Hierarchical decomposition and simulation of manufacturing cells using Ada

Charles J. Antonelli, Richard A. Volz and Trevor N. Mudge
Center for Robotics and Integrated Manufacturing
Electrical and Computer Engineering Department
The University of Michigan
Ann Arbor, Michigan 48109

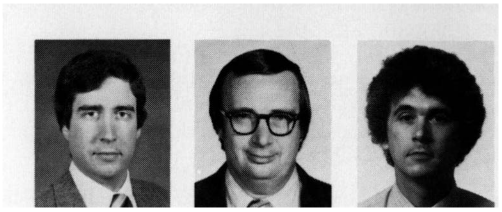
To Dick Volz, who taught me early in my scientific career the difference between a conference paper and a journal article, and at whose side I learned to work independently. Thanks, Dick, for everything.

Charles Antonelli

SIMULATION 46:4, 141-152.
©1986 by Simulation Councils, Inc.
ISSN 0037-5497/86 $2.00+.10
Printed in the United States of America

Key words: functional description, hierarchical decomposition, manufacturing cell simulation, system description language

Hierarchical decomposition and simulation of manufacturing cells using Ada*



Charles J. Antonelli, Richard A. Volz, and **Trevor N. Mudge**
Robot Research Division
Center for Robotics and Integrated Manufacturing
Electrical and Computer Engineering Department
The University of Michigan
Ann Arbor, Michigan 48109

CHARLES J. ANTONELLI received the BSEE and BS in computer engineering in 1977 and the MS degree in computer, information, and control engineering in 1979 from the University of Michigan. He has been employed by Bell Laboratories since 1979 and is currently on educational leave of absence to pursue the PhD degree in the Department of Electrical Engineering and Computer Science at the University of Michigan. Mr. Antonelli is a member of the Institute of Electrical and Electronics Engineers and a member of the Association for Computing Machinery. His research interests include the architecture of distributed systems, distributed operating systems and languages, and software engineering.

RICHARD A. VOLZ is a director of the robot systems division of the Center for Robotics and Integrated Manufacturing at the University of Michigan. Prior to this, he was associate director of the University Computer Center and associate chairman of the Electrical and Computer Engineering Department.

Professor Volz received his education at Northwestern University, receiving a PhD in 1964 and joined the University of Michigan that year. He has worked on computational techniques for automatic control systems and done pioneering work on computer-aided-design methods for control systems. Real-time computing systems are a more recent technical interest of his, and he has led in the design and implementation of both a real-time operating system and a higher level language for real-time control. His current research interests include software/hardware computer architectures to support robot systems and the use of geometric models obtained from computer-aided-design systems for driving robot and sensor programming. Particular projects include (CAD) model driven vision systems, automatic determination of gripping points on objects (from CAD information), graphic programming of robots and distributed systems integration languages for real-time control.

TREVOR MUDGE received the BSc degree in cybernetics from the University of Reading, England, in 1969, and the MS and PhD degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively. He has been with the Department of Electrical Engineering and Computer Science at the University of Michigan since 1977 and currently holds the rank of associate professor. Dr. Mudge is a senior member of the Institute of Electrical and Electronics Engineers, a member of the Association for Computing Machinery, the Institution of Electrical Engineers and the British Computer Society. His research interests include computer architecture, programming languages, VLSI design and computer vision.

**Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).*

ABSTRACT

A useful tool in the development of flexible automation is a system description language which can generate a complete functional description of a manufacturing cell of arbitrary complexity. We propose a description system based on the concept of hierarchical decomposition utilizing the Ada programming language in conjunction with established diagrammatical decomposition methods. The distinguishing aspect of our work is that it takes advantage of certain features of Ada (such as type checking) to create a description that can be automatically verified for consistency. Simulation is often an indispensable tool in the development of manufacturing systems. We show how a simulation of the operation of the manufacturing cell can be embedded in its description. Finally, we apply the methodology to a specific instance of a manufacturing cell.

INTRODUCTION

In general, design begins with a functional description or specification of the object to be designed. The design must then be performed in such a manner as to match the specification. In some areas, particularly software design, the specification process and techniques to ensure consistency between a specification and a design have been highly formalized and automated. Ross and Schoman[5] and Teichroew and Hershey,[8] among others, have demonstrated that a consistent specification methodology and automated consistency checking significantly aid the design process.

A recurring problem in designing manufacturing cells is the lack of a suitable framework on which a correct functional description can be built. Manufacturing cells contain a number of complex subsystems, such as programmable controllers, computer-numerically-controlled (CNC) machines, robots, and material handling and storage systems, whose operations and interactions must be uniformly described. Each such subsystem requires a different set of time-sequenced inputs and outputs in order to perform its function. These inputs and outputs can utilize discrete I/O lines, analog channels, or synchronous and asynchronous communication protocols. Each of these communication media must meet differing rate requirements and

Terms and definitions*

Accept statement – See entry.

Access value – An access value is the value of an access type and designates an object; the access value can be used to read and update the designated object. Access values are known as pointers in other languages.

Allocator – The evaluation of an allocator creates an object and returns a new access value that designates the object.

Declaration – A delaration associates an identifier (or some other notation) with an entity. This association is, in effect, within a region of text called the scope of the declaration. Within this scope, it is possible to use the identifier to refer to the associated declared entity.

Elaboration – The elaboration of a declaration is the process by which the declaration achieves its effect (such as creating an object); this process occurs during program execution.

Entry – An entry is used for communications between tasks. Externally, an entry is called just as a procedure is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called.

Object – An object contains a value. A program creates an object either by elaborating an object declaration or by evaluating an allocator. The declaration or allocator specifies a type for the object: the object can only contain values for that type.

Package – A package specifies a group of logically related entities, such as types, objects of those types, and subprograms with parameters of those types. It is written as a package declaration and a package body. The package declaration has a visible part which contains the declarations of all entities that can be explicitly used outside the package. The package body contains implementations of subprograms (and possibly tasks as other packages) that have been specified in the package declaration.

Parameter – A parameter is one of the named entities associated with a subprogram or entry and is used to communicate with the corresponding subprogram body or accept statement. A formal parameter is an identifier used to denote the named entity within the body. An actual parameter is the particular entity associated with the corresponding formal parameter by a subprogram call or entry call.

Record type – A value of a record type consists of a collection of components which are usually of different types. Such a record object may be used to group together a set of related components.

Rendezvous – A rendezvous is the interaction that occurs between two parallel tasks when one task has called the entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task.

Scope – See declaration.

Subprogram – A subprogram is either a procedure or function. It is written as a subprogram declaration, which specifies its name, formal parameters, and (for a function) its result; and a subprogram body which specifies the sequence of actions. The subprogram call specifies the actual parameters that are to be associated with the formal parameters.

Task – A task operates in parallel with other parts of the program. It is written as a task specification (which specifies the name of the task and the names and formal parameters of its entries), and a task body which defines its execution.

Type – A type characterizes both a set of values and a set of operations applicable to those values.

**Abridged from Appendix D in Reference 9.*

require differing error recovery strategies. Futhermore, a potentially extensive database must be maintained to accurately reflect the current states of all parts flowing through the cell, as well as the current state of all subsystems in the cell. The heterogeneous nature of the cell dictates widely differing data representations, access requirements, and access rates.

In view of the preceding, we believe that a formal functional description technique would be highly valuable in designing manufacturing cells. In general, it will not be possible to match implementation with specification as can be done to some extent in software design; however, extending the description to a simulation of the cell being designed can incur many of the same advantages. Such a description system should have at least the following attributes:

(1) Completeness – The functional description must completely specify the manufacturing cell in question. This implies that all interactions between the components of the cell, implicit and explicit, must be accounted for.

(2) Consistency – The constituent parts of the functional description must be consistent with each other. Rate and protocol of a sender must match those of a receiver; parts output by one subsystem must correspond to the input requirements of a succeeding subsystem.

(3) Ease of understanding – The functional description must be easily understood at varying levels of detail. It must be possible to gain a high-level understanding of the entire cell without the burden of excessive detail; it must also be possible to gain a detailed understanding of any particular component of the cell.

(4) Amenity to simulation – It should be possible to develop a simulation of the system from its description. Either by executing the description directly, or by providing a translation method whereby the description is transformed into a series of simulation statements which can then be executed.

At present, it is possible to give quite specific functional descriptions of each component of a manufacturing cell. These descriptions take the form of manufacturer's specifications, wiring diagrams, shop floor layouts, and so forth. Unfortunately, it is difficult to combine these descriptions into a coherent set of specifications at the manufacturing cell level, particularly one that is amenable to simulation.

In this paper we explore and extensively discuss the development of a complete, consistent, level-sensitive functional description of a manufacturing cell. We observe that our description is amenable to simulation, and explore that concept, and finally, we illustrate our concepts with a simple case study.

DESCRIPTIVE METHODOLOGY

One way of achieving our stated goal is through a *system description language* which can completely describe a manufacturing cell at a suitable level of detail. Examples of such languages and associated analysis systems are well known. Ross and Schoman[5] claim a lack of an adequate approach to requirements definition as a major source of difficulty in systems work, and propose context analysis, functional specification, and design constraints as solutions to this difficulty. Within the context of functional specifications they set forth the Structured Analysis and Design Technique (SADT) as a tool in buliding complex specifications. SADT, a graphical structure which is both modular and hierarchical, is used to describe functional architecture. Sammet, Waugh and Reiter[6] describe PDL/Ada (Program Description Language/Ada), which is a procedural high-level language used in writing software specifications that is based on the Ada programming language.[9] Using PDL/Ada, a modular textual structure is used to describe functional architecture. Teichroew and Hershey[8] discuss PSL/PSA (Problem Statement Language/Problem Statement Analyzer), which is a computer-aided structured documentation and analysis technique used in describing arbitrary information systems. Using PSL, a modular textual structure used to generate functional specifications for information processing systems. The associated analyzer, PSA, can be used to generate a variety of reports based on the PSL descriptions.

These examples make use of the concept of *hierarchical decomposition*; that is, decomposing a difficult problem into several simpler subproblems to allow a solution when direct methods fail. We apply this technique to the problem of generating functional descriptions of manufacturing cells, utilizing two complimentary descriptive formats. We then show that one of the techniques is readily extended to provide a simulation for the system being designed.

In the first format, *diagrammatical decomposition*, we present a diagram of the functional description. The hierarchical decomposition is shown as a series of nested diagrams, and directed lines between elements of the diagram describe the data and control flow. This format is roughly analogous to that of SADT and allows the reader to obtain a quick, intuitive understanding of the manufacturing cell being described.

In the second format, *procedural decomposition*, we present an equivalent functional description written in a procedural description language based on the Ada programming language, along the lines explored in Reference 6. However, there is a key difference. Whereas PDL/Ada does not support the parallel execution of the descriptive components, this concept is central to our description system. The hierarchical decomposition is shown as a series of nested Ada packages, and Ada task rendezvous describe the data and control flow. This procedural decomposition is much more detailed than the diagrammatical one and gives the reader a complete functional decomposition of the cell being described. We believe that both formats are necessary for complete understanding.

Our hierarchical concept imposes a great deal of structure on the description process. While the task of generating a complete description of a large manufacturing cell remains formidable, the method of hierarchical decomposition provides a way of systematically generating correct functional descriptions to any desired level of detail.

Diagrammatical decomposition

The basic unit of diagrammatical decomposition is the *box*. In Figure 1, there is a number of *inputs* to a box, a number of *outputs* from the box, and a *function*, mapping the inputs to the outputs, performed by the box. The first, or top, level of decomposition is a description of the manufacturing cell, and the inputs and outputs are the actual inputs and outputs of the cell. We do not distinguish at this level between physical and nonphysical objects.

The *exterior* of a box encloses the current level of decomposition. The box shown in Figure 1 resides at the first level of decomposition. The inputs are *part*$_1$ and *part*$_2$, the output is *part*$_3$, and the function performed by the box is the joining of the first two parts to yield the third. This level of decomposition does not describe how the assembly is to be performed, only that it is to take place.

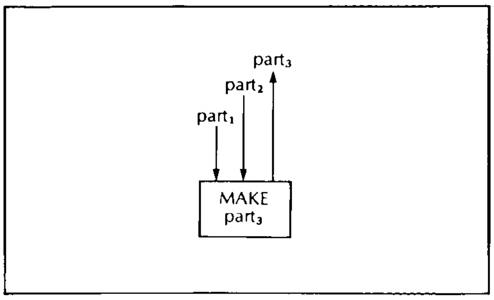


Figure 1. Box exterior.

The *interior* of a box contains a collection of *subboxes*. The collection of subboxes forms the next level of decomposition; their collective function is identical to the function of the enclosing box. One of the subboxes is designated as a *control* subbox, and its function is to serve as a manager of control and data flow within the box by specifying, if required, the order in which the other functional subboxes should be invoked, what inputs they should be invoked with, and what outputs they should return to realize the function of the enclosing box. In Figure 2, the box of Figure 1 has been opened to reveal the subboxes inside. (We call the process of opening a box a *decomposition step*.) The functional subboxes f_1, f_2, and f_3 represent the three operations "pick up part 1," "pick up part 2," and "join parts." Together these three operations realize the function of the enclosing box.

We can recursively descend in the hierarchical decomposition by opening subboxes to reveal other subboxes contained within them. This process continues until a level of decomposition is reached at which further partitioning is unnecessary. In our example, a subbox whose function is "close gripper on robot arm 1" is probably not amenable to further decomposition; we call it a *terminal subbox*. The successive decompositions of a cell can be viewed as a tree which represents a collection of descriptions at different levels of detail.

We emphasize the difference between hierarchy of *decomposition* and hierarchy of *control*. Our hierarchical decomposition is primarily a description of a manufacturing cell. As such, the functional boxes are abstractions and may not in general have physical counterparts in the cell itself. At some level of decomposition, however, the functional boxes should correspond to physical entities or control program procedures, and the inputs and outputs are associated directly with the terminal subboxes.

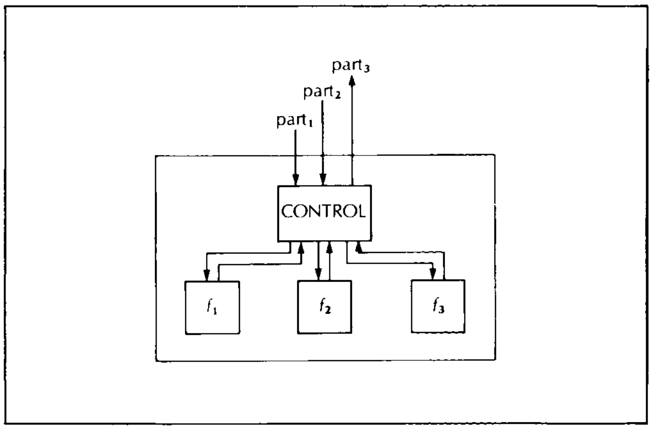


Figure 2. Box interior.

Procedural decomposition

The diagrammatical decomposition method provides an elegant way of decomposing a manufacturing cell. However, futher examination of Figures 1 and 2 points out some insufficiencies in this method. First, considerable descriptive detail is missing from the diagram, such as precise definitions of the parts and operations involved. Second, while it is possible for the function f_1 and f_2 of Figure 2 to happen concurrently, f_3 must wait for them to complete before proceeding if the joining process is to be correctly described. This flow of control is determined by the control subbox, but is not explicitly given in Figure 2. Attempting to correct both of these omissions by increasing the amount of detail in the diagrams would clutter the description to the point where it would be difficult to interpret correctly. To deal with these problems we use a *procedural decomposition language* to complete the hierarchical decomposition.

Ada language review

Our procedural description language is based on the Ada programming language. Hence, we review briefly those portions of the Ada language salient to our needs. Ada[9] is a strongly-typed, block-structured programming language with support for concurrent execution, separate compilation, reusable software modules, and extensive compile- and run-time verification. Ada defines procedures and functions in the usual way. *Tasks* are units of concurrent execution. Execution of the statements contained in a task proceeds independently of the rest of an Ada program except at specifically designated synchronization points.

The *rendezvous* is the only means of synchronization in Ada and can be thought of as an intertask procedure call. During a rendezvous, both tasks are synchronized while any associated parameters are exchanged, after which both continue independently. Like a procedure call, a rendezvous is performed by two parties: the caller, who performs an *entry call*, and the callee, who performs an *accept*. The task performing an entry call is suspended until the called task performs the corresponding accept statement or vice versa. In Figure 3, task SEND executes the entry call RECEIVE.COMMAND(10) while task RECEIVE independently executes the (possibly compound) accept statement accept COMMAND ... end COMMAND. (In *fully qualified or dotted notation*, a call on entry COMMAND in task RECEIVE is written RECEIVE.COMMAND.) During the time task RECEIVE executes the accept statement, the two tasks are said to be in *rendezvous*. During the rendezvous, SEND is suspended while RECEIVE executes the statement COM := VALUE which saves the value of the passed parameter in COM. This is done because the values assigned to formal parameters of an accept statement are available only during the scope of the statement. When the execution of the accept statement is finished the rendezvous is completed and both tasks continue execution independently.

Program and data objects may be grouped together in a *package*. Objects so grouped share identical scope and visibility. In Ada, packages can be used as a passive grouping mechanism. Extensive compile-time checking is performed to ensure that formal and actual parameters do not conflict, and run-time checking is performed to ensure invalid assignments and references are detected. (It is this extensive error checking that provides a large measure of confidence in the consistency of the description of the interacting parts of the manufacturing cell.) Ada supports separate compilation since it does not force an entire program to be recompiled when a change is made to a single module.

```
task body SEND is
begin
  .
  RECEIVE.COMMAND(10);
  .
end SEND;

task body RECEIVE is
  COM: INTEGER;
begin
  .
  accept COMMAND(VALUE: INTEGER) do
    COM := VALUE;
  end COMMAND;
  .
end RECEIVE;
```

Figure 3. Example of a rendezvous.

Procedural decomposition language

We now give a characterization of our procedural description language. The functional units of the procedural decomposition are represented as Ada tasks, just as we represent the functional units of the diagrammatical decomposition as boxes. Tasks were chosen instead of procedures or functions because tasks execute in parallel and thus provide a more realistic description of simultaneous events than do sequential constructs.

At a given level of decomposition, a task representing a functional block must convey the following information. First, it must show the interconnection with other tasks by characterizing the inputs and outputs of the task and by describing how these inputs and outputs are synchronized with each other and with those of the other tasks. The description is provided informally through comments, but all interconnections with the surroundings of the functional block are represented formally so that the overall structure of the system being described may be automatically checked by an Ada compiler at a level which guarantees that part and data flow among different components of the system are consistent. Through hierarchical decomposition (described in greater detail below) this can be carried to any desired level of detail; thus it is possible even to represent timing signals used by the system. The following paragraphs illustrate the description conventions which allow this to be accomplished.

The functional description is provided via comments in the Ada text. Each task specification contains a textual description of the function of the unit the task represents, the inputs to and outputs from that unit, and any synchronization which is to take place with other units. This text is called a *DIO* list (shorthand for Description/Input/Output) and is placed at the beginning of the task specification. See Figure 4 for an abstraction of the use of a task for the functional description of a block.

Each functional block receives certain inputs from, and provides certain outputs to, its surrounding environment, each of which may be synchronized with other functional units. Each separate kind of input or output, whether physical object or data (e.g., raw stock to be machined, finished parts, or data record describing the measured tolerances of a part), is given an Ada type. This establishes different kinds of objects as separate kinds of things and enables the compiler to check the

```
task T is
  -- DIO T:
  --   description is ...
  --   inputs are INPUT_LIST.
  --   outputs are OUTPUT_LIST.
  entry START(INPUT_LIST);
  entry STOP(OUTPUT_LIST);
  entry ...
end T;

task body T is
begin
  loop
    accept START(INPUT_LIST)
      LOCAL_INPUT_LIST := INPUT_LIST;
    end START;

    LOCAL_OUTPUT_LIST := F(LOCAL_INPUT_LIST);

    accept STOP(OUTPUT_LIST) do
      OUTPUT_LIST := LOCAL_OUTPUT_LIST;
    end STOP;
  end loop;
end T;
```

Figure 4. Task implementation.

```
task T' is
  -- DIO T':
  --   description is ...
  --   inputs are INPUT_LIST.
  --   outputs are OUTPUT_LIST.
  entry START(INPUT_LIST);
  entry STOP(OUTPUT_LIST);
end T';

task body T' is
begin
  loop
    accept START(INPUT_LIST)
      LOCAL_INPUT_LIST := INPUT_LIST;
    end START;

      t0.START(LOCAL_INPUT_LIST);
      t0.STOP(LOCAL_OUTPUT_LIST);

    accept STOP(OUTPUT_LIST) do
      OUTPUT_LIST := LOCAL_OUTPUT_LIST;
    end STOP;
  end loop;
end T';
```

Figure 5. Interface task implementation.

consistency of usage. The declarations for these types are external to the task representation of the block and thus do not appear in Figure 4. They are visible to this task, however, and may be referenced. They are represented by the abstractions INPUT__LIST and OUTPUT__LIST in the figure.

Inputs to and outputs from the block are shown formally by providing **entry** statements for each input and output. At a minimum, there will be an input corresponding to the beginning of the block and an output at the end. These are shown by the START and STOP entries in Figure 4. There may be additional **entry** statements associated with synchronization or communication with other parts of the system, as necessary. Corresponding to the **entry** statements, there will be **accept** statements in the body of the task through which the actual passing of the inputs and outputs is represented. The representation of inputs and outputs in this fashion, together with the use of types for different kinds of objects allows an Ada compiler to check for consistency of the interconnections between different parts of the system being described. It is this very type of consistency checking which has proven to be extremely valuable for early error detection in the development of complex software systems, and is one of the principal advantages to the procedural portion of the description system presented here.

The use of Ada rendezvous for representing the input and output operations also provides a natural mechanism for representing the time synchronization among different components of the system. As will be seen later, this is readily extended into a simulation of the system.

Finally, for manufacturing operations, functional blocks typically represent activities that are repeated in time. This is represented by a main loop within the body of the task. The function F shown in the middle of the task body in Figure 4 represents the operation the functional block is to perform; this operation was described in the DIO list formal representation of the functional block implemented by the task. For simulation operations it will be replaced by linkages to a scheduler and a function which increments time by the amount required for the operations discussed in the next section.

We thus see how an Ada task can be used to describe informally (and to a limited extent formally) the operation of a functional block. The task description of the manufacturing system is encompassed by an environment, also represented as an Ada task, which provides the inputs to, and accepts the outputs from, the cell being described. This environment is not shown here, but is somewhat similar in nature to the cell description. It is this environment which includes the necessary type definitions to represent the different kinds of objects in the system.

Next, we examine the manner in which decomposition is accomplished to achieve a more detailed description of the system, and describe the effects of a decomposition step on task T. When such a step is made, task T shown in Figure 4 is replaced by an *interface* task T' shown in Figure 5, a *control* task t_0 corresponding to the control subbox of Figure 2, and a set of *functional* tasks t_i corresponding to the functional subboxes of Figure 2. The control and functional tasks which make up the decomposition of T are identical in form to Figure 4 and are not shown. If there are *n* functional tasks we have the three relations:

$$F \Leftarrow \sum_{i=0}^{n} F_i,$$

$$\text{INPUT\_\_LIST} \subseteq \bigcup_{i=0}^{n} \text{INPUT\_\_LIST}_i,$$

and

$$\text{OUTPUT\_\_LIST} \subseteq \bigcup_{i=0}^{n} \text{OUTPUT\_\_LIST}_i.$$

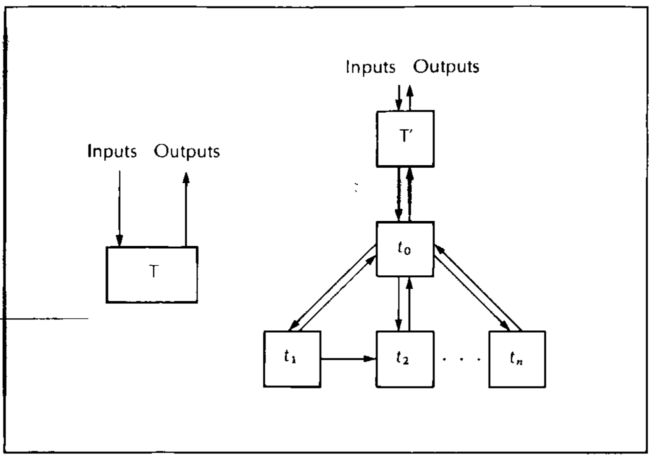


Figure 6. Results of decomposition step.

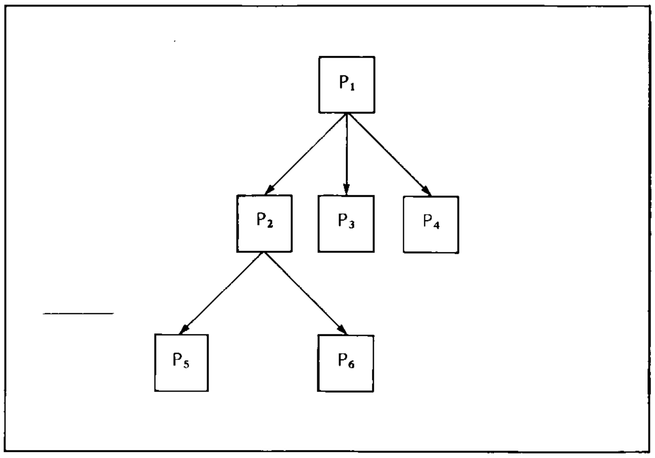


Figure 7. Package tree.

These equations denote that the collective function of the functional tasks replace the function of the original task and that the combined inputs and outputs of the functional tasks subsume the inputs and outputs of the original task. This is shown diagrammatically in Figure 6.

The interface task is introduced in order to isolate other parts of the description from changes required by a decomposition step, and to provide a mechanism for dispersing and collecting the inputs and outputs of T. It is identical to the original task T except that the statements representing the function F are replaced by START and STOP rendezvous calls to the control task t_0. The control task is reponsible for sequencing the execution of the remaining functional tasks $t_1,\ldots,t_n$; its body consists of START and STOP rendezvous calls to the rest of the functional tasks in a manner similar to that shown in Figure 5. Although not evident in this simple abstraction, any additional entries for input, output, or synchronization operations would also be passed downward to the task corresponding to the control subbox at the next level of decomposition.

For convenience in organizing a decomposition, we place the control and functional tasks generated by the decomposition step into a package and make them visible to the interface task which remains outside the package. This partitioning of the decomposition into packages provides for a more understandable description and allows portions of a large description to be compiled separately. The interface task thus replaces T in its package; the control task and the set of functional tasks are enclosed in a new package. Everything that is to be accessible to the exterior of a package must be listed in the package specification according to Ada rules; everything else in the package body is hidden from view. Thus the package specification contains only a description of the control task; it is the only task that must be visible, for it will be invoked from a task in a different package at the next higher level of decomposition. The remaining tasks are invoked from the control task, or can invoke each other, and thus need not be visible outside the package.

The similarity in form between a functional task t_i and the original task T allows further decomposition steps to be taken by considering one of the functional tasks as a new task T and repeating the decomposition procedure. This results in a tree of packages as seen in Figure 7. At some point it becomes unnecessary to decompose a task any further; we call such a task a *terminal* task which will have the form of Figure 4. For example, P_3 and P_4 of Figure 7 contain only terminal tasks, while P_2 contains two tasks which are not terminal. Terminal tasks will play an important role when simulation is considered, as discussed in the next section. Thus the set of all terminal tasks plus all control tasks contains the entire functional description of the manufacturing cell.

A key factor in the procedural description methodology is the fact that the specifications can be compiled by any validated Ada compiler. The compiler error checking mechanisms will detect any inconsistencies in number or type between the formal and actual parameters representing data or physical objects which are passed between different components of the system being described. This is a powerful design aid and is the principal motivation for using the procedural design methodology. It allows many design errors to be caught at a very early stage of the design life cycle where the costs of such errors are minimal.

SIMULATION METHODOLOGY

Our hierarchical description provides the basis for simulation control software which supports a process-oriented simulation scheme. Thus, another major attribute of the procedural description methodology is that it can be extended to support the simulation of the system being designed at the same varying level of detail as the decomposition of the system. We can replace the function descriptions in the terminal tasks with process-oriented simulation statements. These are generally very simple constructs such as wait statements to simulate the passage of time that occurs when the function represented by the task is performed. We also need to add some support software to manage the process-oriented simulation, such as scheduling routines, clock managers, and so forth. While these functions can be provided by any good simulation package, they are simple enough that we have chosen to implement our own simulation support package.

In conventional process-oriented discrete-event simulation systems, a number of simulation processes appear to execute in parallel. In fact, only one such process is executing at a given time, and that process continues to execute until it chooses to stop, at which time the simulation system schedules another process for execution. This process-oriented simulation scheme utilizes one master simulation clock.

We desired a parallel implementation to take advantage of the nature of our description. Since our description is implemented in terms of Ada tasks, it seemed natural to allow these tasks to execute concurrently during the simulation to take advantage of any underlying multiprocessor hardware, as suggested in Reference 3.

In a parallel discrete-event simulator in which there are many processes executing concurrently a single master simulation clock no longer suffices. Consider two processes, A and B, which are executing simultaneously. Suppose A schedules another process, C, to run at time t_1 and subsequently gives up control. Assume C turns out to be the next process that is activated, with the master simulation clock set to t_1. If B, which is still running, schedules another process, D, to run at time t_2, where $t_2 < t_1$, we could be faced with the problem of having to roll back the simulation clock to t_2 and undoing whatever C has had a chance to do in the interval $[t_2, t_1]$. It is evident that we must provide a mechanism for managing the master simulation clock so as to avoid this rollback.

We have developed such a mechanism for use with our description system using a concept first described in Reference 4. We supply each task with a local simulation clock in addition to the global clock. Each task consults its own local clock to determine its course of action; the local clock thus completely determines a task's view of simulation time. This local clock is synchronized with the global clock whenever the task invokes one of the following primitives.

- Wait – A task wishes to advance its local clock by a given amount. When execution of the task resumes, its local clock will be incremented by the specified time.

- Intend to rendezvous – A task wishes to rendezvous with another task by performing an entry call. In this case both the invoker's local clock and the local clock of the task that executes the corresponding accept may require updating. When the rendezvous takes place, both tasks will have their local clocks set to the larger of the original local clocks.

- Intend to accept – A task wishes to rendezvous with another task by performing an accept statement. In this case both the invoker's local clock and the local clock of the task that executes the corresponding entry call may require updating. When the rendezvous takes place, both tasks will have their local clocks set to the larger of the original local clocks. Thus the "intend to rendezvous" and the "intend to accept" primitives cause the local clocks to be updated in the same way.

Whenever a task needs to perform one of these primitives, it performs a rendezvous with the simulation controller which changes the state of the task from running to a wait state. A skeleton of the simulation controller implementation is given in Figure 8. Task sched manages client calls of the aforemen-

```
task body sched is
begin
  -- Perform initialization via sched.activate .
  loop
    select
      accept wait(...)           -- Wait.
      -- Compute wakeup time and set task to WAITING state.
      advance_global_time;
    or
      accept itr(...)            -- Intend to rendezvous (call).
      if (partner already WAITING_TO_ACCEPT) then
       --  Set both local clocks to larger wakeup time and set both tasks WAITING.
      else
        -- Enqueue task on partner's accept queue and set task WAITING_TO_CALL;
      end if;
      advance_global_time;
    or
      accept ita(...) do         -- Intend to accept.
      if (a WAITING_TO_CALL partner already in accept queue) then
        -- Dequeue partner from accept queue.
        -- Set both local clocks to larger wakeup time and set both tasks WAITING.
      else
        -- Set task WAITING_TO_ACCEPT;
      end if;
      advance_global_time;
    end select;
  end loop;
end sched;

procedure advance_global_time is
begin
  if (no tasks are running) then
    if (at least one task is WAITING) then
      -- Update global clock to smallest of WAITING local clocks.
      -- Resume all tasks whose local_clocks match the new global_clock.
    else
      -- DEADLOCK; no tasks RUNNING or WAITING.
    end if;
  end if; -- No update yet; at least one task RUNNING.
end advance_global_time;
```

Figure 8. Simulation controller.

tioned primitives at entries **wait**, **itr** (intend to rendezvous), and **ita** (intend to accept). Task **sched** also handles client task initialization, which is not shown. If the desired action is "wait," then the simulation controller calculates the time at which the task should resume executing, based on the task's local clock and desired wait interval, and updates the wakeup time for that task. As long as there are still other running tasks no further action is taken; the remainder of the running tasks are allowed to continue. This is the key concept that removes any requirement of rolling back the master clock. Only when there are no more running tasks will the simulation controller examine the list of waiting processes, determine the new global clock value from the waiting task with the smallest wakeup time, and resume running all waiting tasks whose wakeup times are equal to the new global clock. The simulation controller also sets the local clocks of all resumed tasks equal to the current global clock; the tasks henceforth reference their local clocks.

The "intend to rendezvous" and "intend to accept" primitives are managed somewhat differently. Since a rendezvous requires two parties, a task indicating an intent to rendezvous without a corresponding partner task having previously indicated an intent to accept, or vice versa, is suspended and is not allowed to resume execution until the partner task issues its intent to complete the rendezvous. Once both tasks have indicated their intent to rendezvous the simulation controller updates their wakeup time to the larger of the two task local clocks and places them in a wait state. The tasks are then resumed as in the preceding paragraph. Note that tasks paired through a rendezvous are resumed at the same time due to their identical wakeup times. Because of the asymmetric nature of the Ada rendezvous in which the task issuing an accept does not know the identity of the task making the rendezvous it is necessary to queue tasks which have indicated an intent to rendezvous with a target task until that target task indicates an intent to accept. The queuing discipline is FIFO and is provided in the simulation controller.

We now describe briefly the way in which the scheduler resumes waiting tasks. This is done via a rendezvous with the **send** entry point of an element of the **agents** array (the agent task structure is shown in Figure 9). There is a copy of this agent task corresponding to each task identifier; when resuming a task, the scheduler will rendezvous with the agent and the agent will rendezvous with the task. This eliminates the need for keys in a situation where a client requests a service to be completed but cannot poll the server. (A detailed discussion on using agents in this capacity may be found in Reference 1, pp. 236-238.) For example, when the scheduler has determined that a task previously suspended on a call to **sched.wait** may continue execution because the global clock has advanced sufficiently, the scheduler gives the updated global clock value to the agent via the **send** rendezvous. Subsequently, the agent passes it on to the waiting task via the **recv** rendezvous, and the task resumes execution with an updated local clock. The use of this mechanism will be apparent in the case study.

```
task type agent is
  entry send(local_clock: in time);
  entry recv(local_clock: out time);
end agent;

task body agent is
  clock_storage: time;
begin
  loop
    accept send(local_clock: in time) do
      clock_storage :=local_clock;
    end send;
    accept recv(local_clock: out time) do
      local_clock := clock_storage;
    end recv;
  end loop;
end agent;

agents: array(1..N) of agent;
```

Figure 9. Agent task structure.

CASE STUDY

Utilizing our method of hierarchical decomposition we have generated a description of the machining cell shown in Figure 10. The manufacturing process involves machining preformed metal stock by milling, turning, and rolling threads. The cell contains two robot loading and unloading a CNC mill, a CNC lathe, and rolling and gauging machines. Both robots have two sets of grippers so that a finished part may be unloaded from a machine and a new part inserted into the same machine without the need for moving the robot between these operations.

Diagrammatical implementation

The hierarchical description comprises three levels. The first level describes the operation of the complete cell, and lists the inputs and outputs to the manufacturing cell as a whole. For example, an input is "stock," which describes the metal stock the cell takes in; and an output is "good parts," which describes a properly manufactured part which the cell puts out.

The second level provides more detail and splits the box into a control subbox plus three functional subboxes:

- Milling – A description of the first third of the manufacturing cycle, in which the first robot accepts parts from a parts presenter and causes the parts to be milled and gauged.
- Turning – A description of the second third of the manufacturing cycle in which the first robot causes the parts to be turned and gauged.
- Thread rolling – A description of the final third of the manufacturing cycle in which the second robot causes the threaded portion of the parts to be rolled and gauged.

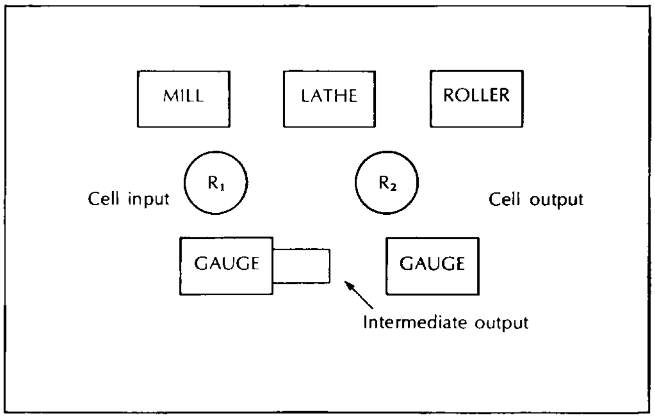


Figure 10. Machining cell.

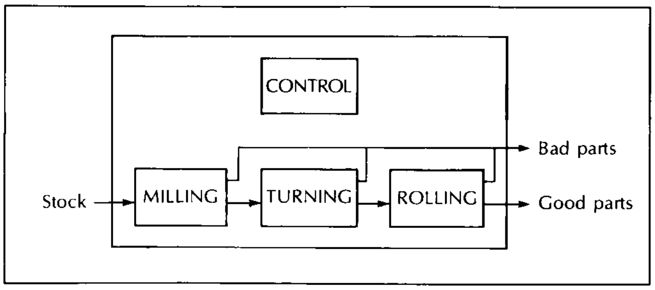


Figure 11. Diagrammatical example.

The corresponding diagrammatical decomposition is shown in Figure 11; the directed lines indicating control flow between the control and functional subboxes have been omitted for clarity. The execution sequence of these functional subboxes is determined by the control subbox so that the flow shown in Figure 11 is achieved.

The third and final level of decomposition splits each of the level 2 subboxes above into more subboxes. The number of boxes then increases beyond the scope of this paper; for example, the milling and gauging subbox is further decomposed into a level 3 control subbox and twelve terminal subboxes. The other two level 2 functional subboxes controlling the manufacturing cycle are decomposed in exactly the same fashion.

Procedural implementation

We shall illustrate the decomposition process in greater detail by considering the subbox which describes (and simulates) the acceptance of input stock by the first robot. The task that represents this subbox performs this simulation by advancing its local clock by an amount indicative of the time needed for the robot to accept the part from the parts presenter. The task also updates the simulation variables we have chosen to describe the state of each component of the cell.

Figure 12 gives the specification for the task. The specification includes a DIO list which denotes the description, inputs, and outputs of the task. The name of the hierarchical superior subbox and a list of the other subboxes at this level of decomposition are also given. Finally, the **START** and **STOP** entries matching the input and output lists are given. The corresponding task body is more detailed and is not given in a figure, but is interspersed with the following text.

First we give the task body prologue in which we define variables to be used in the task's simulation activities. In the scope of the task body **local__input__part** will hold a local copy of the stock part description, **local__output__part** will hold a local copy of the green part description, and **local__clock** gives this task's view of time. The types **stock** and **green__part** are user-defined types which help describe a part as it moves through the system, and **time** is a scalar type which has the obvious meaning.

```
task body get_stock is
  local_input_part: stock;
  local_output_part: green_part;
  local_clock: time := 0;
begin
```

If only a description were being constructed, the types **stock** and **green__part** could simply be enumerated types, each with a single possible value, since their utility in the description would lie in the implied semantics of the names chosen and in the fact that they are separate types whose usage can be checked formally by a compiler. For simulation purposes, however, it is convenient to make them record types and have objects of these types carry information relative to the parts

```
task get_stock is
  --specification identification is get_stock;
  --
  --superior is milling_and_gauging;
  --siblings are (
  --      level_3a_control, move_to_parts_presenter, move_to_mill,
  --      unload_milled_parts, load_mill, mill, move_to_gauge,
  --      unload_gauged_parts, load_gauge, gauge, move_to_parts_disposer,
  --      dispose_bad_part);
  --
  --description is
  --      Obtains a stock part from the parts presenter
  --      and places it into the lower grippers.  At this
  --      point the part is known as a green, or unprocessed,
  --      part.
  --end description;
  --
  --input list is
  --      (stock);
  --
  --output list is
  --      (green_part);

  entry START(input_part: stock);
  entry STOP(output_part: green_part);
end get_stock;
```

Figure 12. Procedural example specification.

they represent. The choice of component types for the record is strictly dependent upon the output information desired from the simulation. Our simulation output provides status information of all parts in the system at various points in time; we use these record components to store this information. Indeed, one might want to keep similar information in a real system to be carried along with each part, particularly information related to the gauging of the parts (not shown in this example) which might be used for detecting tool wear.

The record components relevant to the example are:

```
process_initiated:         boolean;  -- Indicates part has started
                                     -- through the system.
part_code:                 integer;  -- Uniquely identifies part.
parts_presenter_unloaded:  boolean;  -- Indicates part has been
                                     -- unloaded from presenter.
```

The record types in which these components appear are used in other tasks as well as get__stock. Accordingly the full definitions of the types stock and green__part do not appear in get__stock, but were made in a hierarchical superior task. For this example, one merely needs to understand that the declaration of objects of types stock or green__part associates with those objects' components of the types given above.

Returning to the example task body, we next define our task to the scheduler by reporting its activation via activate; ggp__port is an identifier which uniquely identifies this task to the scheduler. Then we output a statement indicating that get__green__parts has started processing.

```
-- Define task to sched; ggp_port identifies the task.
sched.activate(ggp_port);
put_line("get_green_parts:  start");
```

We now enter the main loop and indicate our intent to rendezvous to the scheduler by passing our task indentifier and the current value of our local clock to the sched.ita entry. We next await notification from the scheduler that we may continue; when our rendezvous partner indicates a corresponding intent to make an entry call, the scheduler gives our agents task array member the correct updated value of the global clock, and we obtain it via the recv entry. Thereupon we continue execution, having set our local clock to the new global clock value, and perform our half of the intended rendezvous by executing the accept statement. When the control task rendezvous with us, it will pass an input__part; we save it in the local variable local__input__part for our use.

```
loop

  --  Indicate intent to accept rendezvous.
  sched.ita(ggp_port,local_clock);
  -- Receive updated local clock from sched when OK to continue.
  agents(ggp_port).recv(local_clock);
  -- Perform accept.
  accept START(input_part: stock) do
    local_input_part := input_part;
  end START;
```

At this point we have been called with a new input part. We output a comment to that effect and update the process__initiated and part__code record components associated with the output part.

```
-- Record event at current local time.
comment("Accepting part from parts presenter");
-- Update part description variables.
local_output_part.process_initiated := true;
local_output_part.part_code := local_input_part.part_code;
```

We next need to wait for an amount of time indicative of the time required to obtain the part from the parts presenter. We simulate this by asking the scheduler to delay us for present__part__time units of time, again passing our task identifier and local clock, and again receiving a new local clock value when the scheduler has determined that we may proceed. For localization and ease of modification purposes, all explicit time values such as present__part__time are defined in the task at the top of the hierarchy and thus do not appear in the local declarations of get__green__parts.

```
-- Indicate wait.
sched.wait(ggp_port,local_clock,present_part_time);
-- Receive updated local clock from sched when OK to continue.
agents(ggp_port).recv(local_clock);
```

Having delayed by the proper amount of time, we next set the parts__presenter__unloaded flag associated with the output part.

```
-- Set "part unloaded" attribute.
local_output_part.parts_presenter_unloaded := true;
```

We now perform a rendezvous with the control task to pass back the output part description via the STOP entry. Finally, we loop back to the START accept block to await the next invocation, and the task body ends.

```
    -- Indicate intent to rendezvous.
    sched.ita(ggp_port.local_clock);
    -- Receive updated local clock from sched when OK to continue.
    agents(ggp_port).recv(local_clock);
    -- Perform accept.
    accept STOP(output_part: out green_part) do
      output_part := local_output_part;
    end STOP;
  end loop;
end get_green_parts;
```

Simulation output

The output provided by the execution of the simulation is shown in Figure 13. It consists of a time-ordered series of event reports and additional information about the state of the simulation. Lines of the form "task__name: processing" indicate that the task identified by the task__name has just received a new set of inputs and is starting to perform the function outlined in its DIO list. Every control task in the description indicates the start of a cycle in this manner. Lines of the form "time: event" indicate that event occurred at time on the global clock. For example, the mill was started 520 time units after the start of the simulation at time zero. Thus these lines give a time-ordered view of the simulation.

In addition, the level 2 control tasks output a block of information at the completion of every cycle which lists the contents of the various stations in the cell. In particular, the contents of the stations in the mill and lathe portion of the manufacturing cell are listed at the end of the milling and turning cycle. The part residing in each station is listed along with the current description record associated with the part. The description records have been omitted due to space considerations.

The simulation is executed until the desired amount of data has been obtained about the manfacturing cell. It is a simple matter to change the time required to perform the various activities and obtain multiple simulation runs. It is only slightly more difficult to change the model by modifying the description and the affected task bodies and to compare results for different cell configurations.

```
level_3a_control:  processing
  449:  Moving robot 1 to parts presenter
  459:  Accepting part from parts presenter.
  461:  Moving robot 1 to mill.
  464:  Lathe gauging complete.
  464:  Lathe gauge has accepted part.
  464:  Lathe gauge output full; stopping mill and lathe cells.
  516:  Milling complete.
  516:  Unloading previously processed part from mill.
  518:  Loading green part into mill.
  520:  Starting mill.
  520:  Moving robot 1 to mill gauge.
  526:  Lathe milling complete.
  530:  Unloading previously processed part from mill gauge.
  532:  Loading previously processed part into mill gauge.
  534:  Starting mill gauge.
level_3b_control:  processing
  534:  Moving robot 1 to lathe.
  549:  Unloading previously processed part from lathe.
  552:  Loading previously processed part into lathe.
  554:  Mill gauging complete.
  554:  Mill gauge has accepted part.
  555:  Starting lathe.
  555:  Moving robot 1 to lathe gauge.
  570:  Loading previously processed part into lathe gauge.
  573:  Starting lathe gauge.
```

Figure 13. Simulation output.

EVALUATION

During the several months spent in developing our case study – a simplified version of which is presented here – we learned several things with respect to writing software in Ada. Although they are not necessarily new insights, we pass them on here to those considering writing their first application in Ada.

The Ada language is very strict. In particular, the concept of strong typing leads to a developmental cycle in which many passes are required to "get the program through the compiler," particularly if one is not familiar with the Ada language. In return, we find that once an Ada program can be correctly compiled, it usually executes correctly the first time it is run. This is a favorable contrast to typeless languages such as BASIC, FORTRAN, and C, and moves a large part of the program debugging effort from the execution to the compilation phase. Anyone familiar with the problems of debugging high-level software with low-level tools – particularly a compile-download-test environment where many people must compete for the development hardware – can appreciate the concept of eliminating as many errors as possible *before* the debugging phase.

It appears that one must have a great deal of knowledge of the Ada language before one can write reasonable programs with it. For example, one needs to know about generics, instantiations thereof, subprogram overloading and the intricacies of TEXT_IO before the value of an integer can be output. One needs to read quite a bit about string types, the new operator, and unconstrained arrays before one understands how to copy a string from one place to another. Most of these problems can be solved through better education, learning by doing, and examination of the work of others; what this means to novice Ada users, however, is a greater time investment during the learning phase than required by other traditional programming languages.

The strong typing of Ada presents some problems. A fairly common situation that arises when developing software according to the client-server model is the need for the server to call back the client sometime after a client request has been made. For example, we needed sched to call back waiting tasks when it was time for them to resume execution. In a less strongly-typed language, an address is passed by the client when making a request. The server stores the address and calls it at some later time when the client request has been filled. (This address is sometimes called a *delay-return address*.) This is a simple albeit dangerous and error prone solution which is not available in Ada, which expressly forbids the calling of passed addresses. Thus it is necessary to use a scheme such as Barnes' agents, which effectively doubles the number of rendezvous required over the delay-return model. It can also be argued that delay-return calls present real problems in practice and can lead to bugs that can be nearly impossible to find or reproduce, and programs are better off without them; in that case, work needs to be done in finding efficient alternatives to the delay-return model.

One of the major reasons for investigating the use of Ada in developing descriptions was the notion that strong typing would help in ensuring consistency among the descriptive components. While this turned out to be a true statement, we also became aware of the fact that the level of type checking was to a great extent definable by the modeler, perhaps too much so. We realize that any large system description involves a formidable level of detail, and that a good system description language would not allow a large degree of variation in the descriptions written in it. Our description system, since it is based heavily on Ada, allows a wide range of descriptive styles. The way in which we intended the system to be used is as shown in the case study: different types for different objects. In practice, this notion results in a large number of nearly duplicate object descriptions, with each task copying the contents of the input object to the output object (since they are of different types, a single object-to-object copy cannot be done, and each element must be separately copied) and then updating the output object with the task's own actions. This allows maximal type checking but becomes tedious. If one makes all objects the same type, and simply copies the input to the output object, then a significant amount of type checking is lost. A variation of this scheme is to use the same object for input and output. We wrote our case study this way. While it must be noted that a complete and detailed cell description utilizing any methodology is going to be voluminous, it seems that we are torn between tedium and type checking, and it appears that a solution to this dilemma lies in the automatic generation of a large part of the description. This would free the modeler from rewriting the same code dozens or hundreds of times, and would discourage the programming simplifications that erode the benefits of type checking.

CONCLUSIONS

We have shown how the well-known idea of hierarchical decomposition can be applied to the problem of supplying detailed descriptions of an arbitrary manufacturing cell, and how a suitable choice of a procedural decomposition language makes possible the simulation of a manufacturing cell so described.

A further area of investigation would involve defining and providing a procedural decomposition language that could generate Ada-based descriptions and simulations directly. Another possibility would be the investigation of the ramifications of replacing the terminal function descriptions with real control software to supervise directly the operation of an actual manufacturing cell.

ACKNOWLEDGMENT

We would like to acknowledge the contributions of A.W. Naylor to the early phases of this effort.

REFERENCES

1 BARNES, J. G. P.
Programming in Ada. 2d ed. Addison-Wesley, London (1984).

2 BÉZEVIN, J. and IMBERT, H.
"Adapting a Simulation Language to a Distributed Environment." *Proceedings, 3rd International Conference on Distributed Computing Systems.* Miami, Florida (1982), 596-603.

3 CHRISTOPHER, T.; EVENS, M.; GARGEYA, R. R.; and LEONHARDT, T.
"Structure of a Distributed Simulation System." *Proceedings, 3rd International Conference on Distributed Computing Systems.* Miami, Florida (1982), 584-589.

4 LAMPORT, L.
"Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM* 21:7 (1978), 558-565.

5 ROSS, D. T. and SCHOMAN, K. E., Jr.
"Structured Analysis for Requirements Definition." *IEEE Transactions on Software Engineering* SE-3:1 (1977), 6-15.

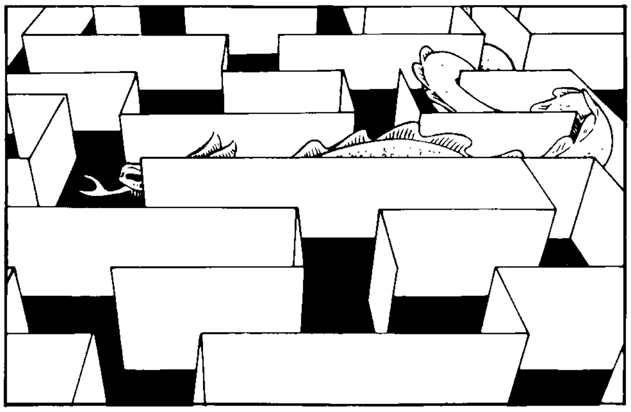
6 SAMMET, J. E.; WAUGH, D. W.; REITER, R. W., Jr.
"PDL/Ada – A Design Language Based on Ada." *ACM '81 Conference Proceedings* (1981), 217-229.

7 SHEFFIELD, J. R. and LINDLEY L.
"Ada Programming Design Languages – A Report on Their Status." *Thirty-fifth IEEE National Aerospace and Electronics Conference.* Dayton, Ohio (May 1983), 968-972.

8 TEICHROEW, D. and HERSHEY E. A., III
"PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering* SE-3:1 (1977), 41-48.

9 THE UNITED STATES DEPARTMENT OF DEFENSE
Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983. Washington, D.C. (February 1983).

IS THERE A DRAGON IN THE MAZE?



FIND IT BEFORE IT FINDS YOU.

You know the bugs are in there, what you need is the most efficient and cost effective means available to eliminate gremlins in your system. You need an integrated simulation analysis package that is adaptable to your specific needs. **SIMPLE_1** provides an integrated modeling environment featuring:

- **An English like modeling language: no computer-gook.**
- **A full screen text editor.**
- **Real time character animation of simulation results.**
- **Collection & analysis of real world and simulation data.**
- **File utilities to avoid the floppy shuffle.**
- **No special hardware requirements.**

With **SIMPLE_1's** integrated modeling environment debugging models is a snap. The compiler locates an error and calls the editor to return you to the problem area so your programs are debugged in record time.

Character graphics features allow you to animate simulation results and plot system variables easily. Need to feed data to the model from the keyboard or from disk files? No problem. **SIMPLE_1** supports READ/WRITE operations associated with disk files, monitor, keyboard you name it. Hardware accessing routines allow you to use your PC to collect data in real time . . no more stop watches and data entry sheets!

Having a problem sorting out your manufacturing parts flow problems? **SIMPLE_1** was designed with manufacturing in mind! Modeling parts flows through assembly & repair systems are a breeze with **SIMPLE_1's** advanced simulation language concepts.

At last a simulation system so intelligent it's **SIMPLE_1**.

Available for IBM PC, XT, AT and true Compatibles. Requires 128K memory, HARD DISK NOT REQUIRED. For further information contact Sierra Simulations & Software:

SIERRA Simulations & Software
303 ESTHER AVE
CAMPBELL CA 95008
(408) 378 6374

"IBM & "PC" are trademarks of International Business Machines Corp.