT, (VALUE, TICK-LENGTH));
                                          -- enable timer interrupts
end SUPERVISOR;
```

APPENDIX B: 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 [13].

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 problems in 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 fo
                                         -- 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 fo the separate
                                 -- thread of control of the service fask
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 calling 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 for 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;         -- infernal name for thread of
                                         -- control of the buffer


    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 ());
                                        -- 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);         -- Acfivafe thread of control of buffer
    end COMMON-EXIT;


    procedure AWAIT is
        NEXT : PROCESSNAME;
    begin
        SET (PROTECTION) ;              -- waif for protection on scheduling
        BUFFER-NAME := MYMAME (); -- setup infernal name for buffer
        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
                ACTIVATE (NEXT) -- and activate
                exit;
            end if;
        end loop:
        RESET (PROTECTION);             -- release scheduling protection
        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 buffer
    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 fo
                                          -- 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.

References

[1] Brinch-Hansen, P.
The Architecture of Concurrent Programs.
Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[2] Brinch-Hansen, P. .
The Design of Edison.
CSD Report, University of Southern California, September, 1980.

[3] Falis, E.
Design and Implementation in Ada of a Runtime Task Supervisor.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada,* pages l-9. ACM, Arlington, Virginia, October, 1982.

[4] German, S.M., Helmbold, D.P., and Luckham, D.C.
Monitoring for Deadlocks in Ada Tasking.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada,* pages IO-25 ACM, Arlington, Virginia, October, 1982.

[5] Good, D.I.
Gypsy Manual.
University of Texas Report, University of Texas, 1980.

[6] Habermann, A.N. and Nassi, I.R.
Efficient Implementation of Ada Tasks
Technical Report CMU-CS-80-103, Carnegie-Mellon University, January, 1980.

[7] Helmbold, D., and Luckham, D.C.
Techniques for Runtime Detection of Deadness Errors in Ada Tasking.
1983.
Report in preparation.

[8] Holden, J., and Wand, I.C.
An assessment of Modula.
Software Practice and Experience 10593-621, 1980.

[9] Ichbiah, J.D., Krieg-Brueckner, B., Wichmann, B.A., Ledgard, H.F., Heliard, J.-C., Abrial, J.-R., Barnes, J.P.G., Woodger, M., Roubine, O., Hilfinger, P.N., Firth, R.
Reference Manual for the Ada Programming Language: Proposed Standard Document.
US Department of Defense, US Government Printing Office, 1980,

[10] Lohr, K-P.
Beyond Concurrent Pascal.
ACM SIGPLAN :173-180, November, 1977.
Proceedings Sixth ACM Symposium on Operating System Principles.

[11] Luckham, D.C., Larsen, H.J., Stevenson, D.R., and von Henke, F.
ADAM -- An Ada based Language for Multi-processing.
Program Verification Group Report PVG-20, CSD Report STAN-CS-81-867, Stanford University, July, 1981.

[12] Luckham, DC., and Falis, E.P.
The Adam Compiler.
1983.
Program Verification Group Report, forthcoming.

[13] Stevenson, D.R.
Algorithms for Translating Ada Multitasking.
ACM SIGPLAN Notices 15(1 I), November, 1980.
Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language.

[14] Wirth, N.
MODULA- 2.
ETH , March, 1980.
Zurich.