Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

HIERARCHICAL DECOMPOSITION AND SIMULATION OF MANUFACTURING CELLS

Charles J. Antonelli
Richard A. Volz
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
(313) 764-4343

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[1] programming language in conjunction with established diagrammatical decomposition methods. 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.

Keywords

System Description Language, hierarchical decomposition, functional description, manufacturing cell simulation.

INTRODUCTION

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 whose operations and interactions must be uniformly described, such as various types and quantities of programmable controllers, CNC machines, material handling and storage systems, robots, and a host of other general- and special-purpose equipment. Each such unit 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 require differing error recovery strategies.

Manufacturing cells make parts. 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 units 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 functional description of any manufacturing cell should possess at least the following attributes:

[1] Ada is a registered trademark of the U. S. Government Ada Joint Program Office.

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.

Consistency.
The constituent parts of the functional description must be consistent with each other. Rate and protocol must be consistent from sender to receiver on a point-to-point data link; parts output by a unit must correspond to the input requirements of a succeeding unit.

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.

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. However, it is difficult to combine these descriptions into a coherent set of specifications at the manufacturing cell level, particularly one amenable to simulation.

One way of achieving a uniform set of functional descriptions is through a *system description language* which can completely describe a manufacturing cell at a suitable level of detail. Such system description languages are in widespread use. For example, IBM's PDL[1] is a procedural high-level language used in writing software specifications. As another example, the ISDOS Project's[2] PSL/PSA is a database-oriented high-level language used in describing arbitrary information systems. Through use of such a language it is possible to define a regular descriptive methodology that can be applied equally to a broad class of systems.

In this paper we examine the use of the United States Government language Ada[3] as the basis for formally describing manufacturing cells. Descriptions at multiple levels of detail are obtained by a hierarchical decomposition technique. A mapping is defined between a diagrammatical representation of a hierarchical decomposition and a set of Ada tasks. A method of transforming the Ada description into a simulator of the system is also described.

It is well known that decomposing a difficult problem into several simpler subproblems allows a solution to be obtained when direct methods fail. The problem is broken into several subproblems, the subproblems are solved, and the problem solution is defined in terms of the subproblem solutions. If the subproblems themselves are difficult, they are broken into smaller subproblems. This decomposition continues until the individual subproblems can be solved.

We apply this technique to the problem of generating functional descriptions of manufacturing cells, utilizing two complementary descriptive formats. In the *diagrammatical decomposition* format, 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 allows the reader to obtain a quick, intuitive understanding of the manufacturing cell being described. In the *procedural decomposition* format, we present an equivalent functional description written in a procedural description language based on the Ada programming language. The hierarchical decomposition is shown as a series of nested packages, and task rendezvous describe the data and control flow. The procedural decomposition is much more detailed and gives the reader a complete functional decomposition of the cell being described.

This 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.

Both methods are illustrated in greater detail below.

DIAGRAMMATICAL DECOMPOSITION

The basic unit of diagrammatical decomposition is the *box* (see Figure 1). There are 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. This procedure in which a box operates on its inputs to yield its outputs is central to our hierarchical decomposition scheme. 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. Since we are describing the total operation of the cell and we are describing the function of the cell only, we do not distinguish at this level between physical objects and data objects which are operated on by the system; this partitioning occurs only at the bottommost level of decomposition.

In order to perform the hierarchical decomposition, we also consider the box to be an entity which encloses a particular level of decomposition. In this light, the exterior and interior of a box relate to the decomposition operation in the following way.

The *exterior* of a box describes the current level of decomposition. This description takes the form of a *DIO block*, which we define to be a common descriptive unit consisting of a description of the function performed by, a list of inputs to, and a list of outputs from the box representing the current level of decomposition. In other words, the DIO block of the box exterior completely describes the current level of decomposition to the reader. In Figure 1, the depicted box 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 assembly of part 1 and part 2 to produce part 3. This level of decomposition does not describe how the assembly is to be performed, only that it is to take place.

The *interior* of a box contains a collection of *subboxes*. Each subbox is described by a DIO block as stated previously. The collection of subboxes forms the next level of decomposition; their DIO blocks, taken together, form the same functional description as the DIO block of the enclosing box, the critical difference being that the subbox DIO blocks are more detailed.

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 description of the enclosing box's DIO block.

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 description of the enclosing box. The dashed lines represent inputs and outputs between subboxes which are local to the box interior, while the solid lines represent inputs originating from and outputs destined

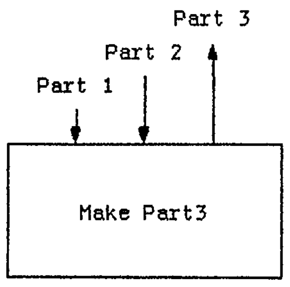

Figure 1: Box Exterior

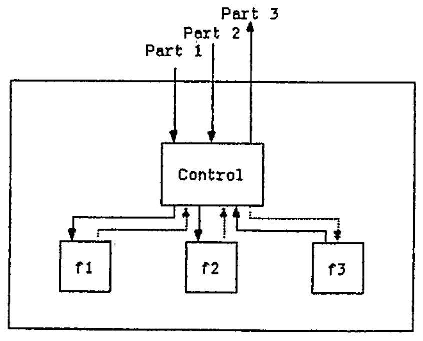

Figure 2: Box Interior

for the box exterior. While it is possible to think of f_1 and f_2 happening concurrently, f_3 must wait for them to complete before proceeding. This flow of control is determined by the control subbox, as indicated by the broken lines.

It is easy to see how one recursively descends 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 DIO block specifies the operation "close gripper on robot arm 1" is probably not amenable to further decomposition. We can view the successive decompositions of a cell as a tree which represents a collection of descriptions at different levels of detail. The leaf nodes of each subtree whose root is identical to the root of the tree itself correspond to a single description.

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 do not in general have physical counterparts in the cell itself. At some level of decomposition, however, the functional boxes should correspond to physical entities, portions thereof, or control program procedures, and the inputs and outputs are associated directly with the terminal subboxes.

PROCEDURAL DECOMPOSITION

The diagrammatical decomposition method provides an elegant way of decomposing a manufacturing cell. However, in its present form it does not provide much information about the timing and synchronization of interactions between subboxes. Secondly, it is difficult to represent a great amount of detail in a concise manner. Finally, it is not immediately obvious how to simulate the actual cell directly from the diagrammatical decomposition. To deal with these problems we use a *procedural decomposition language* to describe the hierarchical decomposition.

Procedural Decomposition Language

We represent the functional units of the procedural decomposition as Ada tasks, just as we represent the functional units of the diagrammatical decomposition as boxes. We have chosen tasks instead of procedures or functions because a task is a more general construct. Tasks can execute in parallel and thus provide a more natural description of simultaneous events than sequential constructs do.

At a given level of decomposition, a task 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 other tasks. Second, the task must provide a description of the function it represents. In general, these two classes of information are intermixed inside the task.

Whenever a step in the decomposition is made, a task is replaced by a set of tasks whose combined input, output, and synchronization characteristics subsume those of the original task. This expressly allows the set of tasks to exchange input, output, and synchronization information among themselves. In Figure 3, task A is replaced by tasks B, C, and D. Task B is also shown to output some local information to task C.

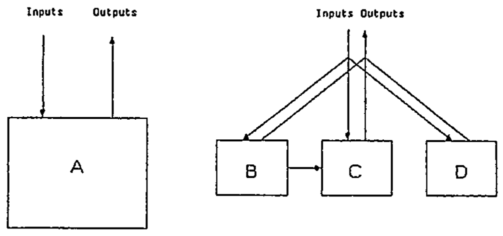

Figure 3: Results of Decomposition Step

The inputs and outputs destined for the original task must now be dispersed to and collected from the set of tasks which replaced it. 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, we introduce the artifice of an interface task which serves as a buffer between the other tasks in the description and the tasks of the current level of decomposition. It does nothing more than present its inputs to these tasks and present their outputs at its outputs. In Figure 4, task A is replaced by tasks B, C, and D as before, but task A' is introduced to provide an interface between them and the rest of the description.

The Ada language provides a passive encapsulation mechanism called a package. Tasks can be grouped inside a package and isolated from or made visible to other tasks outside the package as desired. We place the tasks generated by the decomposition step in 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. In Figure 5, package P surrounds tasks B, C, and D.

The process of decomposition continues by performing decomposition steps on tasks and encapsulating the results of each such decomposition in a package. This results in a tree of packages as seen in Figure 6. At some point it becomes unnecessary to decompose a

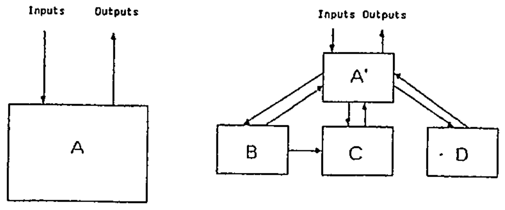

Figure 4: Interface Task

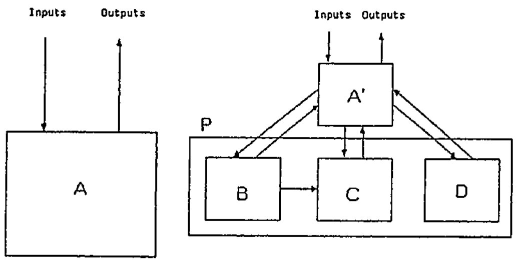

Figure 5: Task Partitioning

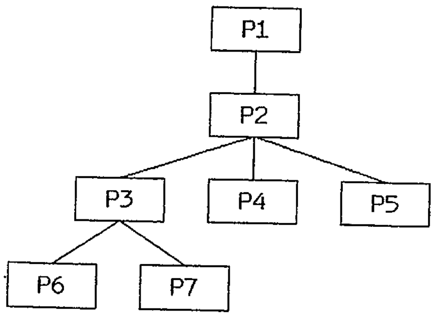

Figure 6: Package Tree

task any further; we call such a task a *terminal* task.

Simulation Considerations

A simulation of a manufacturing cell described in the procedural description language can be realized by observing that the description of inter-task relations using Ada tasking constructs as the basis for the description actually provides the basis for simulation control software which supports a process oriented simulation scheme.

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 passing of time while 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. In this fashion a simulation of the manufacturing cell can be easily obtained.

Sometimes it is also possible to replace the terminal function descriptions with real control software to supervise directly the operation of an actual manufacturing cell. Consider the different types of entities that can be represented by the task inputs and outputs. Actual parts, such as those being constructed by the manufacturing cell, are represented by certain types of data objects. Other data objects represent control signals required to operate the cell. Still other objects represent pieces of data essential to the execution of the simulation. It is important to partition the inputs to tasks representing real control software in such a manner that a task receives only those inputs which it could logically receive in the actual manufacturing cell environment.

Procedural Decomposition Language Implementation

In order to maintain a close relationship between the diagrammatical and procedural decompositions, we have standardized the usage of Ada in the procedural decomposition language in the following way.

As previously stated the functional units of the procedural decomposition are represented as Ada tasks. Each such task has the form shown in Figure 7. A task is identified according to the following naming convention. Each task name consists of the identifier t followed by a number of subscripts, e.g. $t_{i,j,k}$. The number of subscripts indicates the *level* of decomposition at which the task resides. The value of a subscript differentiates between tasks at a particular level. For example, t_0 indicates the first task of the decomposition which resides at level 1. After a step in the decomposition is made, $t_{0,2}$ indicates the third task of the set which replaced task t_0 . This process continues to an arbitrary number of levels. For notational brevity we shall use t_i to represent a task at an arbitrary level of decomposition.

The task specification contains a DIO block and a list of entry points to the task. The DIO block performs the same function for tasks as for boxes; that is, it describes the function, inputs, and outputs of the task. It is implemented in the form of Ada comments. The inputs to the task are represented by I_i and outputs from the task are represented by O_i entries in the DIO block. Each entry consists of a list of items input to or output from the task.

Each task has at least two entry points which are invoked by the Ada *rendezvous* mechanism: *start*, at which point the inputs required by the task as listed in the DIO block are accepted, and *stop*, at which point the outputs generated by the task as listed in the DIO block are returned. Additional entry points accessed by other

```
task $t_i$ is
 -- DIO $t_i$:
 --  description is ...
 --  inputs are $I_i$ .
 --  outputs are $O_i$ .
 entry start ($I_i$) ;
 entry stop ($O_i$) ;
 entry ...
end $t_i$ ;

task body $t_i$ is
begin
 loop
   accept start ($I_i$) do
     $LOCAL\_I_i$ := $I_i$ ;
   end start;

   $LOCAL\_O_i$ := $F_i(LOCAL\_I_i)$ ;

   accept stop ($O_i$) do
     $O_i$ := $LOCAL\_O_i$ ;
   end stop;
  end loop;
end $t_i$ ;
```

Figure 7: Task Implementation

tasks in the same package are also listed in the specification.

The task body contains statements which realize the function of the box as described in the DIO block. The task body is in the form of an infinite loop. The *start* rendezvous accepts the inputs to the task and makes a local copy of these inputs in $LOCAL\_I_i$. This is done because the actual parameters are accessible only in the body of the accept statement. The task then performs its function as described in the DIO block, indicated by the function F_i which represents a series of executable statements operating on the local copy of the inputs and yielding a local set of outputs in $LOCAL\_O_i$. The task may rendezvous with other tasks in the same package in order to carry out F_i. Finally, the *stop* rendezvous returns the outputs of the task by copying them from the local outputs. Following this, the task loops to accept a new set of inputs. This repetitive sequence of accepting inputs, performing the function, and returning outputs is called a *cycle*.

When a step in the decomposition is made, the task shown in Figure 7 is replaced by the one shown in Figure 8, and a set of tasks named $t_{i,0}$ through $t_{i,n}$ is generated. In general, n will be different for each decomposition step. Figure 9 shows the general form for each of these tasks by describing task $t_{i,j}$, where $0 \le j \le n$. The relationship between the original task and the interface and generated tasks is governed by the three relations

$$F_i \equiv \bigcup_{j=0}^{n} F_{i,j}$$

$$I_i \subseteq \bigcup_{j=0}^{n} I_{i,j}$$

$$O_i \subseteq \bigcup_{j=0}^{n} O_{i,j}$$

which show that the collective function of the generated tasks is identical to the function of the original task and that the collective input's and outputs of the generated tasks subsume the inputs and outputs of the original task.

The set of tasks is enclosed in a package named p_i as shown in Figure 10. The naming convention for packages is identical to that for tasks; a package name is given the same subscripts as the that of the original task. There are two parts to an Ada package as shown in Figure 10: the specification and the body. Everything that is to be accessible to the exterior of a package must be listed in the package specification; everything else in the package body is hidden from view. Thus the package specification only contains 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 interface task is identical to the original task except that the statements representing the function F_i are replaced by *start* and *stop* rendezvous calls to the generated task $t_{i,0}$. We have arbitrarily given $t_{i,0}$ the job of sequencing the execution of the remaining tasks in p_i ; therefore we call it the *control task*. The control task body is shown in Figure 10 and is similar to the original task; the difference is that the function performed, $F_{i,0}$, consists primarily of *start* and *stop* rendezvous calls with the rest of the tasks in p_i . Each of these rendezvous passes the subset $LOCAL\_I_{i,j}$ of local inputs $LOCAL\_I_i$ required by $t_{i,j}$ and returns the subset $LOCAL\_O_{i,j}$ of local outputs $LOCAL\_O_i$ generated by $t_{i,j}$. This is shown in Figure 10, where the *subset* function indicates the appropriate subset.

There is a similarity in form between the generated task $t_{i,j}$ and the original task t_i . This similarity allows further decomposition steps to be taken by replacing $t_{i,j}$ with an interface task and by generating a set of replacement tasks enclosed in a new package $p_{i,j}$. This process of decomposition continues until the terminal level of the decomposition is reached. At this level each task contains statements to perform or describe the

```
task $t_i$ is
 -- DIO $t_i$:
 --  description is ...
 --  inputs are $I_i$ .
 --  outputs are $O_i$ .
 entry start ($I_i$) ;
 entry stop ($O_i$) ;
end $t_i$ ;

task body $t_i$ is
begin
 loop
   accept start ($I_i$) do
     $LOCAL\_I_i$ := $I_i$ ;
   end start;

   $t_{i,0}$.start($LOCAL\_I_i$) ;
   $t_{i,0}$.stop($LOCAL\_O_i$) ;

   accept stop ($O_i$) do
     $O_i$ := $LOCAL\_O_i$ ;
   end stop;
 end loop;
end $t_i$ ;
```

Figure 8: Interface Task Implementation

```
task $t_{i,j}$ is
 -- DIO $t_{i,j}$:
 --  description is ...
 --  inputs are $I_{i,j}$ .
 --  outputs are $O_{i,j}$ .
 entry start ($I_{i,j}$) ;
 entry stop ($O_{i,j}$) ;
 entry ...
end $t_{i,j}$ ;

task body $t_{i,j}$ is
begin
 loop
   accept start ($I_{i,j}$) do
     $LOCAL\_I_{i,j}$ := $I_{i,j}$ ;
   end start;

   $LOCAL\_O_{i,j}$ := $F_{i,j}$($LOCAL\_I_{i,j}$) ;

   accept stop ($O_{i,j}$) do
     $O_{i,j}$ := $LOCAL\_O_{i,j}$ ;
   end stop;
 end loop;
end $t_{i,j}$ ;
```

Figure 9: Generated Task Implementation

```
package $p_i$ is
 task $t_{i,0}$ is

 end $t_{i,0}$ ;
end $p_i$ ;
package body $p_i$ is
 -- DIO $t_{i,0}$
 task body $t_{i,0}$ is
 begin
  loop
   accept start ($I_i$) do
    $LOCAL\_I_i := I_i$ ;
   end start;


   $t_{i,j}.start(subset(LOCAL\_I_i))$ ;
   $t_{i,j}.stop(subset(LOCAL\_O_i))$ ;


   accept stop ($O_i$) do
    $O_i := LOCAL\_O_i$ ;
   end stop;
  end loop;
 end $t_{i,0}$ ;



 task $t_{i,j}$ is

 end $t_{i,j}$ ;
 -- DIO $t_{i,j}$
 task body $t_{i,j}$ is
 begin

 end $t_{i,j}$ ;



end $p_i$ ;
```

Figure 10: Generated Tasks with Enclosing Package

functions that can no longer be subdivided. Thus the set of all terminal tasks plus all control tasks contains the entire functional description of the manufacturing cell. By including appropriate simulation statements at the terminal level it is possible to generate a simulation of the operation of the cell.

SIMULATION METHODOLOGY

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 and a list of processes that are scheduled to run at various simulated times. The simulation scheduler, when informed that the currently executing process wishes to relinquish control, adds the previously active process to the list of waiting processes, chooses the process with the smallest simulation clock value, and executes it. Since only one process executes at a given time, it is never necessary to roll back simulated time in the course of a simulation.

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 are 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 in an appropriate manner to avoid this rollback.

We have developed such a mechanism for use with our description system. As previously stated Ada tasks represent the functional units of the procedural decomposition. Possibly executing in parallel, these tasks must mutually manage the master simulation clock. We supply each task with a local simulation clock in addition to the master, or 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 master clock whenever the task invokes one of the primitives explained below.

Each of the tasks must be able to advance its local clock and rendezvous with other tasks as required to carry out the simulation. The following four primitives are sufficient:

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. In this case both the invoker's local clock and the local clock of the task having executed 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 accept a rendezvous with another task. In this case both the invoker's local clock and the local clock of the task having executed the corresponding rendezvous may require updating. When the rendezvous takes place, both tasks will have their local clocks set to the larger of the original local clocks.

Relinquish.
: A task wishes to relinquish control without specifying a time at which it is to resume. Execution of the task will resume at a future time after other tasks have been given a chance to run; its local clock will be set equal to the new global clock. Inclusion of this primitive is necessitated only by a lack of multitasking support in our current Ada run time system.

The executing tasks are managed as follows. At any given instant there are a number of executing tasks as well as a number of tasks waiting to execute at specific times. Each such task is called a *client* task and is described by a *state*, which identifies whether the task is running or in one of several wait states, and a *wakeup_time*, which gives the global clock value at which time the task wishes to resume running. There is also a *simulation controller* which serves as a scheduler for the client tasks and contains entry points, in the form of accept statements, for each of the actions listed above.

Whenever a task needs to perform one of the four actions, it performs a rendezvous with the simulation controller which changes the state of the task from running to a wait state. 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_times 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 queueing discipline is FIFO and is provided in the simulation controller.

Finally, the "relinquish" primitive places the task in an indefinite wait state. When the simulation controller next updates the global clock the task will be resumed with its local clock set to the new global clock. The wakeup time of the task is undefined and plays no part in the calculation of the new global time.

CASE STUDY

Utilizing our method of hierarchical decomposition we have generated a description of a machining cell which is shown in Figure 11. The manufacturing process involves machining preformed metal stock by milling, turning, and rolling threads. The cell contains two robots loading and unloading a CNC mill, CNC lathe, and rolling and gaging 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. The mill and lathe occasionally require the first robot to exchange dull tools for sharp ones; the tool carriers are similar in size to the parts and may be handled with the same grippers.

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

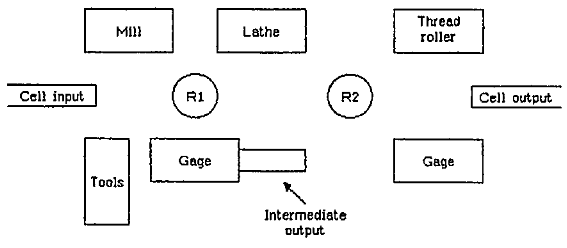

Figure 11: Machining Cell

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 five functional subboxes:

Milling and gaging.
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 gaged.

Turning and gaging.
A description of the second third of the manufacturing cycle in which the first robot causes the parts to be turned and gaged.

Thread rolling and gaging.
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 gaged.

Mill tool change.
A description of the mill tool changing procedure, which is required after a given number of parts have been milled.

Lathe tool change.
A description of the lathe tool changing procedure, which is required after a given number of parts have been turned.

The corresponding diagrammatical decomposition is shown in Figure 12, where 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. Note that the milling and turning portions of the part cycle must be performed sequentially in the order stated since both the mill and lathe are served by the same robot. This restriction does not apply to the thread rolling portion of the cycle since it is served by the second robot. It is natural, therefore, to write the control subbox as two independent tasks, each of which controls one of the two robots. Further, the tool change operations and the machining operations are mutually exclusive, and the control subbox must prevent the milling and turning control tasks from executing while any tool changes are in progress.

The third and final level of decomposition splits each of the first three level 2 subboxes above into more subboxes. The tool change cycle is not further decomposed so that level 2 represents the terminal level for tool change portion of the description. This

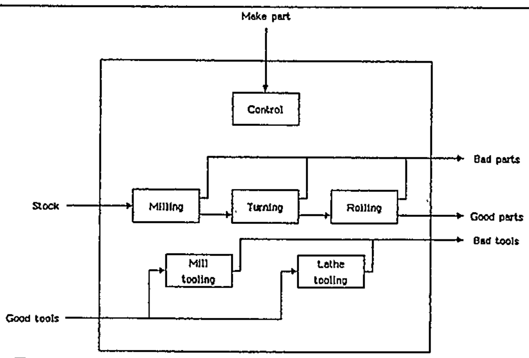

Figure 12: Sample Diagrammatical Decomposition

illustrates that portions of the description may be more detailed than others depending on the needs of the modeler. For instance, the milling and gaging subbox is further decomposed into a level 3 control subbox and twelve terminal subboxes. One of these subboxes simulates the acceptance of input stock by the first robot. This is simulated through the task that represents this subbox which advances its local clock by an amount of time indicative of the time needed for the robot to accept the part from the parts presenter and updates some values in a data structure to indicate that a new part has been obtained. The task representing this subbox is shown in Figure 13.

```
--specification identification is get_green_parts;
--
--decomposition identification is f_3_3;
--decomposition level is 3;
--specification type is functional;
--superior is milling_and_gaging;
--siblings are (
--      level_3a_control,
--      move_to_parts_presenter,
--      move_to_mill,
--      unload_milled_parts,
--      load_mill,
--      mill,
--      move_to_gage,
--      unload_gaged_parts,
--      load_gage,
--      gage,
--      move_to_parts_disposer,
--      dispose_bad_part);
--
--description is
--      Obtains two green parts from parts presenter,
--      placing them into the two lower grippers.
--end description;
--
--input list is
--      (stock);
--
--control list is                 -- Part of input list
--      (make_parts);             -- expanded for clarity.
--
--output list is
--      (green_parts);

 task body get_green_parts is
  local_entity_access: entity_record_access;
  part_number: natural := 1;
  local_clock: time := 0;
 begin
  -- Define task to sched (ggp_port identifies the task).
  sched.activate(ggp_port);
  put_line("get_green_parts: start");
  loop

   -- Indicate intent to rendezvous.
   sched.ita(ggp_port,local_clock);
   -- Receive updated local clock from sched
   -- when OK to continue.
   port_ggp.recv(local_clock);
   -- Perform accept.
   accept start;

   -- Record event at current local time.
   comment("Accepting part from parts presenter");
   -- Generate new entity (part).
   local_entity_access := new entity_record_type;
   local_entity_access.part_description.process_initiated
    := true;
   local_entity_access.part_description.part_code
    := part_number;
   part_number := part_number + 1;

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

   -- Set "part unloaded" attribute.
   local_entity_access.part_description
    .parts_presenter_unloaded := true;

   -- Indicate intent to rendezvous.
   sched.ita(ggp_port,local_clock);
   -- Receive updated local clock from sched
   -- when OK to continue.
   port_ggp.recv(local_clock);
   -- Perform accept.
   accept stop(output_entity_access:
    out entity_record_access)
   do
    output_entity_access := local_entity_access;
   end stop;
  end loop;
 end get_green_parts;
```

Figure 13: Sample Task

The DIO block lists the description, inputs, and outputs of the task.

The other two level 2 functional subboxes controlling the manufacturing cycle are decomposed in exactly the same fashion. The milling and turning portion of the cell may produce parts at a faster or slower rate than the thread rolling portion of the cell; a bounded buffer has been provided to model an intermediate part storage unit. The simulation is capable of stopping the first robot when the capacity of intermediate storage is exceeded, and of starting it again when the number of parts in storage has been reduced.

The output provided by the execution of the description is shown in Figure 14. 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 *task_name* has just received a new set of inputs and is starting to perform the function outlined in its DIO block. 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 24 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 attributes of the part. Attributes help describe the state a part is in at a given time during the manufacturing process; for example, the condition of being milled is an attribute. If a part possesses an attribute a corresponding indicator is set true, otherwise it is set false. In Figure 14, the mill is shown to contain a part on which processing has been initiated (PI), which has been unloaded from the parts presenter (PPU), and which has been loaded into the mill (ML). The part does not possess any other attributes at this stage of the manufacturing cycle. The rest of the stations are shown to be empty.

The description is executed until the desired amount of data has been obtained about the manufacturing 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.

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 can generate Ada-based descriptions and simulations directly.

ACKNOWLEDGEMENT

We gratefully acknowledge the contributions of A. W. Naylor to this effort.

References

1. Sammet, Jean E., Douglas W. Waugh, and Robert W. Reiter, Jr., "PDL/ADA - A design Language Based on Ada," *Proc. ACM Annual Conference*, (November 1981).
2. *Problem Statement Language (PSL) / Problem Statement Analyzer (PSA) User's Reference Manual (PSA Version A5.2)*, Department of Industrial and Operational Engineering, The University of Michigan , Ann Arbor, Michigan (July 1982).
3. *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A-1983)*, Ada Joint Program Office, OUSD(R&E), Washington, DC 20301 (February 17, 1983).

```
level 3a control   processing
level 3c control   processing
   0:  Moving robot 1 to parts presenter.
   0.  Moving robot 2 to lathe gage acceptor.
  10:  Accepting part from parts presenter.
  12:  Moving robot 1 to mill.
  22.  No previously processed part in mill.
  22:  Loading green part into mill.
  24:  Starting mill.
  24.  Moving robot 1 to mill gage.
  34:  No previously processed part in mill gage.
  34:  No previously processed part to load into mill gage.
  34:  Mill gage empty.
lathe turning and gaging:  processing
level 3b control:  processing
  34:  Moving robot 1 to lathe.
  49:  No previously processed part in lathe.
  49:  Nothing to load into lathe.
  49:  Lathe empty
  49:  Moving robot 1 to lathe gage.
  64:  No previously processed part to load into lathe gage.
  64:  Lathe gage empty
----- PART NAME ---- PART ------ MILLING ------ ----- TURNING ----- --------- ROLLING ---------
                     CODE P P M M   M M M M M   M M M M L T   T T L L L   L L L L L T T   T T T T T   T T T G B   P P
                          I P L C   T R U G G   G G G D L C   T R U G G   G G D A A R R   R R R G G   G G G P P   C T
                            U             L C   T R U L             L C   T R L L U L C   T R U L C   T R U       T

Robot 1 upper gripper:
<empty>
Robot 1 lower gripper:
<empty>
Mill:
Part                    1 T T T F   0 F F F F   0 F F F F F   0 F F F F   0 F F F F F F   0 F F F F   0 F F F F   0 F

Mill gage.
<empty>
Mill disposer:
<empty>
Lathe:
<empty>
Lathe gage:
<empty>
Lathe disposer:
<empty>
Lathe acceptor:
<empty>
```

Figure 14: Sample Output