67P.

IN-11549

CT393801

A HIGH-ORDER LANGUAGE FOR A SYSTEM OF CLOSELY COUPLED PROCESSING ELEMENTS

Final Report

NASA Grant NAG-3-232

Stefan Feyock and W. Robert Collins

Department of Computer Science

College of William and Mary

Williamsburg, Virginia 23185

July 1986

ABSTRACT

The research reported in this paper was occasioned by the requirements on part of the Real-Time Digital Simulator (RTDS) project under way at NASA Lewis Research Center. The RTDS simulation scheme employs a network of CPUs running lock-step cycles in the parallel computations of jet airplane simulations. Their need for a high order language (HOL) that would allow non-experts to write simulation applications and that could be implemented on a possibly varying network can best be fulfilled by using the programming language Ada*. We describe how the simulation problems can be modeled in Ada, how to map a single, multi-processing Ada program into code for individual processors, regardless of network reconfiguration, and why some Ada language features are particularly well-suited to network simulations.

---

* Ada is a trademark of the Department of Defense

INTRODUCTION

The need for ever more detailed information about systems whose sophistication and complexity is continually growing inevitably places increasingly rigorous demands on the simulation models on which this information depends. The work described in this report was occasioned by the efforts of workers at NASA/Lewis Research Center to develop high-performance computer hardware to support real-time simulation of jet engines, both for the purpose of detailed analysis of system dynamics, and to support the development of digital controls for such propulsion systems [1]. The hardware is structured in the form of a network of communicating microprocessors running in parallel. The need for a higher-order language capability for programming such a network has led to the research described in this report.

HARDWARE CONSIDERATIONS

We will begin by describing the hardware being developed; a more detailed discussion may be found in [2], on which our description is based.

The development of complex digital electronic controls for aircraft propulsion systems requires engine simulations that run in real time and provide a high degree of accuracy and user interaction. In addition, the use of propulsion system simulations in many hardware-in-the-loop applications adds the further requirement that these simulations be implemented on dedicated, portable, and reliable hardware. The advent of microcomputer technology has made compact, low cost, portable computing power readily available. Currently available off-the-shelf microcomputers, however, do not of themselves possess the necessary computational speeds to perform accurate real-time simulations of complex dynamic systems such as aircraft propulsion systems. The approach to this problem adopted by NASA Lewis Research Center in its Real-Time Digital Simulator (RTDS) project is the use of microcomputers in parallel. By using parallel processing it is possible to retain the cost, size, and portability advantages of microcomputers and achieve the accuracy necessary for real-time simulation by increasing the number of computations per unit time.

As work on this project progressed, it became clear that it was not necessary for the program model to reflect low-level details of the computer hardware on which it was to run. By means of progressive abstraction it was possible to create a high-level model that can be effectively mapped to a variety of

hardware configurations, ranging from the lock-step regime originally envisioned to the more sophisticated data-flow architecture that is currently being investigated. To lay the groundwork, we first present the hardware plan as originally conceived, and then indicate how it can be abstracted to obtain a more general model of network computation.

The original structure of the simulator is shown in Figure 1 (from [2]). The core of the system consists of a transfer schema which synchronizes up to 10 16-bit processing elements (PEs) on a high-speed transfer bus. All but two of the PEs perform simulation computations. One of the remaining PEs is of the same architecture but dedicated to input/output functions. The last PE is a special-purpose processor to link low-speed, operator-type functions with the high-speed simulator core. The Front End Processor provides an operator interface as well as handling of peripheral communications and other simulator overhead, such as downloading of programs to the simulator's PEs.

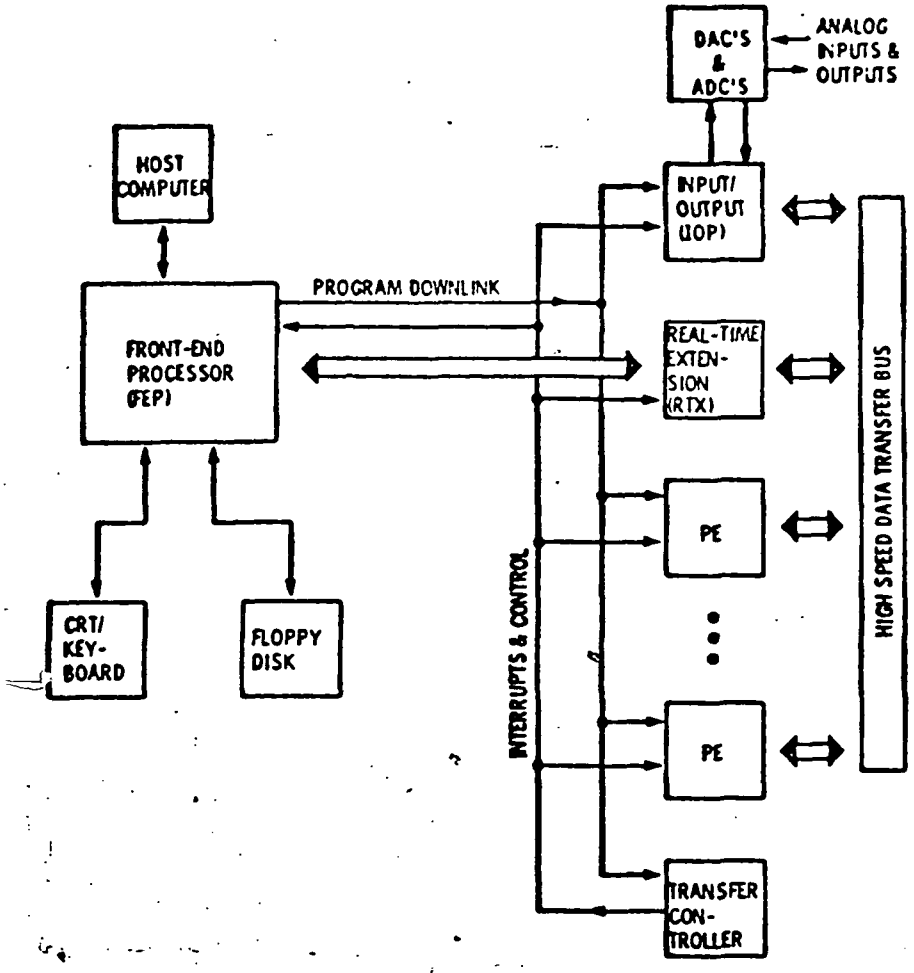

Figure 1

The simulator operation is separated into two basic cycles - a compute cycle and a transfer cycle. During the compute cycle, each PE performs the numerical computations for a pre-defined part of the simulator task. Upon completion of these computations, the PE sets a transfer flag to indicate that it is

ready to enter the transfer cycle. The transfer schema initiates a transfer cycle when all PEs have set their transfer flags. Operator control over the simulator is accomplished via the Front-End Processor and the Real-Time Executive. Such functions as simulator programming, mode control, operator advisories, and commands are provided. The Front-End Processor handles the peripheral communications for the simulator (CRT, keyboard, floppy disk, etc.). There is also a host computer interface which allows uplinking and downlinking of data to and from the host.

The Abstract Model

It is immediately clear that several aspects of this configuration can be generalized; there is no reason that the model should remain specific to, say, ten 16-bit processors. The step to a system of arbitrary processors undergoing a synchronized series of compute and data transfer cycles under the supervision of a transfer schema is not difficult to make. It is less obvious, however, that the transfer schema need not be an actual piece of hardware, but may be virtual: the embodiment within the program model of the data transfer discipline that is in effect. Once this has been realized, it becomes clear that the requirement of lock-step cycles can be relaxed: the program model has been abstracted to a set of modules specifying the code for each processor, and the discipline for transferring data among them. A data flow architecture is thus among the possible instantiations of this model; the data transfer discipline in this case becomes

- begin computation when all required input has arrived;
- transmit data to all specified recipients when computation of this data is complete.

It is important to keep this "virtuality" of the transfer schema in mind during the subsequent discussion.

PROGRAMMING LANGUAGE CONSIDERATIONS

The use of programming languages of an abstraction level higher than that of assembly language is now so widespread both for systems and applications program development that it is difficult to recall how controversial such use was until recent years. The ability of the assembly language programmer to maximize program efficiency by means of direct control of machine operations was deemed more important than the convenience and programming speed gained by use of high-order programming languages (HOLs).

The change in programming practice in recent years leading away from this state of affairs is well known. Hardware costs have dropped drastically, both in absolute terms and with respect to software development costs. Software systems have increased in size and complexity, emphasizing the need for code clarity and maintainability. Finally, the development of integrated microelectronic digital circuitry has led to the widespread use of embedded computer systems in military and aerospace environments that require absolute software reliability.

The result of these developments has been to make the use of HOLs standard practice in an overwhelming number of software

development efforts. The urgency of the requirement for reliable and maintainable code has produced intensive research efforts in the area of programming languages and systems, with the result that modern HOLs not only encourage and facilitate the development of high-quality software while achieving efficiency levels competetive with hand-coded assembly language programs, but can be implemented expeditiously by means of the powerful compiler construction tools that have been developed in recent years. The resulting availability of (cross-)compilers has made programming even quite rudimentary microcomputers in a HOL common practice.

The advent of networks of microcomputers, however, has resulted in a software lag once again. While compilers can be generated for single machines quite rapidly, each configuration of a network is logically equivalent to a different computer, requiring a new compiler to distribute code among the nodes. An additional problem is the dependence of the HOL itself upon the network. Allowing the different microcomputers to communicate among each other is a hardware implementation problem. How the HOL facilitates the generation of efficient code to provide for rapid communication and synchronous behavior is a software problem which is just beginning to be addressed.

RESEARCH OBJECTIVES

Any HOL being considered has to satisfy a host of constraints and requirements necessitated by the general properties of simulation practices and the particular microcomputer network. Some of these requirements are:

1. The HOL must be implementable on any computer or combination of computers. In particular, it is useful to be able to run the simulation on a uniprocessor.
2. The HOL must have the capability for specifying communicating parallel processes.
3. The HOL must support the special requirements of interactive-mode simulations applications.

An evaluation of existing HOLs led to the choice of Ada [3] as best suited to these requirements. A discussion of this evaluation and the considerations influencing this decision is contained in [4]. In the present report we describe

A. A determination of suitable means of mapping the abstract structures of Ada into the hardware configuration.
B. A precompiler that performs this mapping.
C. Advantages of using Ada as the programming vehicle for this project.

PROGRAMMING MODEL REQUIREMENTS

A consideration in the suitability of Ada for the RTDS project is how well the language allows the expression of a good programming model of the underlying physical reality. We imposed several constraints on the programming model itself:

1. The program model must be executable directly on a uniprocessor.
2. The program model must be as simple and natural as possible, since it must be readily programmed by non-experts and should not, therefore, involve complicated synchronization concepts.
3. The program model must be safe, that is, modules contained within should not be able to tamper with or be affected by other modules' data or execution.
4. The program model should be standardized sufficiently in order that it can easily be mapped to the individual programs suitable for the nodes of specific distributed networks.

Any solution to the problem of modeling a simulator network in terms of Ada must fulfill the basic requirements imposed by the application: it must be efficient and it must be independent of the particular structure of the network. Our approach was to tailor the program to reflect the structure of the problem, not of the hardware. Since the hardware itself is presumably

designed with efficient execution of this class of problems in mind, efficiency is a natural consequence of this approach. Our solution fulfills the machine-independence requirement as well: the resulting program can be run equally well on a time-slicing uniprocessor, and, by employing the techniques to be discussed, on the network that is the ultimate target machine.

As indicated, our approach is based on having program structure mirror problem structure as closely as possible. A representative case employs concurrent processes running in parallel to perform the requisite computations, transmit data to each other when done, and then resume. Our Ada model program follows this structure exactly: an independent concurrent program unit corresponds to each independent process of the problem, and these units follow the compute/transfer cycle just outlined.

A central idea of our model was to collect all information pertaining to any one processor into a coherent, self-contained module, allowing a clear and elegant notation for specifying both computation and data transfers. As will be seen, the Ada package concept appears tailor-made for this purpose, and the Ada task concept is a natural implementation of concurrency.

Processes that can execute concurrently are specified in Ada by tasks. The process specified by a task begins execution when the task's declarations are elaborated; in this sense tasks resemble main programs rather than subroutines. Concurrently active tasks can communicate with each other by means of entry calls. An entry of a task is specified by means of an accept statement, which has the (simplified) syntax

```
accept <entry> ( <parameters> ) do
   <statement_sequence>
end;
```

A task T1 can call an entry E in another task T2 by specifying the name of the called task and entry:

```
T2.E;
```

The effect of such a call is to force process synchronization: if T2 has not reached the corresponding

```
accept E;
```

statement, then T1 must queue up until T2 does. If, on the other hand, T2 reaches the

```
accept E;
```

statement before another task has called entry E, T2 must wait until an entry call to this entry occurs. Once either condition is satisfied, a rendesvous takes place: the code specified in <statement_sequence> is executed, with inter-task data transfer occurring via the entry parameters. Upon completion of the

rendesvous the tasks resume independent concurrent operation.

Tasks are usually declared as a two-part entity in Ada programs: the task specification and the task body. The task specification specifies the names of the task's entries and the names and types of the paramenters. It constitutes at once a "forward declaration" and a user interface for the subsequent task body.

The task body, in turn, contains the code specifying the process's activity. Outside entities may in general communicate with this code solely via entries; the task body is closed to them otherwise. Figure 6 gives an example of a task specification, while Figure 7 contains the corresponding body.

Ada Packages

Data/Process Encapsulation.

The prospect of multiple processes running in parallel involves certain problems with respect to data access. In particular, obvious difficulties arise if two processes are allowed to update the same data simultaneously, or if one tries to read data that another is updating. The need to impose discipline on such contention led to the concept of data encapsulation. Data subject to contention is placed inside

programming language constructs that force processes to access the data using a set of strictly circumscribed functions.

Packages are the encapsulation mechanism provided by Ada. Program resources may be collected into a coherent unit by means of this facility, and made available to tasks and subprograms that require access to these resources. It is important to note that the ecapsulated resources may include not only type and data declarations but also subprograms and tasks.

As is the case with other Ada program units, packages are specified in two parts: the package specification and the package body. The package specification contains all the information that is to be accessible ("visible") to the user, in particular the data he may manipulate, and the specifications of subprograms and tasks he may reference. It should be emphasised that for tasks it is only the task entries that are specified in the task specification part, which in turn is the only part of the task that is present in the package specification. Figure 4 illustrates a package specification.

A package body contains all the machinery needed to implement the subprograms and tasks whose specifications are to be found in the package specification: the subprogram and task bodies, as well as any variables and types required by this machinery. Constructs within a package body are in general invisible to the

user, who may access only what has been made available to him in the package specification. Figure 5 depicts a package body containing the task body for CODE; it also illustrates the mechanism for making a package available to a pro gram unit: the with statement. In this case it is the package TRANSFER_SCHEMA that is made available to package body FAN_INLET.

THE ADA MODEL

The Ada model combines the two distinct Ada constructs, tasks and packages, for the two programming requirements of concurrency and efficient data transfer. The code for each of the hardware processing elements is specified by an Ada task, which we call the hardware task pertaining to that processor. Using packages and visibility commands, the flow of data between concurrent processes can be specified and controlled by a single process, called the transfer schema. Consequently, if the transfer schema is designed and programmed correctly, then all communications are correct.

As indicated above, the best way to model the processing elements is to use a single package for each processing element. The package body (normally invisible to other programming modules) contains the hardware task which corresponds to the code to be executed on the processing element. The package

specification (or visible part) contains all the variables needed for import/export and the task entries needed for synchronization. The major benefit of this standardization and data hiding is that the conversion of the model to a program suitable for a network is made tractable.

MAPPING THE MODEL TO THE HARDWARE

Many of the advantages of using a suitable HOL in distributed programming will be lost unless a good way is found to map the programming model constructed in the HOL to the individual nodes in the hardware network. There does not exist any compiler that will translate abstract programming models into code for any RTDS network. Such a compiler would be expensive to construct and would have limited utility, for any change to the network would necessitate major changes in the compiler. If, however, a single program (or compilation) is written for a network and a series of programs, one for each node in the network, is desired, then a solution is to convert the program text for the whole network into a series of individual program texts suitable for each processor. At that point a standard compiler for the HOL for the individual processor may be employed to derive code for the processor. The conversion from a single text to multiple texts is accomplished by a program called a precompiler.

The elegance, utility, and power of the Ada model synergistically coupled with especially useful Ada constructs argue convincingly in favor of a precompiler with Ada source and target texts as the best solution to the HOL-network problem. The expected proliferation of Ada compilers also makes the Ada-to-Ada precompiler solution attractive, obviating the construction of code generators for each kind of target computer. There will be more Ada compilers available for different processors than for any other real-time language. The Ada language itself is particularly well-suited to the precompiler solution. One of Ada's useful features in bare-computer, real-time computing is the representation specification. The programmer is allowed to insert machine dependencies into Ada code; for example, he may specify the absolute address of variables or insert assembly language code. The ability to reach through the HOL virtual computer to the actual hardware is generally considered harmful because of potential programmer abuse. However, applications programmers will not be employing these representation specifications; the precompiler will use them to convert rendezvous code and other machine-dependent code into the code necessary to effect bus communications. Bus communication usually involves knowing absolute addresses and manipulating bits, both of which are difficult or impossible in most HOLs. However, the precompiler will have no trouble inserting such code, and will still produce an Ada program rather than an assembly language program.

A second feature of Ada well-suited to the precompiler solution is the pragma, or compiler directive. Programmers may use pragmas almost anywhere in Ada text for almost any purpose. Some pragmas are built in the language, for example, the pragma OPTIMIZE, which takes one of two parameters, TIME or SPACE. Other pragmas are allowed by particular implementations. If an implementation does not recognize a pragma, the pragma is ignored. We intend that the Ada program model contain pragmas (for example, CODE_MAP) meant for the precompiler to aid the precompiler in its execution. These same pragmas will have no effect when compiled by a uniprocessor compiler, thus allowing the exact same text to work on a uniprocessor directly (with simulated parallelism) or on a network after precompiling.

THE OPERATION OF THE PRECOMPILER

The precompiler was generated from a LALR(1) grammar for Ada by the PARGEN parser generator component of the Mystro Translator Writing System [5] developed at the College of William and Mary. It employs two passes to delineate precisely which variables are intended for transfer, which variables must be placed in absolute memory locations, which constructs correspond to the hardware tasks, and so on. Its final pass produces a series of text files corresponding to uniprocessor Ada programs.

The precompiler operates on two assumptions. The first is that the coding conventions dictated by the programming model are followed. For example, each separate processing element must appear in a distinct package, the first task in that package is the code for the element, all interprocess communication is done via calls to the transfer controller package, etc. These conventions are tailored to the problem to be solved. Changes to the conventions may necessitate changes to the precompiler. The precompiler can therefore only be used in simulations which conform to the programming model. This is not unduly restrictive, since the programming model is general enough to encompass a large class of simulations.

The second major assumption is that all processing elements must synchronize after each computation cycle. This synchronicity is exploited to simplify the structure of the transfer controller package and the loops in the resulting single processor code.

The precompiler splits a multitasking program which satisfies the programming model into a set of single-processor programs. The two conceptual steps the translator must perform are:

Determine the names of packages that represent processing elements and the transfer controller.

For each processor package that represents a processing element, create a procedure to run on a separate processor. This procedure is formed from information obtained from the original processor package and the transfer controller.

The collection of separate programs (Ada procedures) produced by the precompiler must be functionally equivalent to the original multitasking program. As has been described above, the original package used to represent a processing element communicates its values to other packages via a package called the transfer controller. After splitting, communication must be accomplished via a bus. The transfer logic resident in the transfer controller must thus be distributed to the split procedures. This is accomplished by the precompiler replacing waits for the transfer controller by calls to a bus package, followed by a wait in a busy loop. These calls explicitly pass or receive the values to be transferred and the destination address.

PRECOMPILER EXAMPLES

Details of how these steps are performed are given in a subsequent section. We first illustrate these steps for two sample processing element packages A and B, and a transfer controller package called TRANSFER_CONTROLLER. These packages

are identified to the precompiler via the pragma compiler directive. We then show the effect of the precompiler on the fan inlet example of Figures 4, 5, 6, and 7.

Here is the original Ada program. This program will run correctly on a uniprocessor, or can be processed by the precompiler to produce the split procedures shown below.

```
pragma code_map(internal => A, actual => "CPU_A");
pragma code_map(internal => B, actual => "CPU_B");
-- the above pragmas tell the precompiler which package
-- ("hardware task") will be mapped to which actual machine

pragma transfer(TRANSFER_CONTROLLER);
-- This pragma tells the transfer controller that the data
-- transfers are specified in the package named TRANSFER_CONTROLLER

package A is
  x, y: integer := 1; -- moved to split procedure

  task A_code is
     entry START_UP; -- replaced by precompiler
     entry RESUME;   -- replaced by precompiler
  end A_CODE;
end A;

package body A is

  task body A_CODE is
  begin
     accept START_UP; -- replaced by precompiler
     loop
       x := x + y; -- or any arbitrary computation
       TRANSFER_CONTROLLER.SIGNAL; -- signal completion
       accept RESUME; -- replaced by precompiler
     end loop;
  end A_CODE;
end A;
```

Figure 2.a

```
package B is
  x, y: integer := 1; -- moved to split procedure

  task B_code is
     entry START_UP; -- replaced by precompiler
     entry RESUME;   -- replaced by precompiler
  end B_CODE;
end B;

package body B is

  task body B_CODE is
  begin
     accept START_UP; -- replaced by precompiler
     loop
       x := x + y; -- or any arbitrary computation
       TRANSFER_CONTROLLER.SIGNAL; -- signal completion
       accept RESUME; -- replaced by precompiler
     end loop;
  end B_CODE;
end B;
```

Figure 2.b

```
task TRANSFER_CONTROLLER is
  entry SIGNAL;
end  TRANSFER_CONTROLLER;

task body TRANSFER_CONTROLLER is
  No_of_processors: constant = 2;
  Signal_count: integer range 0 .. No_of_processors;

begin
  -- start up both processes:
  A.START_UP;
  B.START_UP.

  loop
     Signal_count := No_of_processors;
     while Signal_count > 0 loop
        accept SIGNAL;
        Signal_count := Signal_count - 1;
     end loop; -- busy wait for everybody to finish

     A.y := B.x; -- moved to split procedure
     B.y := A.x; -- moved to split procedure
     A_CODE.RESUME;
     B_CODE.RESUME;
  end loop;
end TRANSFER_CONTROLLER;
```

Figure 2.c

The packages shown in Figures 2.a, b, and c will run perfectly well on a uniprocessor, simulating concurrency and allowing the programs in question to be debugged. When desired, they can be mapped by the precompiler to Ada code that will run on separate machines, communicating via a hardware bus. The precompiler produces as output the following Ada programs:

```
with BUS; use BUS;
procedure A is
  x, y: integer := 1; -- moved from original package

begin
  -- the following loop is created and inserted by
  -- the precompiler
  loop
     exit when INPUT_READY;
     -- busy loop, waiting for signal
     -- corresponds to accept START_UP in original
  end loop;

  loop
     MOVE(TO => y, FROM => x_LOC);
     -- MOVE is a bus package procedure. This call is
     -- created and inserted by the precompiler

     x := x + y;

     -- TRANSFER is a bus package procedure. This call is
     -- created and inserted by the precompiler
     TRANSFER(VALUE => x, SEND_TO => B, ADDRESS => y_LOC);

     -- the following loop is created and inserted by
     -- the precompiler
     loop
        exit when INPUT_READY;
        -- busy loop, waiting for signal
        -- corresponds to accept RESUME in original
     end loop;
  end loop;
end A;
```

Figure 3.a

The procedure for B is similar:

```
procedure B is
  x, y: integer := 1;

begin
  loop
     exit when INPUT_READY;
     -- busy loop, waiting for signal
     -- corresponds to accept START_UP in original
  end loop;
  loop
     MOVE(TO => y, FROM => x_LOC);
     x := x + y;
     loop
        exit when INPUT_READY;
        -- busy loop, waiting for signal
        -- corresponds to accept RESUME in original
     end loop;
     TRANSFER(VALUE => x, SEND_TO => A, ADDRESS => y_LOC);
  end loop;

end B;
```

Figure 3.b

A JET ENGINE SIMULATION EXAMPLE

We now give a more realistic example, representing a portion of an actual jet engine simulation. Suppose that the code for the FAN_INLET computations of a jet engine simulation is to be assigned to hardware processor 1. This assignment is specified by means of the pragma shown in Figure 4. The code depicted there corresponds to the visible part of the FAN_INLET routine. The entries START_UP and RESUME are needed for synchronization. When either is called (like a subroutine), the execution of the code for FAN_INLET can start or resume. Each of these package specifications can and should be compiled separately.

```
pragma CODE_MAP(INTERNAL => FAN_INLET,
                ACTUAL   => "processor 1");
   -- Informs the precompiler that
   -- code for FAN_INLET will be
   -- on CPU node processor 1

pragma transfer(TRANSFER_SCHEMA);

package FAN_INLET is
   -- Here are the declarations of
   -- the transfer variables.
   -- They will need addresses for
   -- bus transfer and the data base:
   A, B, C : VECTOR;

   -- Here is the task specification
   -- with synchronization entries:

   task CODE is
      entry START_UP;
      entry RESUME;
   end CODE;

end FAN_INLET;
```

Figure 4

The Ada compilation unit which contains the code for FAN_INLET is given in Figure 5. The with statement is a directive to the compiler that this package body should be compiled with the specification of the transfer schema task. This is necessary since entry SIGNAL of the transfer schema is called. The body of the package consists of the task body only. The task body contains three rendezvous which are the Ada constructs used for communications between tasks.

```
with TRANSFER_SCHEMA;
package body FAN_INLET is
   -- Here is the body of the task:

   task body CODE is

      -- Here are local declarations
      -- not involved with data transfer.
      -- These will need  addresses:
      TEMP : VECTOR;

   begin
      accept START_UP;
      loop
         TEMP := A;
         A    := A + B;
         B    := TEMP - C;
         TRANSFER_SCHEMA.SIGNAL;
         accept RESUME;
      end loop;
   end CODE;
end FAN_INLET;
```

Figure 5

The text for CODE has these semantics: Task CODE is suspended until it receives a call (from the transfer schema) to the entry START_UP. The task then enters an infinite loop which consists of its calculations, a call to an entry of the transfer schema indicating that its calculations are done and its export variables are ready for export, and suspension until it receives a call (from the transfer schema) to the entry RESUME indicating that the variables necessary for the next cycle have been imported.

As can be seen from the models for the hardware processing elements, a critical cog in the overall model is the transfer

schema task. Its specification, given in Figure 6, must be compiled with the task bodies described in Figure 5. The body of TRANSFER_SCHEMA, given in Figure 7, must be compiled with the package specifications corresponding to the processing elements since the transfer schema task must be aware of the import/export variables and the synchronization entries.

```
task TRANSFER_SCHEMA is
   entry SIGNAL;
end TRANSFER_SCHEMA;
```

Figure 6

The body of the transfer schema contains two local declarations: a constant TOTAL indicating the total number of processing elements to be synchronized and a counter variable COUNT to tell when all the processing elements have completed their calculations.

```
with FAN_INLET;
with REAR_DUCT;
with FORWARD_SENSOR;

task body TRANSFER_SCHEMA is

   No_of_processors : constant := 3;
   Signal_count : INTEGER range 0..No_of_processors;

begin
   -- start up all three processes:
   FAN_INLET.CODE.START_UP;
   REAR_DUCT.CODE.START_UP;
   FORWARD_SENSOR.CODE.START_UP;

   loop
      Signal_count := No_of_processors;
      while Signal_count > 0 loop
         accept SIGNAL;
         Signal_count := Signal_count - 1;
      end loop; -- busy wait for everybody to finish

      FORWARD_SENSOR.W := FAN_INLET.A;
      REAR_DUCT.X      := FAN_INLET.C;

      FAN_INLET.CODE.RESUME;
      REAR_DUCT.CODE.RESUME;
      FORWARD_SENSOR.CODE.RESUME;

   end loop;
end TRANSFER_SCHEMA;
```

Figure 7

The code for the transfer schema has these semantics: all the hardware tasks are started by calls to the START_UP entry in each hardware task. Then the transfer schema enters an infinite loop in which it awaits entry calls from the hardware tasks indicating that they have finished their computations. The "accept SIGNAL" in the transfer schema is matched with the "TRANSFER_SCHEMA.SIGNAL" entry calls in the tasks for rendezvous.

After all the tasks have signaled completion, the transfer schema transfers the variables.

FORWARD_SENSOR.W := FAN_INLET.A

means that the value of variable A in FAN_INLET is to be stored in the location of the variable W in FORWARD_SENSOR. In a uniprocessor, this is a straightforward assignment. In a network, the assignment will be converted to instructions (calls to a bus handler package) to allow the value of A to be communicated by the bus to the location of W. After the variables have been transferred, the transfer schema signals each hardware task to resume execution by calling the RESUME entry of the task. Recall that the tasks have been suspended while the variables were transfered because of the "accept RESUME" statements. This completes the cycle of execution in the transfer schema.

The Ada program model for a processing element in Figures 4 and 5 will be converted by the precompiler to the main program given in Figure 8. The two busy loops are broken either by interrupts or a switched bit (depending on the nature of the bus communications) to synchronize the startup and the import of data. The system library function INPUT_READY may be coded independently of the precompiler to accomodate changes in the network configuration or basic design. The system library procedures MOVE and TRANSFER control the moving of data from the

bus depot to their memory locations and the moving of data from memory to the bus depot and then through the bus itself. The code for these system library routines may be high-level Ada code, assembly language, a call to a hardware procedure, or a combination of these that moves the export variables to the bus depot and signals that the import variables have all arrived. The three routines are located in the package BUS, and may be named directly because of the "with" and "use" clauses preceeding the main program FAN_INLET. The rest of the code mimics that of the original hardware task.

```
with BUS; use BUS;
procedure FAN_INLET is
   A, B, C : VECTOR;
   for A use at 16#A0#;
   for B use at 16#A8#;
   for C use at 16#B0#;
   for TEMP use at 16#B8#;
   -- 16# indicates that the
   -- addresses are hexadecimal

begin
   -- the following loop is created and inserted by
   -- the precompiler
   loop
      exit when INPUT_READY;
      -- Busy loop, waiting for signal
      -- that input arrived at depot.
      -- Corresponds to START_UP.
   end loop;

   loop

      -- Move variables from bus depot
      -- to their memory locations.
      MOVE(TO => A, FROM => A_LOC);
      MOVE(TO => B, FROM => B_LOC);
      MOVE(TO => C, FROM => C_LOC);

      TEMP := A;
      A    := A + B;
      B    := TEMP - C;

      -- The value of A will be sent
      -- to FORWARD_SENSOR to be
      -- stored in the bus depot
      -- for variable W there.
      TRANSFER(VALUE   => A,
               SEND_TO => FORWARD_SENSOR,
               ADDRESS => W_LOC);
      TRANSFER(VALUE   => B,
               SEND_TO => REAR_DUCT,
               ADDRESS => X_LOC);
```

```
      -- the following loop is created and inserted by
      -- the precompiler
         loop
            exit when INPUT_READY;
            -- Corresponds to RESUME in original
         end loop;
      end loop;
   end FAN_INLET;
```

Figure 8

PRECOMPILER CONSTRUCTION TOOLS

The MYSTRO translator writing system [5] was used to implement the precompiler. Many of the problems encountered in constructing compilers or, in this case, a precompiler, admit the same solutions regardless of the specific language being translated. MYSTRO employs several skeleton compilers appropriate to most programming languages. Except for minor, clearly-marked areas, any skeleton's code can be used to produce a complete listing, read lines for parsing, produce symbolic cross-references, and so on. The particular skeleton chosen for this project also includes hashing routines and multi-level error recovery.

The initial precompiler was generated by the MYSTRO parser generator PARGEN, which computed and merged parse tables for a complete Ada grammar into the skeleton compiler. Pascal semantics were included in the input grammar, and automatically inserted into the SYNTHESIZE procedure, which associates semantics with the appropriate syntax.

OPERATION OF THE PRECOMPILER

In order to split the original multiprocessing input program into separate uniprocessing programs that will run on the nodes of the network, the precompiler makes two passes: an information-gathering first pass, and an output second pass. While gathering information, the precompiler must know which packages represent processing elements and mark sections of their code. It does this by creating, as part of its semantics for the CODE_MAP pragma, a list of the packages that represent processing elements. Each element of this list holds information needed to split the program into the intended separate programs.

Once a processing element package specification is found, the location of the start of the specification is noted in that package's descriptor. The first task specification encountered after processing the package specification designator is marked in the descriptor and designates the end of information needed from the package specification. At this point the precompiler also records in the descriptor the names of all the entries declared within the nested task specification.

When the body of a processing element package is found, the package descriptor is stacked to allow for package nesting, thus preventing erroneous location information. The task body's

location inside the package body is recorded in that package's descriptor. This task body corresponds to the nested task specification found in the package specification. Inside this task body, the loop and end loop for the outermost loop are both recorded in the descriptor to allow for the transfer of bus variables in and out of the simulated processor. Throughout the task body, entry names found in accept statements are compared with the entry list within the package's descriptor. The locations of those that match are recorded and the rest ignored. These accept statements will be converted to busy loops in the rewriting phase of the precompiler. The end of the package body is also recorded as the end of the information needed to complete this processing element package.

Information regarding the bus variables and the synchronizing entries must also be gathered during this first pass; they are found inside the specification and body of the transfer controller. Several lists are created during the first pass: entries declared within the transfer controller's specification, variables to be moved into each processor at each loop iteration within the processor, and variables to be transferred to the bus depot for use in another processor at the end of each loop iteration.

The information-gathering first pass is by far the more complex of the two passes. It is a straightforward matter to

separate the file containing the input program into several files containing processing element programs.

The complexity of the first pass is mitigated by the fact that the precompiler is syntax-directed. The Ada grammar consists of nearly five hundred rules, only a small portion of which affect the precompiler's task. Each rule is like a small program; the programmer need only concern himself with developing correct semantics for that rule and passing information through the semantics stack to other rules. For example, the rule

<pragma> ::= pragma <identifier>

can be used to associate with CODE_MAP semantics that enquire about the identifier. In fact, the SYNTHESIZE procedure contains the following case:

```
(* <pragma> ::= pragma <identifier> *)
if <identifier>.id = 'CODE_MAP      ' then
  <pragma>.flag := true
else
  <pragma>.flag := false;
```

MYSTRO contains utilities to translate notation such as <identifier>.id into the appropriate stack references.

In addition to representation specifications and pragmas, Ada has a variety of programming features especially suited to interactive-mode simulation applications. Some of these are described below.

Safety in the Multi-Programming Mode. Ada encourages two of the main software engineering techniques to facilitate the rapid construction of reliable software for large and complex software projects. These two techniques, data encapsulation and safe separate compilation, are employed in the packages that mimic network nodes. The package body (normally invisible to other programming modules) contains the hardware task which corresponds to the code to be executed on the processing element. The package specification (or visible part) contains all the variables needed for import/export and the task entries needed for synchronization. Finally, use of Ada separate compilation facilities guarantees that processing elements cannot communicate directly with each other, that is, a programmer cannot make use of the "innards" of one processing element when describing the behavior of another. This frees the programmer of the responsibility of effecting the bus communications directly and also allows the Ada programs to run on uniprocessors without any change in code. Such orthogonality

allows programmers and engineers to concentrate on individual processing element correctness and efficiency without worrying about ripple effects on the other processing elements.

Abstract Data Types. Ada's abstract data type capability diminishes the distance between the programming model and the original simulation applications. Through the generic and package constructs, new data types specific to the application can be created together with the operations necessary to manipulate these types. These operations are allowed to have standard forms such as +, -, <, and so on. For example, in a package specification we may create a type VECTOR together with plus operations (all denoted by +) for various combinations or scalar and vector addition. It is expected that many packages particularly suited to real-time simulation applications will be constructed and sold by commercial vendors (perhaps in Ada Package Stores). Consequently program systems may be partially built with off-the-shelf components instead of being hand-crafted each time.

Real-Time Constructs. Ada has a variety of real-time features which allow real-time constraints to be employed in simulation applications. These include the ability to deactivate a task for a specified period of time, as well as wait a specified time before aborting a prospective rendezvous. Moreover, a predeclared package CALENDAR allows arithmetic on

wall-clock times and durations, as well as access to the system clock. One specific application is to monitor lock-step compute-data cycles.

CONCLUSION

The concept of implementing a higher-order language on a computer network by means of a precompiler has proven to be extremely fruitful. Not only was it possible to map programs for the original lock-step network design onto the hardware, but it now appears feasible to apply this technique to more general network designs. Moreover, many of the system facilities required for interactive-mode simulation can be implemented by means of precompilation. Our research has demonstrated the usefulness of this approach both on the original hardware design and on networks of more general structure.

REFERENCES

1. Krosel, Susan M. and Milner, Edward J., "Application of Integration Algorithms in a Parallel Processing Environment for the Simulation of Jet Engines", Proceedings of the 15th Annual Simulation Symposium, Tampa, March 1982, pp. 121-144.

2. Blech, Richard A. and Arpasi, Dale J., "An Approach to Real-Time Simulation using Parallel Processing", NASA Lewis Research Center, Cleveland, Ohio.

3. Ichbiah, Jean et al., Reference Manual for the Ada Programming Language, United States Department of Defense, July, 1980.

4. Feyock, Stefan and Collins, W. Robert, "Ada and Multi-processor Real-time Simulation", Proceedings of the 16th Annual Simulation Symposium, Tampa, March 1983.

5. Collins, W. Robert and Noonan, Robert E., The Mystro Parser Generator User's Manual, Version 6.3, College of William and Mary, Williamsburg, Virginia, October 1982.

APPENDIX

EXAMPLE PRECOMPILER RUNS

ORIGINAL PROGRAM INPUT TO PRECOMPILER

The following program is the result of translating a sample FORTRAN simulation program furnished by NASA/Lewis into Ada. As can be seen, the format of this Ada program conforms to the Ada model described in the report. It consists of tasks A, B, C, D, and IOP, and a TRANSFER_CONTROLLER to move data among them. This program will run on any machine with a full Ada compiler. It was processed by the precompiler, which split it into separate procedures intended to run on the nodes of a network.

College of William and Mary Source Listing 27/11/84 14:36:48

```
Line#    Source Line

    1    : pragma code_map (internal => A, actual => "cpu_A");
    2    : pragma code_map (actual => "cpu_B", internal => B);
    3    : pragma code_map ("cpu_C", internal => C);
    4    : pragma code_map (internal => D,"cpu_D");
    5    : pragma code_map (internal => IOP, actual => "IOP_thing");
    6    :
    7    : pragma transfer (TRANSFER_CONTROLLER);
    8    :
    9    : package VECTORS is
   10    :    type COORDINATE is (X, Y);
   11    :    type VECTOR is array (COORDINATE) of FLOAT;
   12    :    ORIGIN : constant VECTOR := (X => 0.0, Y => 0.0);
   13    :
   14    :    function "+" (C, D : VECTOR) return VECTOR;
   15    :    function "-" (C, D : VECTOR) return VECTOR;
   16    :    function "*" (C : FLOAT; D : VECTOR) return VECTOR;
   17    :
   18    : end VECTORS;
   19    :
   20    :
   21    :
   22    : package body VECTORS is
   23    :
   24    :    function "+" (C, D : VECTOR) return VECTOR is
   25    :    begin
   26    :       return (X => C(X) + D(X), Y => C(Y) + D(Y));
   27    :    end "+";
   28    :
   29    :    function "-" (C, D : VECTOR) return VECTOR is
   30    :    begin
   31    :       return (X => C(X) - D(X), Y => C(Y) - D(Y));
   32    :    end "-";
   33    :
   34    :    function "*" (C : FLOAT; D : VECTOR) return VECTOR is
   35    :    begin
   36    :       return (X => C * D(X), Y => C * D(Y));
   37    :    end "*";
   38    :
```

College of William and Mary Source Listing 27/11/84 14:36:48

```
Lines    Source Line

   39    : end VECTORS;
   40    :
   41    :
   42    :
   43    : with VECTORS; use VECTORS;
   44    : package GLOBAL is
   45    :    A  : FLOAT   := 1.00;
   46    :    B  : FLOAT   := 1.00;
   47    :    DT : FLOAT   := 0.02;
   48    :    T  : FLOAT   := 0.04;
   49    :    L  : FLOAT   := 10.00;
   50    :    N  : INTEGER := INTEGER(L/DT);
   51    :    UN : FLOAT   := 1.00;
   52    :
   53    :    function DERIVATIVE (X : VECTOR) return VECTOR;
   54    : end GLOBAL;
   55    :
   56    :
   57    :
   58    : package body GLOBAL is
   59    :
   60    :    function DERIVATIVE (C : VECTOR) return VECTOR is
   61    :    begin
   62    :       return (X => C(Y), Y => (1.0/A) * (UN - C(X)) - (B/A) * C(Y) );
   63    :    end DERIVATIVE;
   64    : end GLOBAL;
   65    :
   66    :
   67    :
   68    : with VECTORS; use VECTORS;
   69    : with GLOBAL;  use GLOBAL;
   70    : with TEXT_IO;
   71    : procedure RUNGE_INTEGRATION_SCHEME is
   72    :
   73    :    package A is
   74    :       XMP2, XPDM    : VECTOR;
   75    :       XPMX2, XPDMX2 : VECTOR := ORIGIN;
   76    :
```

```
Lines    Source Line

   77    :        task A_CODE is
   78    :           entry START_UP;
   79    :           entry RESUME;
   80    :        end A_CODE;
   81    :     end A;
   82    :
   83    :     package B is
   84    :        XNP2, XPON, XPONP : VECTOR;
   85    :        XPNX, XPONX       : VECTOR := ORIGIN;
   86    :
   87    :        task B_CODE is
   88    :           entry START_UP;
   89    :           entry RESUME;
   90    :        end B_CODE;
   91    :     end B;
   92    :
   93    :     package C is
   94    :        XNP3, XPCN, XPCNP : VECTOR;
   95    :        XCN, XN           : VECTOR := ORIGIN;
   96    :
   97    :        task C_CODE is
   98    :           entry START_UP;
   99    :           entry RESUME;
  100    :        end C_CODE;
  101    :     end C;
  102    :
  103    :     package D is
  104    :        XDPN2, XNPB : VECTOR;
  105    :        XNP         : VECTOR := ORIGIN;
  106    :
  107    :        task D_CODE is
  108    :           entry START_UP;
  109    :           entry RESUME;
  110    :        end D_CODE;
  111    :     end D;
  112    :
  113    :     package IOP is
  114    :        XPNX,
```

```
Lines    Source Line

  115    :        XPNX2 : VECTOR;
  116    :
  117    :        task IOP_CODE is
  118    :           entry START_UP;
  119    :           entry RESUME;
  120    :        end IOP_CODE;
  121    :     end IOP;
  122    :
  123    :     task TRANSFER_CONTROLLER is
  124    :        entry SIGNAL;
  125    :     end TRANSFER_CONTROLLER;
  126    :
  127    :
  128    :
  129    :     package body A is
  130    :
  131    :        task body A_CODE is
  132    :        begin
  133    :           accept START_UP;
  134    :           loop
  135    :              XPNX2  := XNP2 + 4.0*H*XPDN;
  136    :              XPDNX2 := DERIVATIVE(XPNX2);
  137    :              TRANSFER_CONTROLLER.SIGNAL;
  138    :              accept RESUME;
  139    :           end loop;
  140    :        end A_CODE;
  141    :     end A;
  142    :
  143    :
  144    :     package body B is
  145    :
  146    :        task body B_CODE is
  147    :        begin
  148    :           accept START_UP;
  149    :           loop
  150    :              XPNX  := XNP1 + 1.5*H*(XPDN + XPDNP);
  151    :              XPDNX := DERIVATIVE(XPNX);
  152    :              TRANSFER_CONTROLLER.SIGNAL;
```

```
Lines   Source Line

  153   :                accept RESUME;
  154   :             end loop;
  155   :          end B_CODE;
  156   :       end B;
  157   :
  158   :
  159   :       package body C is
  160   :
  161   :          task body C_CODE is
  162   :          begin
  163   :             accept START_UP;
  164   :             loop
  165   :                XN  := XNP3 - 1.5*H*(XPDN - 3.0*XPDNP);
  166   :                XDN := DERIVATIVE(XN);
  167   :                TRANSFER_CONTROLLER.SIGNAL;
  168   :                accept RESUME;
  169   :             end loop;
  170   :          end C_CODE;
  171   :       end C;
  172   :
  173   :
  174   :       package body D is
  175   :
  176   :          task body D_CODE is
  177   :          begin
  178   :             accept START_UP;
  179   :             loop
  180   :                XNP := XNP3 + 2.0 * H * XDNP2;
  181   :                XDNP := DERIVATIVE(XNP);
  182   :                TRANSFER_CONTROLLER.SIGNAL;
  183   :                accept RESUME;
  184   :             end loop;
  185   :          end D_CODE;
  186   :       end D;
  187   :
  188   :
  189   :       package body IOP is
  190   :
```

```
Lines#    Source Line

  191   :        task body IOP_CODE is
  192   :           use TEXT_IO;
  193   :           package INT_IO is new INTEGER_IO(INTEGER);
  194   :           package REAL_IO is new FLOAT_IO(FLOAT);
  195   :           use INT_IO, REAL_IO;
  196   :
  197   :           S1  : INTEGER := 1;
  198   :           TIN : FLOAT := 0.08;
  199   :           TV  : array (1..500) of FLOAT;
  200   :           XV  : array (1..500) of FLOAT;
  201   :
  202   :        begin
  203   :           accept RESUME;
  204   :           TV(1) := TIN;
  205   :           TRANSFER_CONTROLLER.SIGNAL;
  206   :           for I in 1..N loop
  207   :              accept RESUME;
  208   :              XV(S1    ) := FIRST(XPNX);
  209   :              XV(S1 + 1) := FIRST(XPNX2);
  210   :              TRANSFER_CONTROLLER.SIGNAL;
  211   :              if NT > 1 then
  212   :                 TV(S1) := TV(S1 - 1) + H;
  213   :              end if;
  214   :              TV(S1 + 1) := TV(S1) + H;
  215   :              S1         := S1 + 2;
  216   :           end loop;
  217   :           PUT(N);
  218   :           PUT_LINE;
  219   :           for I in 1..N + 1 loop
  220   :              PUT(TV(I));
  221   :              PUT(XV(I));
  222   :              PUT_LINE;
  223   :           end loop;
  224   :        end IOP_CODE;
  225   :     end IOP;
  226   :
  227   :
  228   :     task body TRANSFER_CONTROLLER is
```

```
Lines    Source Line

  229    :        use A, B, C, D, IOP;
  230    :        NO_OF_PROCESSORS : constant := 5;
  231    :        SIGNAL_COUNT : INTEGER range 0..NO_OF_PROCESSORS;
  232    :    begin
  233    :        for NT in 1..N loop
  234    :
  235    :            A.XNP2     := C.XN;
  236    :            B.XNP2     := C.XN;
  237    :            C.XNP3     := D.XNP;
  238    :            D.XNP3     := D.XNP;
  239    :            D.XDNP2    := C.XDN;
  240    :            B.XPDNP    := B.XPDNX;
  241    :            C.XPDNP    := B.XPDNX;
  242    :            A.XPDN     := A.XPDNX2;
  243    :            B.XPDN     := A.XPDNX2;
  244    :            C.XPDN     := A.XPDNX2;
  245    :            IOP.XPNX   := B.XPNX;
  246    :            IOP.XPNX2  := A.XPNX2;
  247    :
  248    :            A_CODE.RESUME;
  249    :            B_CODE.RESUME;
  250    :            C_CODE.RESUME;
  251    :            D_CODE.RESUME;
  252    :            IOP_CODE.RESUME;
  253    :
  254    :            SIGNAL_COUNT := NO_OF_PROCESSORS;
  255    :            while SIGNAL_COUNT > 0 loop
  256    :               accept SIGNAL;
  257    :               SIGNAL_COUNT := SIGNAL_COUNT - 1;
  258    :            end loop;
  259    :
  260    :        end loop;
  261    :    end TRANSFER_CONTROLLER;
  262    :
  263    : begin  -- dummy main procedure
  264    :    null;
  265    : end;
```

OUTPUT OF THE PRECOMPILER

The following Ada procedures A, B, C, D, and IOP were produced as output by the precompiler processing the previous program. The intent is that each of these procedures be assigned to a processor of the network, as specified by the pragmas of the original program. Note that the the precompiler has replaced the data transfers specified in TRANSFER_CONTROLLER by calls to the MOVE and TRANSFER entries of the bus package.

```
procedure A is
```

```
procedure A is
   XNP2, XPDN     : VECTOR;
   XPNX2, XPDNX2 : VECTOR := ORIGIN;

   begin
      loop
         exit when INPUT_READY;
         --Busy loop, waiting for signal
         --corresponds to START_UP
      end loop;
      loop
         MOVE(TO => XPDN, FROM => XPDN_LOC );
         MOVE(TO => XNP2, FROM => XNP2_LOC );
         XPNX2  := XNP2 + 4.0*XPDN;
         XPDNX2 := DERIVATIVE(XPNX2);
         TRANSFER(VALUE   => XPNX2,
                  SEND_TO => IOP,
                  ADDRESS => XPNX2_LOC );
         TRANSFER(VALUE   => XPDNX2,
                  SEND_TO => C,
                  ADDRESS => XPDN_LOC );
         TRANSFER(VALUE   => XPDNX2,
                  SEND_TO => B,
                  ADDRESS => XPDN_LOC );
         TRANSFER(VALUE   => XPDNX2,
                  SEND_TO => A,
                  ADDRESS => XPDN_LOC );
      loop
         exit when INPUT_READY;
         --Busy loop, waiting for signal
         --corresponds to RESUME
      end loop;
      end loop;
end A;
```

```
procedure 2 is


procedure 3 is
   XNP2, XPDN, XPDNP : VECTOR;
   XPNX, XPDNX       : VECTOR := ORIGIN;

   begin
      loop
         exit when INPUT_READY;
         --busy loop, waiting for signal
         --corresponds to START_UP
      end loop;
      loop
         MOVE(TO => XPDN, FROM => XPDN_LOC );
         MOVE(TO => XPDNP, FROM => XPDNP_LOC );
         MOVE(TO => XNP2, FROM => XNP2_LOC );
         XPNX  := XNP2 + 1.5*H*(XPDN + XPDNP);
         XPDNX := DERIVATIVE(XPNX);
         TRANSFER(VALUE   => XPNX,
                  SEND_TO => IOP,
                  ADDRESS => XPNX_LOC );
         TRANSFER(VALUE   => XPDNX,
                  SEND_TO => 0,
                  ADDRESS => XPDNP_LOC );
         TRANSFER(VALUE   => XPDNX,
                  SEND_TO => 3,
                  ADDRESS => XPDNP_LOC );
      loop
         exit when INPUT_READY;
         --busy loop, waiting for signal
         --corresponds to RESUME
      end loop;
      end loop;
end 3;
```

```
procedure C is


procedure C is
   XNP3, XPDN, XPDNP : VECTOR;
   XDN, XN           : VECTOR := ORIGIN;

   begin
      loop
         exit when INPUT_READY;
         --busy loop, waiting for signal
         --corresponds to START_UP
      end loop;
      loop
         MOVE(TO => XPDN, FROM => XPDN_LOC );
         MOVE(TO => XPDNP, FROM => XPDNP_LOC );
         MOVE(TO => XNP3, FROM => XNP3_LOC );
         XN  := XNP3 - 1.5*H*(XPDN - 3.0*XPDNP);
         XDN := DERIVATIVE(XN);
         TRANSFER(VALUE   => XDN,
                  SEND_TO => D,
                  ADDRESS => XDNP2_LOC );
         TRANSFER(VALUE   => XN,
                  SEND_TO => B,
                  ADDRESS => XNP2_LOC );
         TRANSFER(VALUE   => XN,
                  SEND_TO => A,
                  ADDRESS => XNP2_LOC );
      loop
         exit when INPUT_READY;
         --busy loop, waiting for signal
         --corresponds to RESUME
      end loop;
      end loop;
end C;
```

```
   procedure D is

   procedure D is
      XDPN2, XNP3 : VECTOR;
      XNP         : VECTOR := ORIGIN;

      begin
         loop
            exit when INPUT_READY;
            --busy loop, waiting for signal
            --corresponds to START_UP
         end loop;
         loop
            MOVE(TO => XDNP2, FROM => XDNP2_LOC );
            MOVE(TO => XNP3, FROM => XNP3_LOC );
            XNP := XNP3 + 2.0 * H * XDNP2;
            XDNP := DERIVATIVE(XNP);
            TRANSFER(VALUE   => XNP,
                     SEND_TO => D,
                     ADDRESS => XNP3_LOC );
            TRANSFER(VALUE   => XNP,
                     SEND_TO => C,
                     ADDRESS => XNP2_LOC );
         loop
            exit when INPUT_READY;
            --busy loop, waiting for signal
            --corresponds to RESUME
         end loop;
         end loop;
   end D;
```

```
procedure IOP is


procedure IOP is
   XPNX,
   XPNX2 : VECTOR;

      use TEXT_IO;
      package INT_IO is new INTEGER_IO(INTEGER);
      package REAL_IO is new FLOAT_IO(FLOAT);
      use INT_IO, REAL_IO;

      S1  : INTEGER := 1;
      TIN : FLOAT := 0.08;
      TV  : array (1..500) of FLOAT;
      XV  : array (1..500) of FLOAT;

   begin
      accept RESUME;
      TV(1) := TIN;
      TRANSFER CONTROLLER.SIGNAL;
      for I in 1... loop
         MOVE(TO => XPNX2, FROM => XPNX2_LOC );
         MOVE(TO => XPNX, FROM => XPNX_LOC );
      loop
         exit when INPUT_READY;
         --busy loop, waiting for signal
         --corresponds to RESUME
      end loop;
         XV(S1    ) := FIRST(XPNX);
         XV(S1 + 1) := FIRST(XPNX2);
         if NT > 1 then
            TV(S1) := TV(S1 - 1) + H;
         end if;
         TV(S1 + 1) := TV(S1) + H;
         S1         := S1 + 2;
      end loop;
      PUT(N);
      PUT_LINE;
      for I in 1..N + 1 loop
         PUT(TV(I));
         PUT(XV(I));
         PUT_LINE;



procedure IOP is


      end loop;
end IOP;
```

RUN OF THE PRECOMPILER OUTPUT ON A SIMULATED NETWORK

The procedures output by the precompiler were run on a network simulated by a set of Ada tasks running on a WICAT computer on which a large subset of Ada is implemented. Each task of the following program represents a processor node of a network.

After the original program was split by the preprocessor, the components were moved to the WICAT and all non-supported Ada features were removed (manually). The components were then recombined into the following program, compiled using the WICAT Ada compiler, and run.

```
--  This Ada program consists of several compilation units and compiles


--  This Ada program consists of several compilation units and compiles
--  on the Wicat Ada compiler
--  It was produced by  running the precompiler written by Laurie King
--  on the file Another.ada which contains the source (almost) equivalent to
--  the Ada program originally run at ICASE.
--
--  After splitting the program the components were moved to the Wicat and all
--  non-supported Ada features were removed.  The components were then
--  recombined and compiled.  Two discrepencies from the ICASE-correct version
--  were discovered and corrected:  Variables were mistyped resulting in
--  other program variable names.
--
    with global; use global;
    with vectors; use vectors;
    with bus; use bus;
    WITH TEXT_IO; use TEXT_IO;
         --WITH INTEGER_IO,
           --FLOAT_IO(FLOAT);
    Use integer_io,float_io;
    Procedure main is
    task A is end;
    task B is end;
    task C is end;
    task D is end;
    task IOP is end;

    task body  A is
      XNP1, XPDN     : VECTOR;
      XPNX2, XPDNX2 : VECTOR := ORIGIN;

      begin

         loop
            exit when INPUT_READY;
            --Busy_loop, waiting for signal
            --corresponds to START_UP
            end loop;

         loop
            MOVE(XPDN, XPDN_LOC);
```

```
--  This Ada program consists of several compilation units and compiles

                            put_line("A");
MOVE(XNP2, XNP2_LOC );
            XPNX2  := XNP2 + 4.0*H*XPDN;
            XPDNX2 := DERIVATIVE(XPNX2);
            TRANSFER(XPNX2,
                     IOP_task,
                     XPNX2_LOC );
            TRANSFER(XPDNX2,
                     C_task,
                     XPDN_LOC );
            TRANSFER(XPDNX2,
                     B_task,
                     XPDN_LOC );
            TRANSFER(XPDNX2,
                     A_task,
                     XPDN_LOC );
        loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to RESUME
         end loop;
         end loop;
    end A;

 task body  B is
      XNP2, XPDN, XPDNP : VECTOR;
      XPNX, XPDNX       : VECTOR := ORIGIN;

      begin
         loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to START_UP
         end loop;
         loop
            MOVE(XPDN, XPDN_LOC );
            put_line("B");
            MOVE(XPDNP, XPDNP_LOC );
            MOVE(XNP2, XNP2_LOC );
            XPNX  := XNP2 + 1.5*H*(XPDN + XPDNP);
```

```
-- This Ada program consists of several compilation units and compiles

          XPDNX := DERIVATIVE(XPNX);
          TRANSFER(XPNX,
                   IOP_task,
                   XPNX_LOC );
          TRANSFER(XPDNX,
                   C_task,
                   XPDNP_LOC );
          TRANSFER(XPDNX,
                   B_task,
                   XPDNP_LOC );
       loop
          exit when INPUT_READY;
          --Busy loop, waiting for signal
          --corresponds to RESUME
       end loop;
       end loop;
end B;


task body C is
   XNP3, XPDN, XPDNP : VECTOR;
   XDN, XN           : VECTOR := ORIGIN;

   begin
      loop
         exit when INPUT_READY;
         --Busy loop, waiting for signal
         --corresponds to START_UP
      end loop;
      loop

         MOVE(XPDN, XPDN_LOC );
                          PUT_LINE("C");
MOVE(XPDNP, XPDNP_LOC );
         MOVE(XNP3, XNP3_LOC );
         XN  := XNP3 - 1.5*H*(XPDN - 3.0*XPDNP);
         XDN := DERIVATIVE(XN);
         TRANSFER(XDN,
                  D_task,
                  XDNP3_LOC );
```

```
-- This Ada program consists of several compilation units and compiles

            TRANSFER(XN,
                     B_task,
                     XNP2_LOC );
            TRANSFER(XN,
                     A_task,
                     XNP2_LOC );
         loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to RESUME
         end loop;
         end loop;
   end C;

   task body  D is
      XDNP2, XNP3 : VECTOR;
      XNP         : VECTOR := ORIGIN;

   begin
                  loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to START_UP
         end loop;
         loop
            MOVE(XDNP2, XDNP2_LOC );
                        PUT_LINE("D");
MOVE(XNP3, XNP3_LOC );
            XNP := XNP3 + 2.0 * H * XDNP2;
            TRANSFER(XNP,
                     D_task,
                     XNP3_LOC );
            TRANSFER(XNP,
                     C_task,
                     XNP3_LOC );
         loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to RESUME
```

```
-- This Ada program consists of several compilation units and compiles

          end loop;
          end loop;
   end D;

   task body  IOP is
      XPNX,
      XPNX2 : VECTOR;

          S1  : INTEGER := 1;
          TIN : FLOAT := 0.09;
          TV  : array (1..500) of FLOAT;
          XV  : array (1..500) of FLOAT;

      begin
         loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to START_UP
         end loop;
--accept RESUME;
          TV(1) := TIN;
          --TRANSFER_CONTROLLER.SIGNAL;

         for I in 1..N loop
            PUT_LINE("P");
            --accept RESUME;
            MOVE(XPNX2, XPNX2_LOC );
            MOVE(XPNX, XPNX_LOC );
          loop
            exit when INPUT_READY;
            --Busy loop, waiting for signal
            --corresponds to RESUME
         end loop;
            XV(S1    ) := FIRST(XPNX);
            XV(S1 + 1) := FIRST(XPNX2);
         if I > 1 then
               TV(S1) := TV(S1 - 1) + H;
            end if;
```

```
-- This Ada program consists of several compilation units and compiles

            TV(S1 + 1) := TV(S1) + H;
            S1         := S1 + 2;
         end loop;
         PUT(N);
         text_io.PUT_LINE("*");
         for I in 1..N + 1 loop
            PUT(TV(I));
            PUT(XV(I));
            text_io.PUT_LINE("*");
         end loop;
   end IOP;

begin
   NULL;
   PUT_LINE("XXX"); put_line("Main");
end main;
```

```
with global; use global;

with global; use global;
with vectors; use vectors;

package BUS is

      INPUT_READY : boolean := true;

      type BUS_ADDR is (XPDN_LOC, XPNX2_LOC,XDNP2_LOC, XNP2_LOC,
                        XPDNP_LOC, XPNX_LOC, XNP3_LOC);

      BUSV :  ARRAY (BUS_ADDR) of VECTOR;

      procedure MOVE ( TO : out  VECTOR;
                       FROM : BUS_ADDR);

      procedure TRANSFER (Value : VECTOR;
                          SEND_TO : TASK_NAME;
                          ADDRESS : BUS_ADDR);

end BUS;

with text_io; use text_io;
package body BUS is

      procedure MOVE ( TO : out VECTOR;
                       FROM : BUS_ADDR) is

        begin
              put_line("move");
              PUT_LINE("L");
              TO := BUSV(FROM);
              PUT_LINE("L");
          end move;

      procedure TRANSFER (VALUE   : VECTOR;
                          SEND_TO : TASK_NAME;
                          ADDRESS : BUS_ADDR) is
          begin
              put_line("transfer");
```

```
With global; use global;

                BUSV(ADDRESS) := VALUE;
            null;
        end;

end BUS;
```