Proceedings of

The Fifth Annual CONTROL ENGINEERING CONFERENCE

Held as part of the Control Engineering
Conference & Exposition
O'Hare Exposition Center
Rosemont, IL
May 6-8, 1986

Conference and Exhibition organized by:

Tower Conference Management Co.
331 W. Wesley St.
Wheaton, IL 60187
(312) 668-8100

CONTROL ENGINEERING®

Ada[1] in a Manufacturing Environment

by

Richard Volz
Trevor Mudge
Arch Naylor

The Robotics Research Laboratory
The College of Engineering
The University of Michigan
Ann Arbor, MI 48109
and
Benjamin Brosgol
ALSYS Inc.
1432 Main Street
Waltham, MA 02154

Abstract

This paper identifies software as the key to the computer integrated factory. It then discusses the problem by illustrating the principal components that one typically has to integrate in a manufacturing system. Finally, a solution is proposed that uses Ada software components as standard building blocks. A solution is proposed, illustrated with some simple examples, that use Ada software components as standard building blocks. Finally, the paper summarizes a set of implementation techniques in Ada compilers that help support the use of Ada in manufacturing applications.

1. Introduction

The computer integrated factory has been a goal for the past five years and has been looked upon as the salvation for American industry. Concrete results to date, however, have fallen far short of expectations. One of the principal reasons is the complexity of manufacturing software. Software is the key to successful factory integration. The difficulties in solving the problem of building complex manufacturing software systems are numerous: the diversity of devices and languages used, the distributed nature of the system, the size of systems which must be built, and the persistent intractability of large software systems in general.

Manufacturing software problems are rooted in a long history of building small (by comparison to what is needed) stand alone systems with little regard for interfacing the devices to computers, much less concern for compatibility with other diverse devices. The solutions will also take a long time, and are not amenable to short term patches. The McKinsey Company has recently completed a study on the integration of different levels of CAD/CAM capability into manufacturing systems, and has concluded that retrofitted evolution has only very limited success. Only through careful long term planning for evolution to the ultimate goal will that goal be attained. The kinds of planning and effort needed are comparable to those put forth by DoD in its attack on the large scale embedded software problem that resulted in the development of the Ada programming language. While manufacturing can and should take advantage of DoD's efforts, it has sufficient additional software problems that it must begin its own long range plan on software problems.

Solving the manufacturing software problem requires a full spectrum of activity ranging from both corporate and university research through experimental implementations and testing. To obtain real solutions to the problem, it is likely that cultural shifts will be necessary in both the ways in which automated factories are operated and in relations between the manufacturer and equipment vendors. This paper outlines the nature of the software problem and suggests how Ada can help provide a solution.

2. The Manufacturing Software Problem

The integrated manufacturing software problem is strongly related to both the complexity and size of the system being integrated, and to the diversity of computers, languages and operating systems being used to accomplish the integration. As a basis for discussion consider the key characteristics, from a software point of view, of some of the major kinds of manufacturing devices.

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

- NC machines
  - — Programmed in APT like languages.
  - — Complex controller with only a limited interface to the outside world.
  - — Controllers are typically special purpose devices.
- Robots
  - — Programmed in VAL, Karel, or AML like languages.
  - — Limited interfaces to the outside world.
  - — Controllers typically built upon general purpose computers.
- Programmable Controllers (PC's)
  - — Programmed in ladder logic; limited other programming capabilities.
  - — Currently built out of special purpose hardware.
  - — Only limited interfaces to other computers.
- Materials handling equipment
  - — Typically controlled by a general purpose computer.
  - — Variety of languages used, e.g., assembly language, Fortran, Pascal, C.
- Storage retrieval systems
  - — Typically controlled by a general purpose computer.
  - — Variety of languages used, e.g., assembly language, Fortran, Pascal, C.

Next consider the block diagram of Fig. 1 which portrays the kinds of interconnections which are typical of those desired in the factory of the future. The diagram is coded to distinguish interacting physical devices, local electronic communication and factory wide network communication. Symptomatic problems, which are easy to recognize, are described first. Then, the fundamental problems underlying these and whose solutions are the real key to successful future manufacturing software integration, are identified and discussed.

One of the first symptomatic problems is the inability of devices to communicate with one another. Most manufacturing devices have historically operated in a stand alone mode, and it is only relatively recently that it has been recognized that communication interfaces are necessary. Industry is now rapidly settling on RS-232 and MAP as connection standards, and physical interconnections will be less of a problem in the near future.

However, there is very little standardization of the logical (applications level) interfaces between the devices, and, even if physically connected, effective communication among the software programs on different devices is difficult to obtain. Frequently, the functions needed for an application are obtainable only in awkward ways, and there is generally no capability to modify the software in the device controllers to provide the needed functions in an easy-to-use form, even though the device might be physically capable of doing so. More generally, the software to control the collection of factory devices does not exist. No one knows how to program the entire collection of devices.

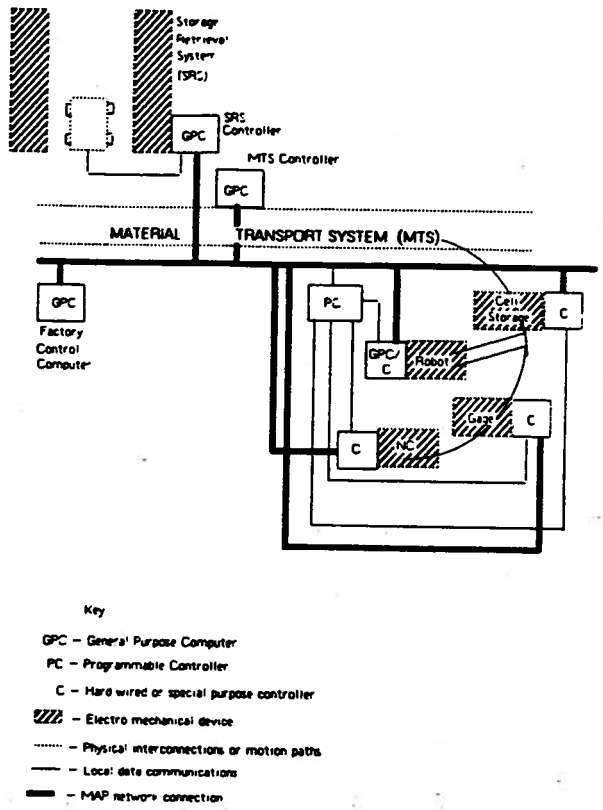

Figure 1

The programming of machines on the shop floor by shop floor personnel when they are part of a larger integrated system, is much more difficult than programming stand alone machines. It is neither clear what constraints exist on the operation of a single machine with respect to others present, nor evident how the programmer can program within these constraints. Debugging of software involving multiple real-time devices is difficult, and latent errors often appear. Problems frequently can be traced to misuse of variables across machine boundaries, and unexpected side effects can occur and create new problems as a consequence of fixing earlier problems.

To make real progress in manufacturing software problems, it is necessary to identify the fundamental underlying problems and focus attention on solving these. First, there is a major lack of understanding and definition of the problem. For example, there is no requirements document for the desired system of the future stating such things as:

- What capabilities should the overall system have?
- What hooks and scars are required for future developments?
- What is the lifecycle view of the systems?

As a result, vendors can only produce what they think the industry needs, without any carefully developed specifications that are consistent with overall integrated manufacturing needs. Often, as marketing ploys, minor differences (called advantages or enhancements) are inserted by competitors into the equipment. Manufacturing equipment, particularly at the software level, are therefore not available as components. This is a major stumbling block to integration. Industry cannot afford to custom design every system to utilize whatever specifications vendors are currently providing.

A plethora of inadequate languages are currently used. These appear to have been developed in ignorance of many important software concepts which have grown out of the past decade and a half of programming language research. These (needed, but often missing) concepts include data abstraction, compile time error checking, large scale programming support through separate compilation and modularization, concurrent processing mechanisms, and real-time support. They are concepts that are all realized in some form in Ada. Also, the languages currently used lack standardization. Most importantly, none of these languages address the fundamental problem that the systems with which we are dealing are distributed. They typically include no development tools that allow one to directly program and debug distributed systems.

Programmable controllers are a major stumbling block to future progress. They appear by the thousands in modern factories and are programmed by skilled tradesmen rather than manufacturing software professionals. They are programmed in a relatively simple and easily understood language, ladder diagrams. While amenable to being programmed by persons with only modest training, the resulting programs are quite limited, particularly in their ability to perform non-binary functions and communicate with higher levels of control programs on other computers. While the controller industry has provided a succession of higher level programming capabilities within PCs, the improvements have only been incremental and are headed in a direction which will introduce other problems as well. We name just a couple of the problems. As higher levels of programming capabilities are added to PC's the skill required of the programmers will have to increase. Current PC's are based on periodic scanning of a large number of input states. As higher level functions, with arbitrary execution times are inserted into ladder outputs, the timing of the scans will be disrupted. Most importantly, as PC's become absorbed into larger systems and are accountable to higher levels of control, it will become impossible to program them as stand alone devices; we do not yet have the mechanisms, hardware or software, by which they can be programmed in this larger context. In spite of all of these problems, the concept of a programmable controller is critical to advanced manufacturing systems, and new approaches to them must be sought. There are, in fact, a number of completely different approaches to the development of programmable controllers which have the potential of offering vastly improved capabilities, both in terms of hardware performance and programming capabilities, and which can yet be, at the low end, compatible with current ladder diagram schemes of programming.

Finally, mechanisms for real-time event driven control of complex systems are not well understood, and general systems for programming them are not available.

3. Toward a Solution

The software problem for manufacturing systems has, then, a number of aspects. We are faced with a heterogeneous mix of programmable devices. Various programming methods are used. Interconnection of devices is awkward. Even if these were not issues, the size and distributed nature of manufacturing software alone would make for very serious problems. Solving problems of this magnitude requires a well thought out plan of attack extending over a substantial length of time, 10 years in our opinion.

As a basis for developing a plan of attack, we first state some desired long term goals. We emphasize that these are conceptual goals only at this point and that much must be done to mold them into a form which is achievable as we develop a detailed plan of attack.

3.1. A Software Components Industry

The first long term goal is the development of a components industry for manufacturing equipment and software. With this industry in place, manufacturers would specify in a formal way the requirements for the manufacturing equipment they need and the component suppliers would supply manufacturing hardware and software components which would "plug" into the rest of the manufacturers system. This is exactly the opposite of current practice in which the manufacturer assumes the responsibility for custom designing the hardware and software interfaces for integration of the system.

For several important reasons to be outlined below, we will view these components from the perspective of a control program on a general purpose computer. From this perspective, the "components" are software abstractions of the real devices [2]. We thus refer to this components industry as a *software components* industry, although in reality many of the software components do, in fact, provide interfaces to hardware devices. Of course, it is possible to have software components which are truly software alone, e.g., arithmetic or database components. The software component view, however, is the correct one for the purpose of building integrated manufacturing systems because this software component is the only aspect of the component to which the high-level control system has any direct access.

A pre-condition for the development of such a software components industry is the adoption of a standard language that supports modern software engineering concepts and has suitable abstraction capabilities. When one considers other language requirements such as support for tasking and timing, large scale program development and extensibility, there is only one practical choice,

Ada. Furthermore, there will be a major software components industry built around Ada which can be drawn upon to support the manufacturing software components industry proposed here.

With Ada as a base language, the software component view can be described more clearly and the interactions between the industrial manufacturer and the (manufacturing equipment) component supplier described in more detail. Figure 2 shows one of the industrial manufactures control computers connected via a communications network to some manufacturing component. Of particular interest is the use of Ada packages as the abstraction mechanism for the component. The package specification is considered to be part of the control program, while the package body is part of the software component.

Several things derive from this view. First, the industrial manufacturer designs the package specification to provide the view of the manufacturing device necessary for the application at hand. Component suppliers are then given the compiled specification and must provide not only the required hardware, but a body to the component package which is compatible with the manufacturer compiled specification as well. Since the component is now carefully specified, several vendors might bid against each other for the job. Second, since the body must reside in the control computer, the supplier must take responsibility for the applications level communication across the network. The supplied software component is directly plugable into the manufacturer's computer.

Third, since suppliers will have a fixed and standard framework within which they must deliver components, it will both be easier to develop custom products and easier to formulate standards when a class of devices has reached maturity.

3.2. Using Ada for Software Components

Ada has been expressly designed to support the software components approach to constructing software for real-time embedded applications (in particular, see Barnes p. 286 [3]). The support for separate compilation, packages, distinct package specifications, and generics are particularly pertinent. Indeed, many of the companies currently focussing their efforts on Ada compilers are likely to redirect those efforts to the production of reusable Ada software components when the acceptance level of Ada has increased sufficiently.

To provide a more concrete picture of an Ada software component consider the case of a controller for a six-degree-of-freedom robot (for further examples see [4]). It may be defined by the following package specification,

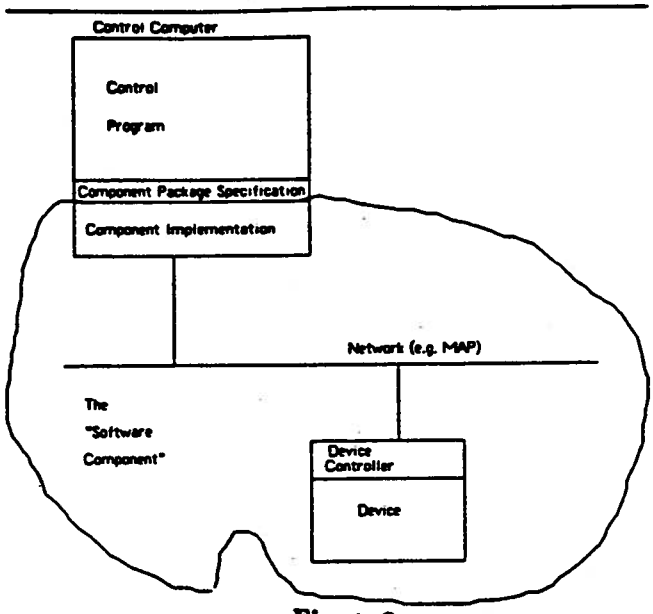

Figure 2

The package specification is bracketed by "**package** ROBOT_6 **is**" and "**end** ROBOT_6;" (Ada reserved words are shown in lowercase bold and variables, types, procedure names, function names, task names and package names are shown in uppercase). In the above case only two procedures are specified—MOVE and HAND. In general, procedures, functions (typed valued procedures), variables, and types can be included in a package specification. The details of the implementations of the two procedures are contained in the body of the package. As far as the user of the robot is concerned it can be manipulated by MOVE and HAND only. These procedures and the number and type of the arguments are the complete specification of the user interface. In terms of Fig. 2 the principal part of these procedures would reside in the control computer. Implementing the actions implied by the procedures would require calls across the network. The important point is that the unnecessary detail about the robot controller has been suppressed. This notion of "information hiding" is the motivation for the package construct in Ada [5]. For instance, the user is not required to know the bit level commands that have to be communicated to the stepper motors, or dc motors, that drive the arm. The user also does not need to know that a coordinate transform is being done in the controller, or the details of the cross-network calls. Of course, this example is a very simple specification. If it were the sum total of the design specification given to a supplier of robots, the supplier would have a lot of freedom—perhaps too much. For example, he would be free to implement his own control law among other things.

```
package ROBOT_6 is
    procedure MOVE ( X, Y, Z: in REAL; DONE: out BOOLEAN);
    procedure HAND ( OPEN, CLOSE: in BOOLEAN; DONE out BOOLEAN);
end ROBOT_6;
```

A good set of requirements would give the minimum ingredients that one would expect to find in a specification like the one above. What do you want to know: the control law—perhaps; bits in the register that determines the position of the third joint—perhaps not. What do you want to hide? In most cases you will probably want to distinguish between a drill press and a robot, although at some level of abstraction, say the flow of the subassemblies through the shop, you may want to regard them both as servers in a network of queues.

In the case where the control computer is responsible for several robots, it may be inefficient to block on calls to MOVE, i.e., to wait until the complete move for a particular robot has taken place before moving another. This need for concurrency can be handled by using a task as follows,

```
package ROBOT_6 is
      procedure COORD (X, Y, Z : REAL ) ;
      function ACK return BOOLEAN;
      procedure HAND (OPEN, CLOSE : in BOOLEAN; DONE : out BOOLEAN);
end ROBOT_6;

package body ROBOT_6 is
      task MOVE is
            entry COORD (X, Y, Z : REAL );
            entry ACK ( DONE : out BOOLEAN );
      end MOVE;

      procedure COORD (X, Y, Z : REAL ) is
      begin
            MOVE.COORD (X, Y, Z) ;
      end MOVE;

      function ACK return BOOLEAN is
            DONE : BOOLEAN;
      begin
            MOVE.ACK(DONE) ;
            return DONE;
      end ACK;

      procedure HAND(OPEN, CLOSE : in BOOLEAN; DONE; out BOOLEAN) is
      begin
            . . .
      end HAND;

      task body MOVE is
            . . .
      end MOVE;

end ROBOT_6;
```

A software components house would have the manufacturer's specification in one hand and the functionality of the robot controller in the other. The component produced could be viewed as a software adapter. Of course the same robot might be used for different applications—some where dynamics are important and other where they are not. These differences can be handled by writing different specifications or by making generic specifications to cover a class of virtual robots. By providing custom package bodies different robots can be made to appear the same. Many possibilities exist. We have used the example of robots, but the above applies equally to other types of manufacturing machinery.

The idea of software specification is an important step in defining the behavior of the subsystems (components) of a manufacturing system. However, there are important extra-linguistic performance parameters. For example, reliability and mean-time-to-failure. These types of things cannot be defined within the language. An interesting question, and one that the benchmark of [6] starts to answer is the issue of specifications that cannot be couched in language terms. In particular, execution times. The language describes function and sequence (and concurrency), but not sequence with respect to a clock, i.e., real-time performance.

3.3. Distributed Programming Capability

It is clear from Fig. 2 that the component vendors must face the problem of writing programs that cross machine boundaries. The same will be true for the manufacturers as well, since even in a modest sized installation there will be multiple control computers which will have to communicate with each other in a manner similar to that shown in Fig. 2. Underlying both groups of software requirements, then, will be a need to write programs that cross machine boundaries. We believe that a promising approach to this requirement is a distributed version of the Ada language, that is one in which a single program can be executed on a set of processors [7]. The point of a distributed language is twofold. First, it would reduce the programmer's view of

interprocessor communication to interprocess communication, which is the programmer's natural view of communication; any special application level communication protocols become transparent to the programmer. It provides a conceptually cleaner view of the program. (Note that MAP makes the communication possible, but not transparent.) Second, a major tool of modern software technology is extensive compile time error checking. The single program view of a distributed system would allow error checking to be done across the entire system instead of, as is now the case, only on the subsets of a program residing on a single processor.

3.4. Smart Manufacturing Development Tools

Once there is a distributed software components industry the software problem for manufacturing will be significantly eased. However, it will not be eliminated. There will still be the problem of software that is peculiar to a given plant and mix of parts. As long as software is tailored to specific situations there will be a continuing software problem. A range of software construction tools are needed. At one end of the spectrum, tools which allow different levels of shop floor and engineering personnel to program parts of the system while retaining overall system integrity are needed. At the other end of the spectrum, and more ideally, what is needed is a way to assemble and adapt assemblies of software components automatically or semi-automatically.

What is important in both cases, however, is that these tools be developed in the overall framework of the manufacturing software structure and not be developed first and then used to force the overall structure into a suboptimal form.

3.5. Fully Flexible Programmable Controllers

There is a strong need for PC's which are much more powerful than those of today. In other words, PC's need to evolve to some form of Industrial Computer. First, controllers must be programmable in a powerful higher level language as well as ladder logic to effect advanced manufacturing controls and communication. To minimize complexity, this language should be the same as used for other parts of the system, i.e., Ada. However, this does not mean that everyone must learn Ada. The user interfaces can be just about anything that seems appropriate. For example, ladder diagrams are not ruled out. The design of these user interfaces should be executed in a way to be compatible with constraints imposed by participation of the controller in a larger control system.

Finally, we note that there are numerous ways in which the industrial computer of the future could be constructed ranging from extended special purpose processors as of today to new multi-processor architectures. A likely profitable direction, however, would seem to be to adapt general purpose computers to the task. They easily make higher level programming languages available and there are techniques for achieving the equivalent of high scan rates. It is also straightforward to provide ladder diagram interfaces to them.

4. Ada Implementation Issues and Strategies

The Ada language has a rich set of facilities for programming the kinds of applications discussed in this paper. Moreover, recent developments in commercial Ada compilers have been extremely encouraging concerning the efficiency of the generated code. Benchmarks, as described in [8], are showing that the run-time efficiency of Ada can match or surpass that of older languages such as C and Pascal.

As an illustration of the current state of the art, this section summarizes the major run time system decisions underlying Alsys' Ada compilers for the Motorola 68000 family. These are validated Ada compilers running on, and generating code for, single-processor systems including Apollo, Sun and Hewlett-Packard workstations with the UNIX operating system. Although these are neither real-time nor distributed systems, an attempt was made in the design to avoid decisions that would interfere with a subsequent adaptation to such environments, and work is currently underway at Alsys on a real-time retarget.

4.1. Principal Design Goals

Efficiency

Efficiency concerns are embodied in the compiler (with two optimization passes), the binder and the run-time system. Particular attention was paid to obtaining fast subroutine linkages, efficient accesses to data, and good performance in task communication.

Support for large programs

Ada is expected to be used for programs comprising tens or hundreds of thousands of lines of code, with extensive data storage requirements. It is important to be careful of implicit restrictions that may be imposed by the machine architecture. For example, on the Motorola 68000 there is a limit of 32K to a stack offset; large objects are therefore not stored in the static part of a stack frame even if their size is static. Alsys' Ada compiler system comprises several hundred thousand lines of Ada and is bootstrapped through itself, requiring that the run-time be able to cope with programs that large.

Retargetability

In the design and implementation attention was paid to separating the machine independent and the machine dependent elements. Decisions such as the implementation of rendezvous were localized to the run-time kernel so that subsequent modifications would not affect the compiler.

4.2. Storage Model

The Ada language has a number of features that require a dynamic scheme of storage allocation at run time and that permit a variety of implementation techniques.

- Subprograms may be nested, with variables from outer levels visible to inner scopes, and subprograms may also be recursive. This combination implies a run-time stack for subprogram data.
- Objects may have dynamically determined sizes. This implies that a stack frame for a subprogram with such objects have a dynamic part as well as a fixed-length part, or that such dynamically sized objects be stored in a separately managed area (the heap).
- Tasks with local data may be created dynamically, and they may be nested. Since a task's statement part can see the names declared in all enclosing scopes, the storage model implied is a "cactus stack". Each task has its own stack; sibling tasks have no access to each other's stack but do have access to their parent's stack.
- The access type facility allows the dynamic association of objects with names. Objects may be explicitly allocated and deallocated by the programmer.

In addition, there are several issues concerning the allocation of global (static) storage—the area occupied by data from library packages:

- Whether there should be a single global area for the whole program, versus individually addressable global areas
- Where the data from package subunits should be allocated

In Alsys' implementation, there are a set of global storage areas, one per library package. Data from package subunits are allocated with the globals of the ancestor library unit. Global areas may be merged into one area under control of the user through an option to the compiler (this option will be supported in a future release).

The main program is regarded as being called from an environment task whose stack length is not bounded a priori. There is a single "heap" area for the whole program; this is used for access type data and for "large" objects (where "large" is a machine dependent value). For programs whose size exceeds the main memory limits, the heap is also used for executable code, since a dynamic loader is available to bring in non-resident units when they are called.

Each child task spawned by the program is provided with a fixed-length stack, allocated from the heap, when it is activated. The stack length is settable as an option to the binder; by default it is 16K bytes.

Global data objects are accessed either directly via an offset from a Global Data register, or, for external globals, through one level of indirection. In the latter case the globals of the unit from which the reference is made contain pointers to the other units' global data areas.

Up-level references are implemented through a "display" mechanism. Each task's stack contains a set of locations, whose size is the maximum static nesting level in the program, that contain the stack addresses of the data currently accessible. The display must be updated in general on subprogram calls and returns, but since Ada does not permit subprograms to be passed as run-time parameters it is possible to implement these updates quite efficiently (more cheaply, for example, than in Pascal). Alsys' compiler implements this more efficient strategy, as well as static analysis that for many cases completely avoids display updates.

4.3. Tasking

The principles of the "cactus stack" approach were summarized above. The whole program is managed as one operating system process, with the run-time kernel responsible for task scheduling, context swapping, etc. Although an alternative approach was considered, to map each Ada task to its own operating system process, this had serious disadvantages. First, UNIX does not allow processes to share common memory as would be required by Ada visibility rules. Second, the implementation of rendezvous through interprocess communication in UNIX would be too inefficient. A difficulty with having a single process for the whole program is that a task performing input/output has the potential to block the entire process. Alsys' implementation avoids this problem for devices such as terminals.

Ada semantics is fairly flexible concerning the task scheduling algorithm. Alsys' implementation allows the programmer to choose whether time slicing is to be used. The current implementation supports ten priority levels but a later release will relax this so that the maximum priority range used is calculated by the Binder.

The rendezvous implementation is "order of arrival": whichever of the two tasks is the later to arrive is the one that executes the accept statement. This has the potential for saving context swaps. The stack used for executing the accept statement is that of the entry caller. This leaves open the possibility of various optimizations of server tasks. Additional optimizations are made when the accept body is empty (e.g., when tasks simply synchronize), again avoiding unnecessary context swaps.

4.4. Memory Management

As mentioned earlier, there is a single heap used for the program. Heap storage is allocated to a program for various reasons. In some cases there is an explicit request (e.g., a call of the allocator function). In other cases the request is implicit: e.g., a new task's stack space, or an object too large to go into some other data area. The Alsys implementation arranges that space which is allocated from the heap will also be returned to the heap when it is safe to do so. Thus, when the scope containing an access type is exited, storage for any objects denoted by values from that type are reclaimed. Analogously, when a task is terminated, its stack space is reclaimed and returned to the heap.

4.5. Exception Handling

Ada's exception handling facility allows the programmer to deal with unusual events or errors through out of

line "handlers" that receive control and complete the execution of the block or unit in which the exception was raised. The philosophy of Alsys' implementation is that exceptions should only add run-time overhead when they are used. Thus the compiler and binder build a set of tables describing where to find handlers at run time. This avoids penalizing normal subprogram calls and returns at the cost of overhead on exception raising and handling.

Exception handling also has an effect on the run-time because of its interactions with other Ada features. Propagation of exceptions out of accept statements influences the rendezvous implementation. Propagation of exceptions out of subprograms means that actions to be performed on exit—e.g., returning storage to the heap—must be done both for normal and abnormal return.

5. Conclusion

Manufacturing systems can be viewed as, among other things, large real-time embedded software systems. Ada has been designed with exactly these kind of systems in mind. It is our belief that if Ada is adopted as a standard by a sufficient number of suppliers of manufacturing equipment, and if Ada is also adopted as a standard of specification by major manufacturers, then the spiraling cost of creating and maintaining manufacturing software can be contained. Furthermore, the concept of software components can lead to truly flexible automation by permitting the function of manufacturing cells to be quickly and reliably modified through interchangeable software components. The increasing sophistication of Ada compilers, particularly in the efficiency of the generated code, is making this concept a reality.

Although the use of Ada as a standard for manufacturing software would represent a quantum jump in solving the wide variety of problems facing the implementor of a manufacturing system, there are still some important language level issues that need further examination. The principal ones relate to the form and meaning of distributed Ada programs.

References

[1] *Ada Programming Language (ANSI/MIL-STD-1815A)*, Washington, DC 20301: Ada Joint Program Office, Department of Defense, OUSD(R&E), January 1983.

[2] R.A. Volz and T.N. Mudge, "Robots are (nothing more than) Abstract Data Types," *Proceedings of the Society of Manufacturing Engineers Conference on Robotics Research*, August 1984.

[3] J.G.P. Barnes, *Programming in Ada*, 2nd ed., Addison-Wesley, London, 1984.

[4] G.D. Buzzard and T.N. Mudge, "Object-Based Computing and the Ada Programming Language," *Computer*, pp. 11-19, March 1985.

[5] M. Shaw, "The Impact of Abstraction Concerns on Modular Programming Languages," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1119-1130, September 1980.

[6] R.M. Clapp, L.D. Duchesneau, R.A. Volz, T.N. Mudge and T. Schultze, "Toward Real-Time Performance Benchmarks for Ada," Robot Systems Division Report, RSD-6-86, Center for Research on Integrated Manufacturing, College of Engineering, University of Michigan, Ann Arbor, MI, January 1986.

[7] R.A. Volz, T.N. Mudge, A.W. Naylor, and J.H. Mayer, "Some Problems in Distributing Real-Time Ada Programs Across Machines," *Ada in use, Proceedings of the 1985 International Ada Conference*, Eds. J.G.P. Barnes and G.A. Fischer, May 1985, pp. 72-84.

[8] M. Gart, "Targeting Ada to 6800/Unix," *1986 Winter USENIX Technical Conference*, January 15-17, 1986, Denver, CO.