Wanted: A New Generation of Software Manufacturing

by

Richard A. Volz
Trevor N. Mudge
Arch W. Naylor

The Robotics Research Laboratory
The College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109

1 Introduction

The flexible computer integrated factory has been a goal of American industry for at least the past five years and has often been looked upon as industry's salvation. 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[1] programming language [1]. 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 rang-

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

Printed in *Standards and Information Technology and Industrial Control*, Eds. N.E. Malagardis and T.J. Williams, North-Holland, Amsterdam, 1988, pp. 153–168.

ing 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 chapter outlines the nature of the software problem and suggests a direction toward a solution.

2 The Manufacturing Software Problem

The integrated manufacturing software problem is strongly related to 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. The machines, robots, material transports, and so forth exist, but the software needed to tie them together into orchestrated, flexible, robust systems does not. The software for the relatively few integrated systems that do exist is very application specific, difficult to maintain, difficult to change, difficult to port to other hardware, not robust and expensive. The following list of typical device characteristics illustrates some of the problems:

- NC machines—Programmed in APT-like languages [2]. Complex special purpose controllers with only a limited interface to the outside world.
- Robots—Programmed in VAL [3], Karel, or AML-like languages [4] [5] or, even more likely, by teaching. Controllers built upon general purpose computers, but interfaces to the outside world often limited.
- Programmable Controllers—Programmed in ladder logic; limited other programming capabilities. Usually built out of special purpose hardware. Only limited interfaces to other computers.
- Materials handling and storage/retrieval systems—Typically controlled by a general purpose computer using languages such as assembly, FORTRAN, Pascal, or C.

Thus, controllers are not designed for interconnection; there are too many languages, many of them inadequate and out of date while others are special purpose; controller hardware often lacks the functionality of general purpose computers; there is almost no use of modern software techniques; and there is no overall theoretical base for integrating the operation of such systems.

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.

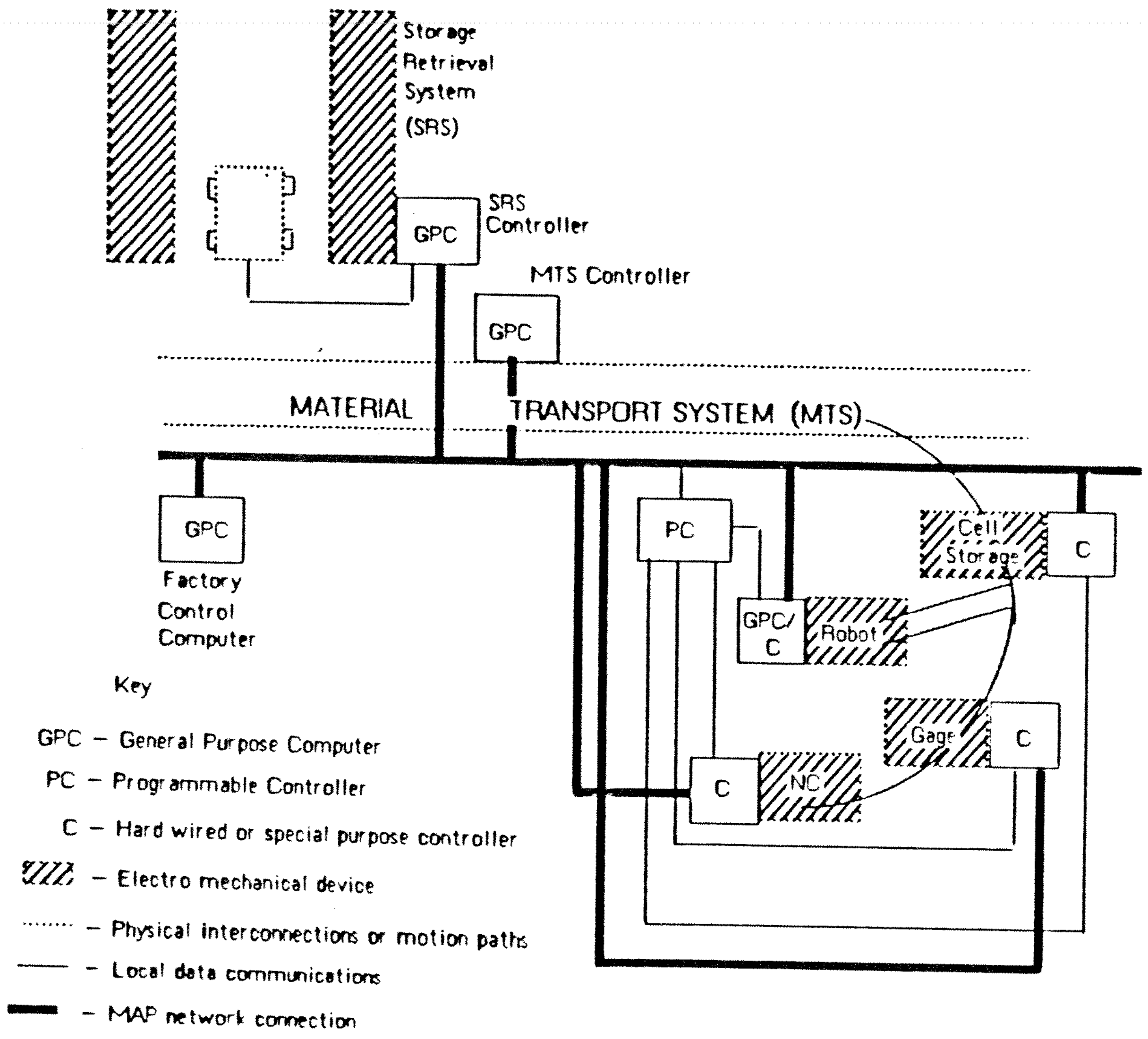
Storage Retrieval System (SRS)
GPC
SRS Controller
MTS Controller
GPC
MATERIAL
TRANSPORT SYSTEM (MTS)
GPC
Factory Control Computer
PC
GPC/ C
Robot
Cell Storage
C
Gage
C
C
NC
Key
GPC – General Purpose Computer
PC – Programmable Controller
C – Hard wired or special purpose controller
▨ – Electro mechanical device
······ – Physical interconnections or motion paths
—— – Local data communications
▬ – MAP network connection

Figure 1:

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, MAP, Ethernet, and token rings as connection standards [6], 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 from a device 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 generally exist.

The programming of machines on the shop floor by shop floor personnel when those machines 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 solving 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 seldom a requirements document for the system being built 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 important concepts include data abstraction, compile time error checking, large scale programming support through separate compilation and modularization, concurrent processing mechanisms, and real-time support. 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.

While they have been a critial factor in the past manufacturing successes, programmable controllers are a major stumbling block to future progress. They appear by the thousands in modern factories and are programmed by skilled tradesmen 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. For example, as higher levels of programming capabilities are added to PC's the skill required of the programmers will 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, at least 10 years in our opinion. The solution must begin with a *new conceptual framework* for integrated manufacturing system control software. The need for a conceptual framework for manufacturing control software was a major conclusion of the recent NSF Workshop on Manufacturing Systems Integration[7].

We propose a conceptual framework based on an intimate blending of modern software concepts and formal models. In particular, we view an integrated manufacturing system as an assemblage of software/hardware components, where "software/hardware component" is a generalization of the software component concept in which, for instance, the internals of a component instead of software might be a robot [8]. The formal models describe the semantics of these components and their assemblages.

Ideally we want these components and assemblages to be generic in a way that allows them to be used in different manufacturing systems, and we would like the creation of assemblages and their models to be computer-aided, semi-automatic, or, perhaps, even automatic. Further, the assemblages are distributed; therefore, issues of distributed processing must be considered and incorporated into the conceptual framework.

Finally, we argue that the foregoing requires a common language environment. However, that does not mean a monolithic software, nor, does it rule out certain parts being written in languages other than the common language. By "common language environment" we mean that the software providing overall structure, and probably most other parts as well, is written in a common language. Software in other languages will frequently be associated with particular devices, for example, a robot programmed in VAL; and these can usually be encapsulated into shells crafted from the common language.

3.1 A Software/Hardware Components Industry

An important 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 [8]. We thus refer to this components industry as a *software/hardware components* industry, emphasing their dual nature. Of course, it is possible to have components which are truly software alone, e.g., arithmetic or database components, just as our components may have a physical aspect to them. The software/hardware component view, however, is the correct one for the purpose of building integrated manufacturing systems because software is the only aspect of the component to which the higher level control system has any direct access.

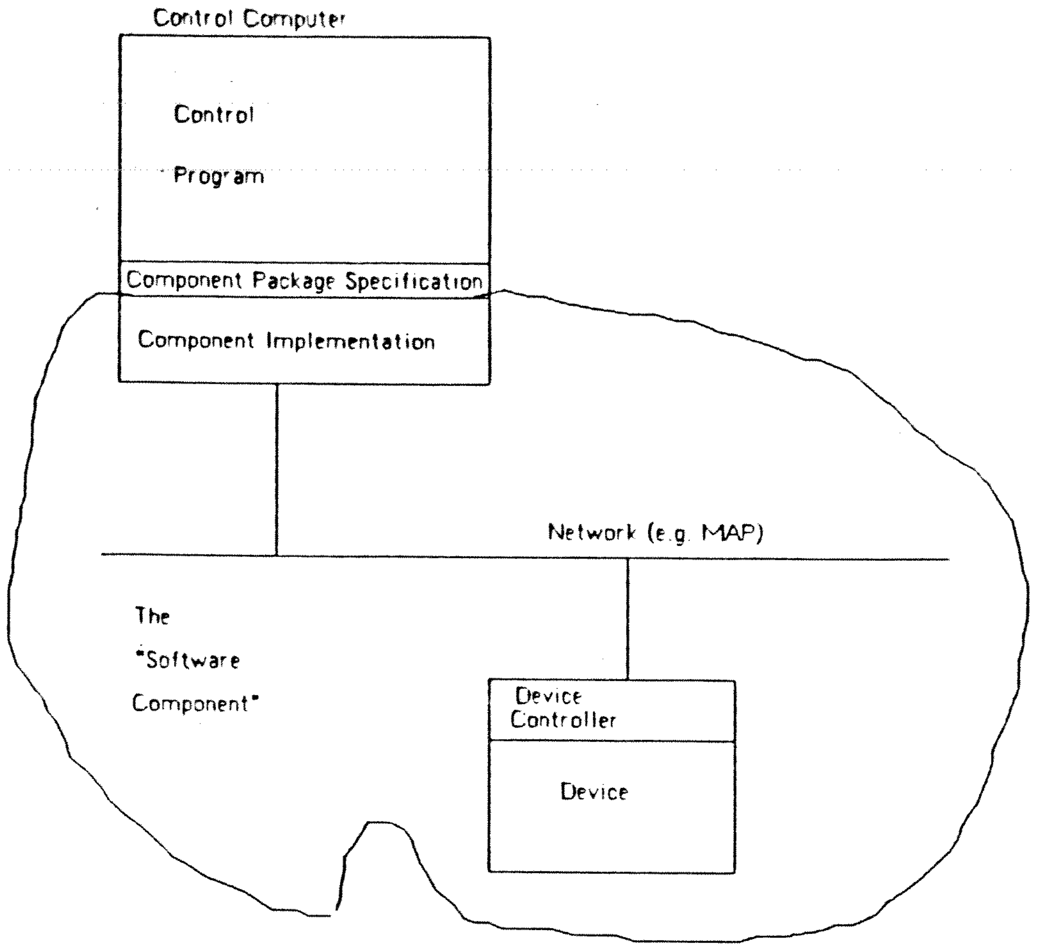

Figure 2: Software Component requiring distributed execution.

A pre-condition for the development of such a software/hardware 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 are few practical choices. The most widespread of these is Ada, and we frame the rest of our discussion in terms of it. Furthermore, there will be a major general software components industry built around Ada which can be drawn upon to support the more general software components industry proposed here.

With Ada as a base language, the software/hardware 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 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/hardware 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/Hardware 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 [9]). 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/hardware component consider the case of a controller for a six-degree-of-freedom robot (for further examples see [10][11]). It may be defined by the following package specification,

```
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;
```

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 variables, types and tasks 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 [12]. 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.

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 specifying MOVE as a task as follows,

```
package ROBOT_6 is
        procedure COORD ( X, Y, Z: in 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: in 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/hardware components house would have the manufacturer's specification in one hand and the functionality of the robot controller in the other hand. 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 others 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[13] starts to answer is the issue of specifications that cannot be couched in language terms. In particular, the time of events. The language describes function and sequence (and concurrency), but not sequence with respect to a clock, ie, real-time performance.

3.3 Formal Models

We need models of the factory floor and process plans to develop control algorithms. Since we realize the factory floor and process plans as assemblages of software/hardware components, we are, in effect, concerned with formal semantic models for such components. The modeling methodology used is described in more detail in [11] [14]. It is a relatively simple formalism based upon as extension to first order logic [15] which is just expressive enough to capture the level of detail—the logical level—with which we are concerned, yet avoids a modeling methodology that is too complicated to be practical. Further, it is a structure that lends itself well to application of artificial intelligence techniques.

Finally, it can be used to represent the process plans that the cell is to implement as well as the actions of the components. The uniform modeling of process plans and software/hardware components simplifies the software structure and allows one to view the process plan as just another component in the system.

One component may include *models* of other components. The models may then used in a predictive simulation manner to examine the likely outcome of a possible control strategy before it is actually applied. And, the formal models of process plans can be converted to actual components that drive the operation of a system. At present the translation from the formal models to actual software is performed manually, but conceptually (at present, and in the future actually) they could be converted automatically. These instantiated components are then used in the control software being developed, as shown in the bottom half of Figure 1.

3.4 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 [16] [17] [18] [19]. 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 modem software technology is extensive compile time error verification. The single program view of a distributed system would allow error verification 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.

Our approach to this need has been to adopt a standard programming language intended for real-time operation and develop a distributed version of it. Because it is basically a good language is subject to intense standardization efforts, and is ostensibly intended for distributed execution, we selected Ada. To achieve distributed execution, we have built a pre-translator that takes a single Ada program as an input and whose output is a collection of pure Ada programs, one for each targeted processor. This is somewhat akin to the way embedded SEQUEL is handled in the DB2 database management system.

Our distributed Ada system [17] allows us to distribute library packages and library subprograms statically among a set of homogeneous processors. We write a single program and use a **pragma** (essentially a complier directive) called SITE to specify the location on which each library unit is to execute. For example, if a simple transport system were controlled by computer number 2 and the cell control using it were on computer 1, a sample of relevant code might look as follows:

```
pragma SITE (2);
package VEHICLE is
        procedure MOVE_FORWARD;
              ⋮
end VEHICLE;
        ⋮
pragma SITE(1);
with VEHICLE;
procedure CONTROL is
  ⋮
begin
  ⋮
  VEHICLE.MOVE_FORWARD;
  ⋮
end;
```

Our translation system would replace the local call to the procedure VEHICLE.MOVE_FORWARD with the appropriate remote call. Similarly any references in CONTROL to data objects defined in package VEHICLE would be translated into appropriate remote references as would task entry calls. Note that the user need only use the normal procedure call mechanism to cause the vehicle to move.

3.5 Smart Manufacturing Development Tools

Once there is a distributed software/hardware 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. As an example of the needed approach we have developed a prototype generic factory floor controller using simulation instead of real components is on a pair of VAX's. The control component is on one VAX and a simulation of the factory floor on the other [11].

3.6 Fully Flexible Programmable Controllers

There is a strong need for controllers which are much more powerful than those of today. In other words, an Industrial Computer is called for. 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 is a major research area.

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 Conclusion

A coherent approach to manufacturing software is one of the most important building blocks needed for U.S. industry to truly develop integrated manufacturing systems. We have described a concept by which coherent manufacturing can be accomplished. However, the theory is not yet complete. Indeed, much remains to be done. Extensions to the formal modeling system are needed to more fully handle generics and distribution of components. The process of instantiation of generics to real components must be extended to allow dynamic instantiations. Distributed languages must be studied in a more general context of multiple forms of memory interconnections, multiple possible binding times, and various degrees of homogeneity (e.g., see the major dimensions of a distributed language defined in [20]).

Yet, we have accomplished enough to demonstrate the viability of the major underlying ideas. A primitive version of a distributed Ada translation system *is* working, and a limited generic real-time factory controller *is* operational, with real factory components replaced by simulation. We believe that when it is fully developed, the approach presented here can become the heart of future integrated manufacturing systems.

References

[1] Ada Joint Program Office, Department of Defense, OUSD(R&D). *Ada Program-*

ming Language (ANSI/MIL-STD-1815A), Washington, D.C., January 1983.

[2] Y. Koren. *Computer Control of Manufacturing Systems.* McGraw-Hill, 1983.

[3] *User's Guide to Val, Version II, 2nd ed.* Unimation Inc., September 1982.

[4] *IBM Robot System/1 AML Reference Manual.* IBM Corp., Boca Raton, Fla., 33432, 1981.

[5] S. Bonner and K. G. Shin. A comparative study of robot languages. *IEEE Computer*, :82–96, December 1982.

[6] M. A. Kaminski. Protocols for communicating in the factory. *IEEE Spectrum*, 23(4):56–62, April 1986.

[7] R.A. Volz and A.W. Naylor. *Final Report of the NSF Workshop on Manufacturing Systems Integration.* Technical Report RSD-TR-17-86, held November 1985 in St. Clair, Michigan and organized by the Robotic Systems Division, Center for Research on Integrated Manufacturing, College of Engineering, The University of Michigan, Ann Arbor, MI 48109, 1985.

[8] R.A. Volz and T.N. Mudge. Robots are (nothing more than) abstract data types. In *Proc. SME Conf. on Robotics Research: The Next 5 Years and Beyond*, pages MS84–493: 1–16, August 14–16 1984.

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

[10] G.D. Buzzard and T.N. Mudge. Object-based computing and the Ada programming language. *Computer*, Mar 1985.

[11] A.W. Naylor and R.A. Volz. Design of integrated manufacturing system control software. *IEEE Trans. on Sys., Man, and Cybernetics*, submitted 1987.

[12] M. Shaw. The impact of abstraction concerns on modular programming languages. *Proc. of the IEEE*, 68(9):1119–1130, September 1980.

[13] R.M. Clapp, L. Duchesneau, R.A. Volz, T.N. Mudge, and T. Schultze. Toward real-time performance benchmarks for Ada. *Communication of the ACM*, (8):760–778, August 1986.

[14] A.W. Naylor and M.C. Maletz. The manufacturing game: a formal approach to manufacturing software. *IEEE Trans. on Sys., Man, and Cybernetics*, SMC–16:321–334, May-June 1986.

[15] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[16] R.A. Volz, T.N. Mudge, G.D. Buzzard, and P. Krishnan. *Translation and Execution of Distibuted Ada Programs: Is It Still Ada?* Technical Report RSD-TR-9-86, Robot Systems Division, Center for Research on Integrated Manufacturing, College of Engineering, University of Michigan, March 1986.

[17] R.A. Volz, P. Krishnan, and R. Theriault. An approach to distributed execution of Ada programs. In *NASA Workshop on Telerobotics*, to appear 1987.

[18] R.A. Volz, T.N. Mudge, A.W. Naylor, and J.H. Mayer. Some problems in distributing real-time Ada programs across machines. In J.G.P. Barnes and G.A. Fischer, editors, *Ada in use, Proc. 1985 International Ada Conf.*, pages 72–84, Cambridge University Press, May 1985.

[19] R.A. Volz and T.N. Mudge. Timing issues in the distributed execution of Ada programs. *IEEE Transactions on Computers, Special issue on Parallel and Distributed Processing*, C–36(4):449–459, April 1987.

[20] R.A. Volz, T.N. Mudge, G.D. Buzzard, and P. Krishnan. Translation and execution of distibuted Ada programs: is it still Ada? *IEEE Transactions on Software, Special Issue on Ada*, to appear 1987.

Notes

As far as a components industry is concerned, one could consider creating generic versions of the above package with control law functions passed in as subprogram parameters.