RSD-15-87

Design of Integrated Manufacturing System Control Software

A.W. Naylor
R.A. Volz

Robotics Research Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

August 1987

Center for Research on Integrated Manufacturing

Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109-2110

TABLE OF CONTENTS

1. Introduction 2

2. Overview of Conceptual Framework 3

3. Software/Hardware Components 5

4. Generics 11

5. Formal Models 16

6. Distributed Language Environments 24

7. Example 29

8. Conclusion 31

References 31

Design of Integrated Manufacturing System Control Software[1]

A. W. Naylor and R. A. Volz

Robotics Research Laboratory
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

This paper describes a coordinated, multifaceted conceptual framework for the development of real-time control software for integrated manufacturing systems. Typically such software is developed using outmoded software tools and is invariably tailored to each manufacturing system. Consequently it is expensive to develop and maintain, inflexible, and relatively unreliable. One reason, though certainly not the only one, there are so few integrated manufacturing systems is that control software has been created in this flawed way. A more orderly, rational and structured approach is obviously needed, and we describe one.

Our approach is based on a blending of modern software concepts and formal semantic models. The software concepts are: (1) the use of software components (extended to include hardware), (2) a common distributed language environment, and (3) generic (i.e., reusable) software. We construct control software as an assemblage of components written in the common distributed language. The formal models model the semantics of the assemblages. Generics are used to enhance the reusability of software.

Our approach was tested using a network of VAX computers as a testbed.

[1]This work sponsored by General Dynamics under Contract Number DEY-605028 and General Motors under Contract Number UM N - 2GM

1 Introduction

Arguably, software *is* the integrated manufacturing problem [1]. 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.

There is a basic need to be able to physically connect things together; that is, there is a need for *physical* and *low-level* interconnection standards, and this has been recognized. MAP [6] and the work at the National Bureau of Standards [7] are outstanding examples of the progress that is being made, and, consequently, interconnection, at least, should cease to be a major hurdle in the near future.

However, as important as low-level interconnection standards are, their existence still leaves major unsolved software problems. Some of these are simply characteristic of large software projects; others are problems of distributed computing; still others arise from the special nature of manufacturing systems themselves. Taken together, they define the next important hurdle for integrated manufacturing systems: How should the large, complex, distributed software systems needed in integrated manufacturing systems be designed, implemented, and maintained? This paper describes an approach for answering some of these questions.

In particular, we develop a *new conceptual framework* for integrated manufacturing system control software. Examples of other conceptual frameworks, for non-manufacturing systems, are the relational database model [8], the ISO network model [9], and the algebraic model for abstract data types [10]. Often they presume, as will ours, a design style. For example, the relational conceptual framework presumes relational databases and effectively excludes hierarchical and network databases. Mainly, conceptual frameworks allow one to describe a situation of interest precisely, to pose problems clearly, and to formulate solution methods. The need for a conceptual framework for manufacturing control software was a major conclusion of the recent NSF Workshop on Manufacturing Systems Integration [11].

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 [12]. 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.

Section 2 provides a brief overview of our conceptual framework. Section 3 describes what we call "software/hardware components". Section 4 presents our view of generics—actually a generalized idea of generics—and the use of generics in manufacturing software. Section 5 presents the modeling formalism that we use for modeling semantics. Section 6 discusses the role of distributed languages. Section 7 discusses briefly an experimental implementation of these ideas.

2 Overview of Conceptual Framework

Our conceptual framework for manufacturing software is based upon the following key concepts which are illustrated in Figure 1:

- recursively defined *hardware/software components* encompassing both the logical and physical aspects of the entity being defined,
- *formal semantic models* of the components and their assemblages, and of process plans.
- *generic* components that can be instantiated into real components by supplying appropriate parameters (either statically at the time a system is built, or during an initialization period with parameters supplied by physical devices attached to the system),
- *libraries* of components, and
- a common *distributed language* environment.

Our view of hardware/software components is both recursive and distributed. That is, components may be built up out of other components, and need not all reside on the same computer. In fact, one component may itself span several computers. Each physical device on the factory floor is encapsulated inside a software/hardware component, and can be thought of as having two faces. One is made up of its interactions on the factory floor. The other is the software interface connected to the higher levels of the manufacturing software system. Our components must encompass both; indeed, the purpose of the software is to control the interactions on the factory floor through the software interface.

The formal modeling part of our framework is expressed in a simple extension to first-order logic [13] that 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.

The generic components are abstractions of real components designed to allow reusability of component software. By *instantiating* them with actual parameters, real components are created and can be used as part of a real factory control system. The top half of Figure 1 illustrates the relationship among the models, generic components and actual parameters. The instantiation process shown here is a generalization of the simple generic instantiation process of languages like Ada[2]. Actual components of the manufacturing system are created from the generic components by supplying parameters that complete the component description.

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.

[2]Ada is a registered trademark of the Ada Joint Program Office.

These instantiated components are then used in the control software being developed, as shown in the bottom half of Figure 1.

The bottom half of Figure 1 illustrates not only a manufacturing software system, but the environments in which it was created and operates. While the underlying digital communications network is shown in this figure, it is logically contained within the distributed language environment. That is, the software for controlling the manufacturing system is written using normal *interprocess* communication mechanisms within the language, without any explicit references to *interprocessor* communication. There is a mapping that assigns program parts to the different processors in the system. The language translation system ensures that interprocess communication is appropriately translated to interprocessor communication when necessary. Thus, the programmer does not have to be concerned with network protocols or inventing application level inter-program protocols.

3 Software/Hardware Components

The main ideas of this section are that software/hardware components and their assemblages offer a powerful way of dealing with manufacturing control software, that their use is best done in a common language environment, and that a common language environment is an ideal environment within which to carry out system integration.

3.1 Software Component Concept

Writing modular programs is an old idea, and the term "software component" is—almost—another name for a module or subroutine. However, there are very important distinctions. Modular programming is largely a programming style, while, as we mean it, "software component" is an enforced language concept. Specifically, our software components have three basic characteristics: (1) a well-defined public interface, (2) an internal implementation which is inaccessible to the user, (see Figure 2), and (3) both the visible part and the inaccessible implementation of software components should be separately compilable from the program components which use them, and it should be possible to separately compile the inaccessible and visible parts. The combination of a public interface and hidden internals (1) provide a clear understanding of how the software component is to be used and (2) precludes access to the internals of the implementation. The latter is crucial for preventing large classes of programming errors, allowing extensive compiler error checking, decreasing debugging time, and increasing program reliability. Separate compilation is important because it allows the development of "plug compatible" software modules and a software components industry, as we explain later.

While the most commonly used languages for manufacturing software, e.g., Fortran, Cobol, Pascal, Basic, do not provide the needed support for software components, what

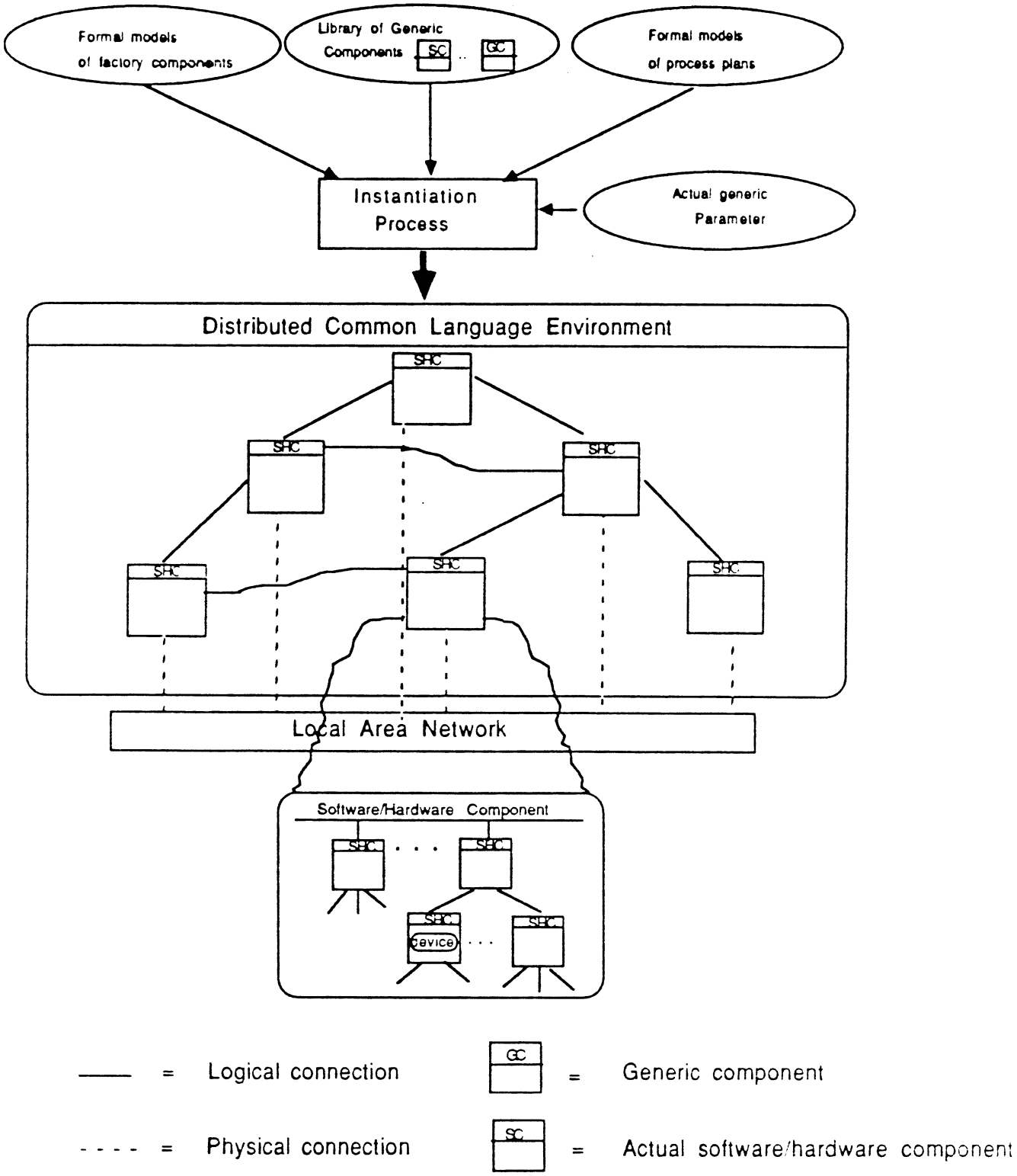
Formal models
of factory components
Library of Generic
Components
SC
GC
Formal models
of process plans
Instantiation
Process
Actual generic
Parameter
Distributed Common Language Environment
SHC
Local Area Network
Software/Hardware Component
device
= Logical connection
= Physical connection
= Generic component
= Actual software/hardware component

Figure 1:

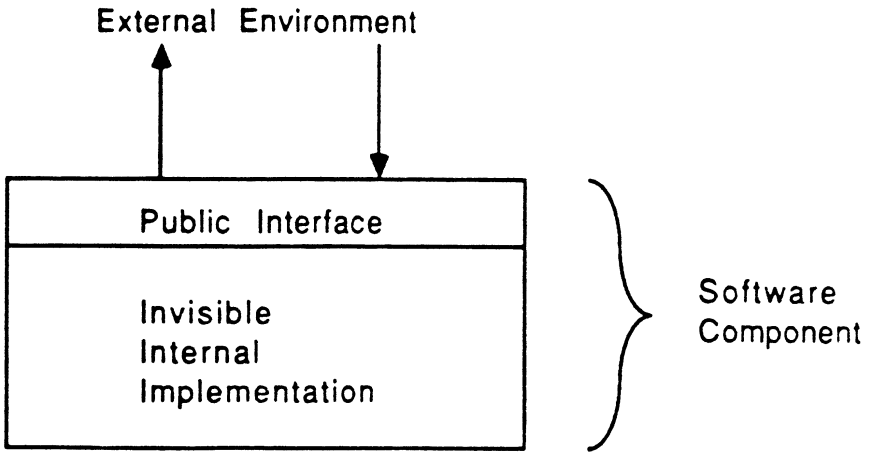

Figure 2:

is needed in a language that would support them is well understood in the software engineering community [14]. And, there do exist languages that meet many of these needs: Modula-2 [15], Ada [16], C++ [17], NIL [18] and CSP [19] are but a few examples, with Modula-2 and Ada probably going furthest.

To illustrate the idea of a software component, we will use the Ada package construct, as Ada is the most widespread of these languages. In Ada, the public interface is called the "specification" and the internal implementation, the "body". A stack is a classic, simple example. The specification might be as follows:

```
package STACK is
        procedure PUSH(X:INTEGER);
        function POP return INTEGER;
end STACK;
```

This and an understanding of the operation (i.e., semantics) of stacks are all the user of this software component needs to know. The body of the stack would be as follows:

```
package body STACK is
        ....
        ....
end STACK;
```

We have purposely not filled in the body to emphasize that it is not available to the user in the sense that it can only be accessed through the specification. Indeed, the user may not even have the source code for the body.

3.2 The Software/Hardware Component Concept

Basic to our conceptual framework is the extension of the concept of software components to include manufacturing subsystems [12]. To illustrate, the specification for a simple material transport vehicle might be as follows:

```
package VEHICLE is
        procedure MOVE_FORWARD;[3]
        procedure MOVE_BACKWARD;
        procedure STOP;
        function MOVING return BOOLEAN;
end VEHICLE;
```

In this case the body of the "software component" includes the actual vehicle, and to signal this fact we refer to it as a "software/hardware component".

In this straightforward mode of operation software components will be available from vendors. The purchaser will be given the compiled specification and body for the target machine and a source code listing for the specification. The user can make calls to the component by referencing only the specification in his or her program. Furthermore, given only the component's compiled specification, the user can compile his or her program. The body is simply linked into the rest of the program at link time. Since a body is compiled after its specification and since the body, having been purchased "off the shelf", does not depend on any other software component in the user's program, this can be easily accomplished. Thus, the specification is a powerful, high-level mechanism of integration that establishes an interconnection between the purchased software component and the rest of the user's program. Different vendors' compiled bodies could be linked in as long as their specifications matched, i.e., plug compatible software can become a reality. Of course, the user can still write his/her own software component.

Usually software components—particularly when thought of as abstract data types—are passive in the sense that they only react to calls. In our case, however, we must leave open the possibility of component being active in that it may originate an event itself, without being called. For example, a contact sensor on a robot may raise an interrupt. There are several explicit ways to handle this asynchronous behaviors. However, we will not go into detail here. The main point of all of them is that a name identifying the event must be exported in one way or another. In particular, it is important that one-way naming be preserved, that is, the software component cannot know the names of it users.

Our extended view of software components also allows an entirely new mode of interaction between manufacturers and suppliers of manufacturing equipment. Manufacturers (purchasers of manufacturing equipment) could design the software component

[3] Although we show only procedures and functions in the examples of specifications in this article, one would also want to allow task entries in practice.

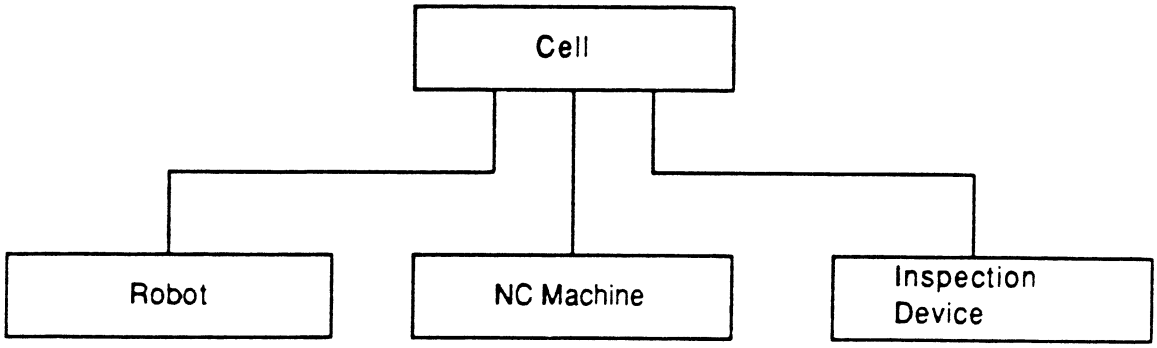

Figure 3:

specifications to provide the view of the manufacturing device necessary for the application at hand. Manufacturing equipment suppliers would then be given the specifications and would have to provide not only the required hardware, but a body to the component package which is compatible with the purchaser specification as well, which would "plug" into the rest of the system. Since the software component would now be carefully specified, several vendors might bid against each other for the job. This is exactly the opposite of current practice in which the purchaser assumes the responsibility for custom designing the software interfaces to vendor supplied devices in order to integrate them into the system. It also places responsibility for development of the body to the software component with the people most familiar with the device which must be controlled.

3.3 Software/Hardware Components as Building Blocks

Robot, NC machine, and inspection device components, might be assembled to create a new software/hardware component called, say, CELL, as shown in Figure 3. As described above, the robot, NC machine, and inspection device each have a public interface (specification) which is available to the internals (body) of the software/hardware component CELL. This component, in turn, will have a public interface that is made available to other parts of the system. For example, several cells might be assembled to form a component called FACTORY_FLOOR as shown in Figure 4. In either case—CELL or FACTORY_FLOOR—we have "multiple inheritance", that is, the component CELL, for example, inherits, modifies, and, perhaps, adds to the functionality of the components, robot, NC machine, and inspection device.

There are two important points. First, the programmer of CELL merely refers to the public interfaces of robot, NC machine, and inspection device, and the language translation system takes care of the actual interconnections. Second, to use the component CELL, or any other one for that matter, we need to understand the interactions of the

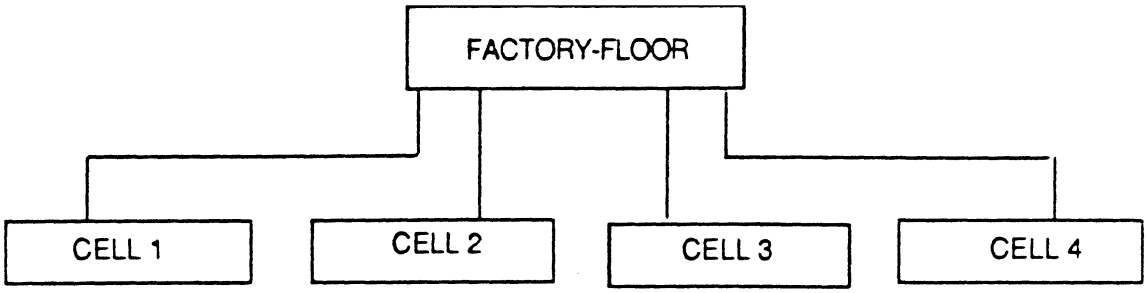

Figure 4:

robot, NC machine, and inspection device as physical devices on the factory floor. That is, we need to know the semantics of each of these components and the way they are set up with respect to one another on the factory floor. This is precisely what the formal semantics models described later give us.

3.4 Integration

We are suggesting a *largely* common distributed language environment, in particular, one in which the system integration is accomplished. That is, except as noted below, the software would be a *single* program written (essentially) in *one* language. This program would, however, be far from monolithic; it would be structured as an assemblage of software/hardware components. This approach offers as least three important advantages.

- It allows the programmers to think about the total software system without being artificially constrained by processor boundaries.
- The compiler can check for errors across the entire system, not just the separate parts.
- The interprocessor communication aspects of integration can be made implicit. Since handling communications explicitly is typically a major programmer overhead task, this is an extremely important saving.

However, in cases where a piece of software written in another language must be used, it is usually possible to encapsulate a software product in a component whose specification is compatible with the rest of the system being developed. The problem is akin to the problem of building a device driver, except that in this case it is (perhaps) purely software that is being interfaced to. The problem reduces to building a single

interface to the program being absorbed. The problem may not be easy, and may require the assistance of the original program builder, but at least there is only a single point of interface to the larger system being built, and once complete the component can be incorporated easily in many different systems. And, note that many pieces of software written in other languages may only need to be "executed" by the control software. For example, from the vantage point of the control software for the integrated system, a parts program is just something to execute. Thus, control software written in the common language might be concerned with the uploading, downloading, starting, terminating and so forth of parts programs written in the other languages.

3.5 Factory Simulation

We are interested in simulations for at least two reasons. First, when developing and debugging control software we would like to replace the actual factory floor with a simulation. Second, certain important control algorithms contain a simulation which is employed to investigate alternative future scenarios. Fortunately, in either case, creating software as an assemblage of software/hardware components lends itself well to simulation. In particular, one can obtain a simulation by replacing the body of each factory floor component—NC machines, robots, and so forth—by software that simulates the semantics of the device. However, one must also simulate interactions of these devices. But doing so using our conceptual framework is easy. In particular, the devices and their interactions can be modeled using the modeling formalism discussed below, and this model can, in effect, be an assemblage of submodels that mirrors the assemblage of software/hardware components.

Further, this approach allows different kinds of simulation clocks to be employed. For example, a clock equal to real time is possible if we want a simulator which, when viewed through its interface, acts exactly like the actual factory floor. Or, if we want a simulator for investigating alternative future scenarios, we can have a clock similar to the one in GPSS and other discrete event simulation languages. Either clock is compatible with our approach to simulation.

The above approach to simulation is an extension of [20] in which formal descriptions of manufacturing cells were extended to include simulations.

4 Generics

Generics are important because they amplify the concept of reusable software and, in our conceptual framework, support semi-automatic model building and programming. Roughly speaking, a generic software component is a template which can be instantiated to get an actual component. A stack is, here also, a classic example.

generic

```
        type ELEMENT is private;
package STACK is
        procedure PUSH (E: in ELEMENT);
        procedure POP (E: out ELEMENT);
end STACK;
```

The point of this generic stack is that it can be used to stack various types of elements. One merely supplies a data type for the generic parameter ELEMENT.
For example,

```
package INTEGER_STACK is new STACK(INTEGER);
```

yields a stack for integers. If stacks for other items are needed, other instantiations of the generic package with different type parameters will create them. Usually, the generic package resides in a library, making it usable in more than one program.

We extend this concept to software/hardware components. Imagine a generic component for material transport vehicles. An instantiation would be a specific vehicle. Information characterizing this vehicle would be passed to the instantiations of the generic component through generic parameters, in the spirit of ELEMENT above, and become part of the real instance of the component, as shown in Figure 5. An instantiation might be as follows:

```
package AGV_7 is new VEHICLE(VOLVO);
```

where VOLVO is an appropriately structured database that characterizes a particular VOLVO automatically guided vehicle. This looks similar to the above instantiation of STACK; however, there are differences. In this case we not only have to supply the information VOLVO, we must supply the actual vehicle. Things might occur as follows: various vehicle vendors would be given the generic interface specification for the generic vehicle component and be told the structure of the generic formal parameters, i.e., the structure that the database VOLVO satisfies. The vendor would supply the generic body (software portion of the internals) of the software/hardware component along with the vehicle.

Although the vendor-user scenario just described is obviously similar to that described for software/hardware components Sec. 3.2, the generic components and declared actual components [4] are different. Contrast the *declaration* of AGV_7_DEC as a component below with the above *instantiation* of the generic component VEHICLE.

[4]By "declared actual components" we mean components that are not the result of instantiating a generic component, that is, these components are *declared* directly. See Figure 1 on page 6.

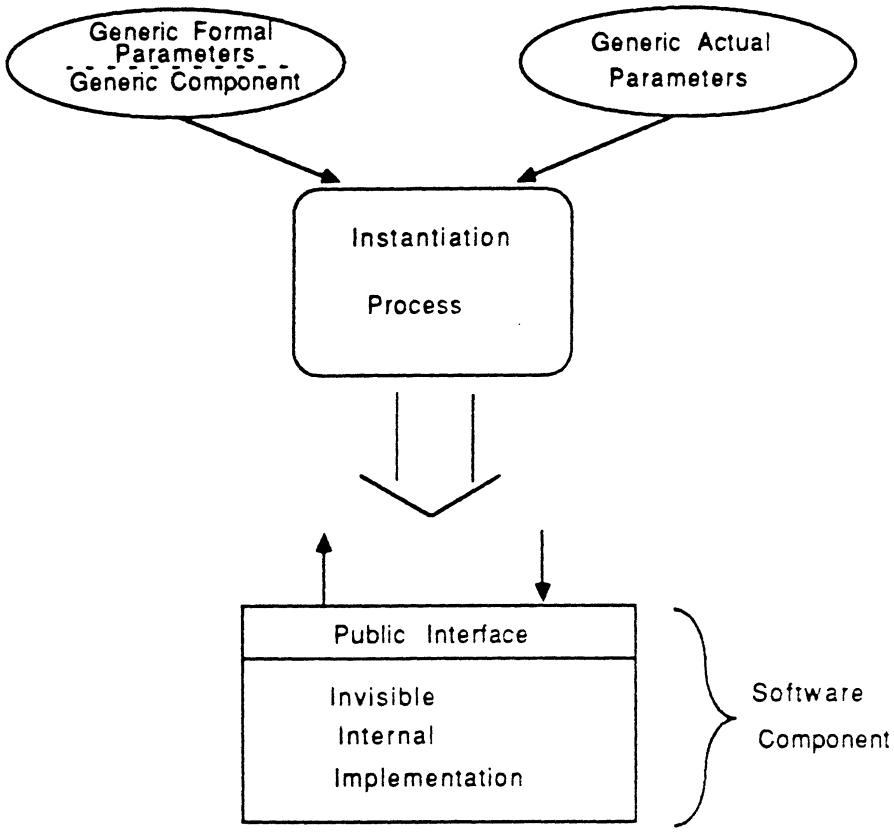
Generic Formal Parameters
Generic Component
Generic Actual Parameters
Instantiation Process
Public Interface
Invisible Internal Implementation
Software Component

Figure 5:

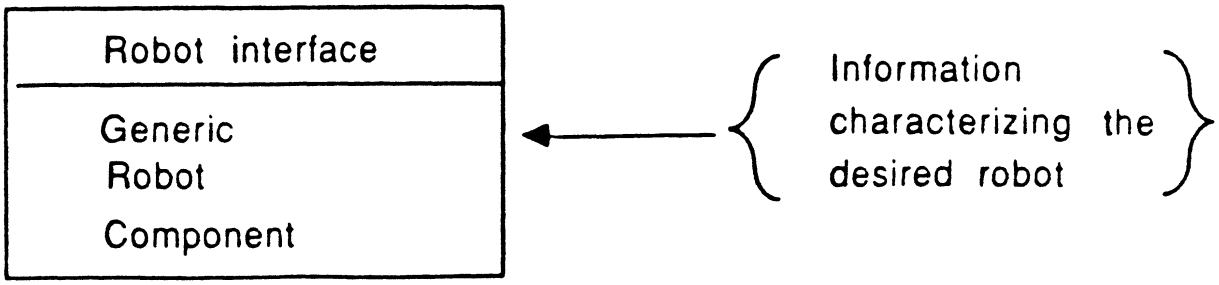

Figure 6:

```
package AGV_7_DEC is
      ...
end AGV_7_DEC;
```

AGV_7_DEC and the instantiation, AGV_7, both refer to specific vehicles. However, we can use the software in the generic component VEHICLE to create actual components for many different vehicles, and these need not be of the same type. For example.

```
package AGV_7 is new VEHICLE(VOLVO);
package AGV_8 is new VEHICLE(VOLVO);
package AGV_9 is new VEHICLE(GM);
```

The package AGV_7_DEC is forever associated with a specific vehicle[5] (or, perhaps, a simulation of it, as discussed in Section 3.5).

Normally, a generic component is instantiated at compile time, but we allow "run-time instantiations" as well. By this, we mean supplying the information at run time. In this case the model of the instantiation looks like that of Figure 6. Whether this should still come under the heading of "generics" and whether we should say "instantiation" is arguable. Perhaps "model reference" might be a preferable term. But, be that as it may, we want to allow both cases: compile-time and run-time instantiation. For example, we might want the generic vehicle component to query the vehicle itself at run time to acquire the information needed for instantiation.

[5]However, it would be possible to associate the *public interface* of AGV_7_DEC with another vehicle by replacing AGV_7_DEC's body with that of another vehicle.

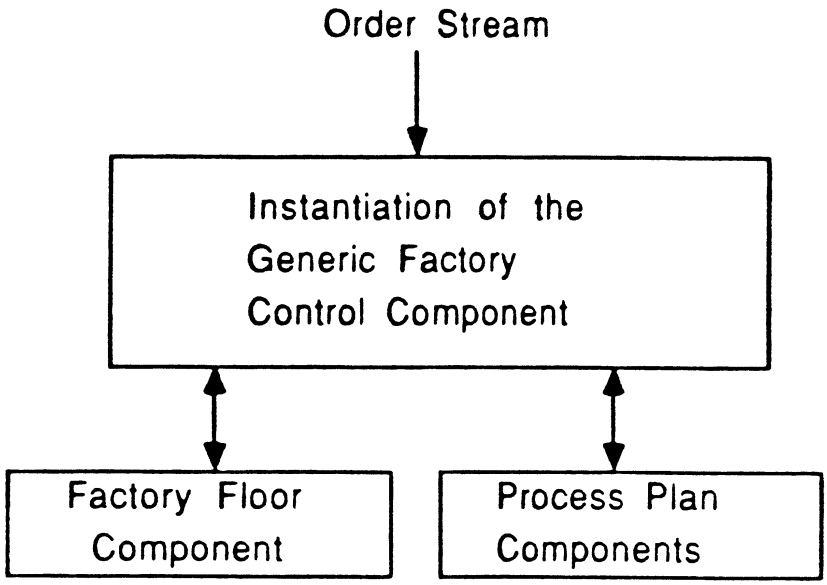

Figure 7:

A far more powerful, and probably more readily achievable, use of generic components occurs in the case of assemblages of components. For example, imagine a generic cell component. It might presuppose that it will use some robots, NC machines, and inspection devices. The information passed to it through generic parameters would specify how many and which kinds of each are in the specific cell. That is, it is given information about the components with which it will interact, and these, in turn, might be the result of earlier instantiations of generic components.

If we continue in this manner assembling ever more elaborate components out of components and instantiations of generic components, we could arrive at a kind of generic component that is of central interest to us: generic factory control components. A given generic factory control component would be capable of controlling any one of a large class of factory floors making any one of a large class of product mixes. The information passed through generic parameters—either at compile time or run time—would include a model of the factory floor to be controlled and descriptions of the process plans for the various parts that are to be made. These models would be formulated using the formal modeling system described below. The factory floor might itself be an instantiation of a generic factory floor component, and, similarly, the process plans might be instantiations of a generic process plan component. Along the way we might have used a generic cell control component. The situation would be as shown in Figure 7. We have assumed that the interface to the control component is simply an input order stream. The control component sends commands to the factory floor component and receives status information, and the process plan component is used to keep track of where each part is in its process plan. Needless to say, we understand that a truly generic factory control component, that is, one for an arbitrary factory making an arbitrary mix of parts, is unlikely to be achievable, but limited versions are.

5 Formal Models

This section describes the way we model the software view of the factory floor and process plans. We need these models 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 [21]. It is relatively simple formalism 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.

5.1 Formal Modeling Concept

Consider a simple a manufacturing cell, i.e. a software/hardware component, whose semantics we want to model, one made up of a robot, an NC machine, a load/unload port, and parts that pass through the system. A simplified interface might be as follows:

```
package MANUFACTURING_CELL is
        procedure LOAD_PART_ONTO_MACHINE;
        procedure UNLOAD_PART_FROM_MACHINE;
        procedure MACHINE_PART;
end MANUFACTURING_CELL;
```

The procedure names and arguments (if present) characterize inputs and outputs. The semantics of the cell correspond to the body of the component MANUFACTURING_CELL, and it is these that we want to model.

We make a distinction between two aspects of manufacturing systems, calling them, respectively, the Euclidean and logical levels. The Euclidean level is characterized by the locations, orientations, and movements of objects expressed in terms of finite dimensional Euclidean spaces. Models are typically coordinate transformations, kinematic constraint equations, differential equations, etc. The logical level is an abstraction of the Euclidean level and is concerned with logical conditions, transformations of logical conditions, and events. Variables simply identify things and, with the exception of time, are not real valued, e.g., a condition might be that "the material transport system is at machine number seven."

Obviously there are important control software problems at both the Euclidean and logical levels. However, the Euclidean control issues are usually local, for example, NC programs and robot programs. The manufacturing system modeling addressed in this paper is primarily concerned with control software for the logical level, since the larger integration issues arise at this level. Still, the software components framework of Sec. 3 is equally valid in both cases.

Our goal, then, is to model the logical view of components such as MANUFACTURING_CELL. Central to our modeling system is the concept of system state. Roughly speaking, we take the state to be a composite of: 1) the set of entities that make up the manufacturing system, 2) the classification of each entity (e.g., robot, part), and 3) "where" the entities are with respect to one another. We say "where" in quotes because at the level of abstraction being addressed by this model it is only necessary to know that the part is on the machine without giving the coordinates of the machine and/or the parts.

More precisely, the state of the model at any given time is characterized by a structure [13] for a first-order language. This language always has a one-place predicate symbol AE and two two-place predicate symbols CT and ST. In addition, it has a number of other, usually one-place, predicate symbols V_α, $V_\beta, \ldots$, and it has constant symbols.

A structure characterizing the state is referred to here as a **configuration**, and it has

- A universe set, called E_u.
- A subset of E_u associated with predicate symbol AE. We denote this set also by AE and rely on context to clarify whether the predicate symbol AE or the set AE is meant. We will treat all predicate symbols and their associated relations in the same manner.
- Subsets CT and ST of $E_u \times E_u$ associated with the predicate symbols CT and ST, respectively.
- Subsets $V_\alpha, V_\beta, \ldots$ of the appropriate Cartesian product of E_u with itself associated with the predicate symbols $V_\alpha, V_\beta, \ldots$.
- Assignments of each constant symbol to an element of E_u.

The elements of E_u are called **entities**, and they are usually things such as parts, assemblies, tools, machines, and messages. That is, they usually have a direct and obvious connection to the real world. E_u is meant to be the set of all entities that will ever be of interest in the model. Those entities that are currently active are in the set AE, for example, the parts currently on the factory floor. The sets $V_\alpha, V_\beta, \ldots$ are **value** sets. For example $x \in V_{Parti}$ or, equivalently, $V_{Parti}(x)$ designates that the entity x is a part of type i. The entities that are active may change with time; however, the values of an entity are fixed forever. That is, a part is always a part. Similarly, the element of E_u assigned to a particular constant symbol never changes.

$CT \subset E_u \times E_u$ is the **contact relation**, and it is used to model such things as the part being on the machine, that is, "the part and the machine in contact." $ST \subset E_u \times E_u$ is the **substructure relation**, and it is used to model such things as the part being a substructure of the assembly.

A configuration, then, is a structure making precise the form of the state of the system. However, we know that some portions of the structure never change—the values, the

assignments of constants, and the universal set E_u—so these can be suppressed in the description of the state, and we can simply keep track of the ordered-triple (AE, CT, ST).

In the simple component MANUFACTURING_CELL given at the beginning of this section the semantics might [6] be as follows: The entity set E_u has entities MACHINE, PORT, MACHINING, and an arbitrary number of parts. The PORT is the place where parts enter and leave the cell. MACHINING would signal the fact that machining was in progress. AE at a given moment would always contain MACHINE and PORT, for we assume they are always active. AE might at the same time contain MACHINING and some number of parts. CT would show where the parts are (i.e., in contact with MACHINE or PORT) and if MACHINE is machining (i.e., MACHINING in contact with MACHINE). Since the example is so simple, there is no need to us the substructure relation ST.

The operation of the system, then, becomes a sequence (AE_k, CT_k, ST_k) of configurations or states. We now describe how state transitions are modeled.

The most basic thing that can happen is a **change**, and, as we will see below, the semantics of the logical view of a software component are modeled with a collection of changes. A change is represented as follows:

$$L(t) \longmapsto R(t_1(t))$$

where L and R are sets of well-formed formulas. The formulas in L refer to the configuration at the start of the change, and those in R refer to the configuration after the change is finished. If all the formulas in L are satisfied, then R, which is usually not satisfied by the current configuration, becomes satisfied by the new configuration. If the change starts at time t, then it finishes at $t_1(t)$, for example, $t_1(t) = t + 12$.

To illustrate the use of changes, let us formulate a semantic model of the logical view of the procedure $MACHINE\_PART$ in the software component $MANUFACTURING\_CELL$. The other procedures would be modeled in a similar manner, and taken together the models of the three procedures would be a model of the software component. Needless to say, there are numerous ways that procedure $MACHINE\_PART$ might work. We will pick one of them. We suppose that if there is a part on the machine and no machining is taking place, then calling the procedure starts machining and a message to that effect is sent. Furthermore, machining is completed 12 minutes later and a message to that effect is sent. If the procedure is called in any other case, it does nothing. The following three changes are for formal semantic model:

$$\{I_{MACHINE\_PART}\}(t) \longmapsto \{\neg I_{MACHINE\_PART}\}(t)$$

[6] We say "might" because it is unrealistic to expect the semantics of a component to be determined by its public interface.

$$\left\{ \begin{array}{l} I_{MACHINE\_PART}, \\ (\exists x)(CT(x, MACHINE) \wedge V_{PART}(x)), \\ \neg CT(MACHINING, MACHINE) \end{array} \right\}(t) \quad \longmapsto$$

$$\left\{ \begin{array}{l} CT(MACHINING, MACHINE,) \\ I_{SEND\_MACHINING\_STARTED\_MESSAGE} \end{array} \right\}(t)$$

$$\left\{ \begin{array}{l} I_{MACHINE\_PART}, \\ (\exists x)(CT(x, MACHINE) \wedge V_{PART}(x)), \\ \neg CT(MACHINING, MACHINE) \end{array} \right\}(t) \quad \longmapsto$$

$$\left\{ \begin{array}{l} \neg CT(MACHINING, MACHINE) \\ I_{SEND\_MACHINING\_FINISHED\_MESSAGE} \end{array} \right\}(t+12)$$

In effect, the logical variable $I_{MACHINE\_PART}$ is an input to the formal model which may be set to true from outside the model. Setting it to true models calling the procedure $MACHINE\_PART$. The first change is really a technicality; it shows that if $I_{MACHINE\_PART}$ is set to true, then it is immediately reset to false from inside the system. The point, of course, is that calling the procedure is a discrete event.

The left side of the second change shows that to start machining there must be a part on the machine, that is, $(\exists x)$ $(CT(x, Machine)$ $\wedge Vpart(x))$, and machining must not already be in progress, that is, $\neg CT(MACHINING, MACHINE)$. In the right side, machining is started, that is, $CT(MACHINING, MACHINE)$, and a message is sent, that is, $I_{SEND\_MACHINING\_STARTED\_MESSAGE}$. Presumably, setting the logical variable $I_{SEND\_MACHINING\_STARTED\_MESSAGE}$ corresponds to calling some other procedure. The times are the same on both sides of the change to indicated that the change is essentially instantaneous. Note that if $I_{MACHINING\_PART}$ is set to true and the rest of the left side is not satisfied, then nothing happens just as we described in words.

The left side of the third change is exactly the same as the second change, so this change occurs if and only if the other one does. The configuration changes to satisfy the right side 12 units of time after machining starts, and it shows that machining has completed, that is, $\neg CT(MACHINING, MACHINE)$. It also sends a message to this effect.

Presumably there will be constraints on the simultaneous execution of the procedures in $MANUFACTURING\_CELL$. We cannot load and unload at the same time. So what happens when several of the procedures are called at essentially the same time? Clearly there is no simple answer. It depends on the way that the real manufacturing cell works, how many processors it has, and so forth; and other software components can be considerably more complicated than our simple cell. But here it is reasonable to assume that no two changes may overlap in time. Further, if, for example, it was

impossible at the time of a call to unload the machine, the procedure might do nothing or it might send a message saying that unloading was currently impossible. The formalism can model these cases as well. It can also model another likely way that the procedures might work. It could be that the body of $MANUFACTURING\_CELL$ effectively queued up calls to the various procedures as is done, for example, at entries to Ada tasks. The formalism can express this also. Thus, it is possible to model highly constrained situations, sequential ones, and highly parallel ones.

The formalism has another important advantage: it blends well with the assembly of software/hardware components. By this we mean that creating a semantic model for an assemblage from the semantic models for the constituent components is relatively easy. Although there are several ways to do this [21], the most useful so far has been based a combination of "unions" and "views" Roughly speaking, the union of two or more models is, as the name suggests, obtained by taking the union of configurations and the union of the sets of changes. For example, if $(AE_1, CT_1, ST_1,)$ and, $(AE_2, CT_2, ST_2,)$ are configurations, their union is $(AE_1 \cup AE_2, CT_1 \cup CT_2, ST_1 \cup ST_2)$; and if C_1 and C_2 are the set of changes for each constituent component the set of changes for the union is $C_1 \cup C_2$. Consider two very simple systems: one has a machine, a loading place, and parts; the other system has the loading place, a vehicle, and parts. The changes in the first allow parts to be moved back and forth between the machine and the loading place. The second allows parts to be moved back and forth between the loading place and the vehicle. In the union parts can be moved from the machine to the vehicle by way of the loading place and vice versa. However, note that the functionality of the union is essentially the union of the functionalities of the constituent components. Since this is often not true of an assemblage, we need to go beyond unions.

A "View" shows part of a system; in particular, it shows part of the configuration and parts of some of the changes. In other words, a view exhibits some of the functionality of the viewed system. For example, in the union just discussed add two new changes: one moves a part from the machine to the vehicle, and the other moves a part from the vehicle to the machine. These additions create a new system. Although neither added change is a change in the union, each can obviously be defined in terms of changes that are in the union. In any event, the union is obviously a view of this new system, since the new system is just the union plus the two added changes. Another, more interesting, view of the new system shows only the machine, vehicle, a part, if one is present, and the two new changes. The loading place is not in the view, nor are the changes in the union. In effect, this view is an assemblage of the two original systems, and its functionality is defined in terms of their functionalities; however, it hides, as most assemblages do, details of the original systems. This example is too simple to show it well, but usually the functionality of such a view will be a restriction on the functionality of the union of the original systems, that is, complete access to the union might allow something that is not possible through the view. This also is the way we expect assemblages of hardware/software components to work.

Another important use of the above formalism is to model process plans. In particular, it is crucial for flexibility that the model of the factory and the models of process plans be separate. One wants to be able to reuse all these models; and if they were intertwined, reuse would be impossible. In a specific application one simply combines (i.e., union) these models to get a model for the application. Thus, it is important to model factories and process in the same way. For example, a process plan for a part in a more elaborate manufacturing system than $MANUFACTURING\_SYSTEM$ might have five steps, each one to be performed on a particular kind of machine. In a formal model each step would be associated with an entity of the type STEP. The changes of this model would enforce the required ordering of these steps. Thus, a condition for starting the second step in the process plan would be that the first step is finished, and so on. Part entities would move through the process plan, moving from step to step. In this way the process plan model could keep track of any number of parts as they progress through the process plan. It would guarantee that the ordering constraints of the process plan were satisfied. However, the process plan model would not show the constraints associated with moving the parts on the factory floor. For example, process plans do not mention material transport systems. We need a formal model of the factory floor, one of the type we have been discussing above. Thus, a factory floor model shows the constraints associated with competition for, say, machines and transport, and the process plan model shows steps and their required ordering. Since we must be concerned with both, these two models must be combined with unions and views. Finally, if more that one kind of part is being manufactured, there will be more than one kind of process plan model in the combination.

5.2 Relation to Other Semantic Models

Our method of modeling software components is reminiscent of formal methods used to model abstract data types and the semantics of programs, but there are differences. The most obvious ones are the explicit role of time and the possibility of genuine parallelism. In other methods, time is rarely part of the model, and parallelism, when considered, is usually treated in terms of concurrency. We can ignore neither.

Another obvious difference is the potentially active nature of the software components. Usually formal models of abstract data types implicitly assume a passive object. Such an assumption is not tenable when one is modeling a manufacturing cell with random delays and spontaneous events, and our formalism does not make such an assumption. Thus, our concept of a software component—the *active* software/hardware component—is a generalization of the usual formal characterization.

Our approach is also similar to operational semantics [22, pages 19–21, 337] in the sense that we seem to define an abstract machine.

However, there is a difference: Here state transitions are defined syntactically in terms of sets of well-formed formulas. This gives rise to a problem that we call the "updating

problem". It arises because configuration or state for some syntactically-defined changes are ambiguous. The problem is essentially the world selection problem in counterfactual reasoning [14], [15]. The problem has been solved in [60] in the sense that simple necessary and sufficient conditions have been developed which identify unambiguous changes.

There is also a similarity to axiomatic semantics [17], [18] in that we have "before" and "after" conditions. However, the before and after conditions of axiomatic semantics are usually not intended to define the program fragment, while this is precisely our intention: we have no other way to say what a change does.

A connection to denotational semantics [22], [23] is also possible, but in our case the emphasis on *how* a configuration is reached is so paramount that denotation of the reached configuration is really secondary. Moreover, our need to allow genuine parallelism seriously limits the usefulness of the process model of program semantics [24], [25].

Our approach is also related to the algebraic approach to modeling and specifying abstract data types [10, pages 189–218], [26], but again there are differences—those already discussed plus another one: it is awkward to treat the software/hardware components that we are interested in as algebras. In particular, before and after conditions are conditions on entities and pairs of entities, whereas **changes** (i.e., operators) alter the relations AE, CT, and ST. This means that there is an implicit second-order logic present, and this is the source of the awkwardness.

For example, the configurations in our formal model can be viewed as relational databases, and this would probably be the way that they would be recorded. **Changes**, then, are in part operations on databases; in fact, they are updating operations. But note that this means that we need two kinds of variables: entity-variables for things such as parts and machines, and the "database"-variables AE, CT, and ST. The latter are predicate-variables, and this, in turn, means that we have a second-order logic [13, page 269].

Yau and Caglayan [27] model software components using modified Petri nets, and many of the differences already discussed apply here. However, they do develop formal methods for interconnecting software components: they "tie" two Petri nets (i.e. components) together.

Finally, it is obvious that our modeling approach is similar to the situation-based models used in planning [28], [29], and [30]. There are differences; however, for the purposes of this article they can be ignored.

5.3 Relation to Manufacturing System Models

There are many modeling techniques for manufacturing systems [31], [32]; however, most are not appropriate for describing, as we want to do, a factory floor and associated

process plans to a generic control component. For example, industrial engineers have used networks of queues [33], [34], [35], [36] and Markov processes [37], [38], [39] extensively to model manufacturing systems.

In queuing networks control is implicit in the structure of the model, a number of details of the logical level are suppressed, and the process plan models are also implicit in the structure of the model. Such models are also inappropriate for our purposes in that their product is equilibrium statistics, and these are of indirect and limited benefit when trying to develop real-time controls. In other words, these models are at too high a level and yield, for our purposes, the wrong perspective on the manufacturing system. Markov models have essentially the same kinds of problems with the one exception that control can be made explicit.

Petri net models of manufacturing systems [40], [41], remove several of these difficulties. They can model at a level of detail appropriate for treating control of the logical level, and generalizations allow time and stochastic behavior to be modeled. However, our approach appears to lead to more compact models and be more natural for the modeler. Particularly important is the fact that the connection between **changes** and software component interfaces of our modeling system yields a natural bridge between software and models. Further, ours is based on a branch of mathematics, mathematical logic, that has decades of developments to call upon, and is in form that allows rather direct use of artificial intelligence and relational database ideas. Still a careful comparison of our formal models and Petri net models needs to be made. For example, it appears that any Petri net model that does not contain Euclidean level information can be modeled more concisely using our formalism. More importantly, it appears that our modeling system is more expressive in that it can model behavior that either cannot be modeled or is awkwardly modeled using even colored Petri nets [42]. On the other hand, there are Petri net concepts such as invariants and linear system theory ideas based on [43] that might be modified to be suited to our models.

Manufacturing systems are also being modeled with rule-based systems [44], and one interpretation is that ours is a rule-based model. However, there is a subtle difference. Simple rule-based models—that is, ones without truth maintenance systems—do not have an updating problem because their rules state precisely what has to be added to the database. In our case a **change** expresses a condition that the updated database must satisfy but does not explicitly specify what the next database should be. That is, in our case specifying a new fact often carries with it the need to change old facts. In other words, we need the truth maintenance system that is a solution to the updating problem discussed above. Since this truth maintenance system is independent of the structure of a particular model, it is not part of the model and remains implicit. It is precisely this implicitness that makes our models concise. Finally, note again that we allow true parallelism, and this is not typical of rule-based systems.

Although automatic or generative process planning has been investigated, little attention has been given to modeling process plans in a manner that allows models of the manufacturing system and models of the process plans to be combined into a common

formal framework while maintaining their separate identities. This is an important feature of our approach. We view the process plans as "programs" for the factory floor "computer", and it is even sometimes useful to think of the generic factory control system as a kind of operating system that sees to the fair and timely execution of these "programs". An interesting advantage is that this "operating system" knows a great deal about the future in that it can look ahead through all the active process plans. Naturally, there is still uncertainty: machines can break down, new orders can arrive or be cancelled, etc. Still this "operating system" has far more knowledge of the future than real operating systems do.

6 Distributed Language Environments

In this section we discuss the programming language aspects of our conceptual framework. In particular, we are concerned with a distributed language environment which supports real-time software/hardware components, their assemblages, and generic components.

6.1 Distributed Computing for the Conceptual Framework

To illustrate the role of distributed program execution in our framework, consider the situation in Figure 8. In this case, the control software resides on one computer, requiring that the specification of the software/hardware component also be available on that computer. The body implementing the component is split between that and another computer. The principal issue is that the logical function being performed extends across more than one processing unit. To us, then, a distributed language environment means more than just writing programs in the same language on each machine in the system. It means *distributed execution* of a *single* program across machine boundaries. In this way the advantages of proven software engineering mechanisms [7] can be achieved in the distributed environment. These capabilities are accepted as being essential for the effective development of large complex programs in a uniprocessing environment [45], [46], [47]. They can certainly be no less important in the distributed environment. Indeed, a distributed language capability is fundamental to our approach to manufacturing software.

6.2 Distributed Language System

A number of distributed languages have been proposed, e.g. [16], [18], [19], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61]. However, most

[7] These mechanisms include such things as: 1) data encapsulation and hiding, 2) abstract data types, 3) modularization of programs, 4) separate compilation (of both modules and specifications), 5) concurrency mechanisms at the language level, and 6) extensive compile time error checking.

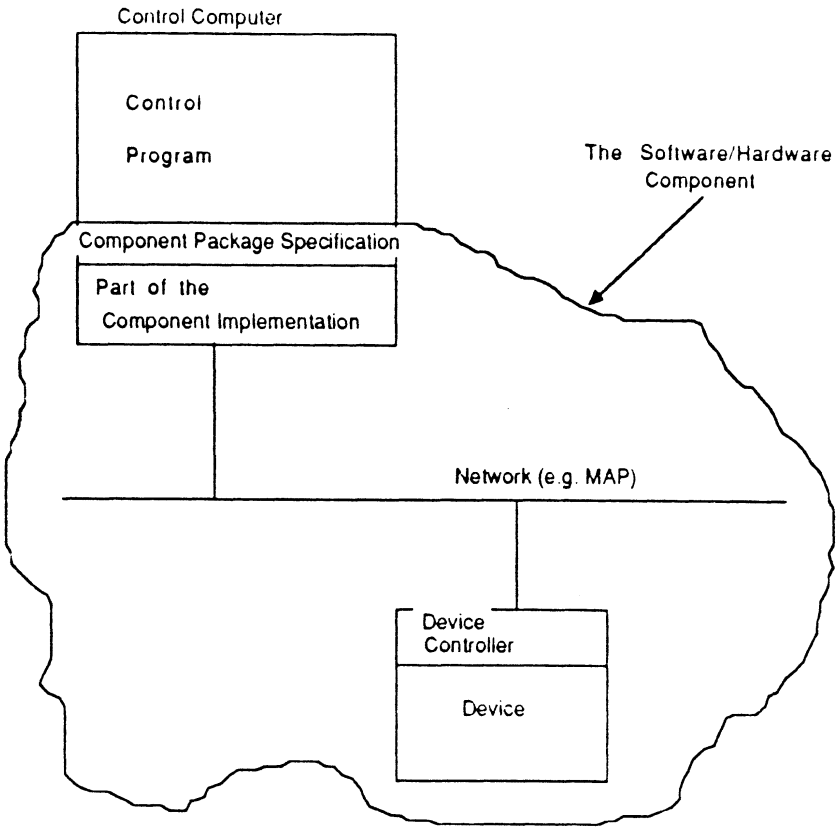

Figure 8: Software Component requiring distributed execution

were intended primarily for studying such things as synchronization and communication primitives. Only a few languages, e.g., [16], [52], [55], seriously consider real-time issues, most do not provide generic capabilities, and there are many aspects of distributed execution they do not consider. There is certainly no widespread available of vendor-implemented, distributed languages, and one will eventually be needed to support our conceptual framework.

Our conceptual framework must operate in real-time as well as on distributed processors. It is one thing to create powerful expressive mechanisms for distributed program representation; it can be quite another to implement them efficiently enough for real-time operation. For example, MAP is a reasonably logical approach to communication, but it is now suspected that the cost of its elegance is slowness, e.g., it takes up to 100 ms. for an end-to-end message transmission, regardless of network speed [11]. We must be careful not to create a comparable problem.

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.

This approach illustrates two key views we hold.

First, we do not believe that a special purpose distributed language should be developed for integrated manufacturing systems. This, we believe, would be a dead end, and guarantee that manufacturing software would remain an expensive cottage industry. Rather, we believe that our conceptual framework—and manufacturing software in general—should, as much as possible, be in the mainstream of software development and utilize the extensibility of modern languages, proposing extensions only where absolutely necessary.

Second, as we have said earlier, we believe that control software should be written as one program in a common distributed language. Our interim approach allows this, and we have experienced some of the advantages discussed earlier, even in this initial version.

Our distributed Ada system 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 the simple transport system mentioned earlier 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.

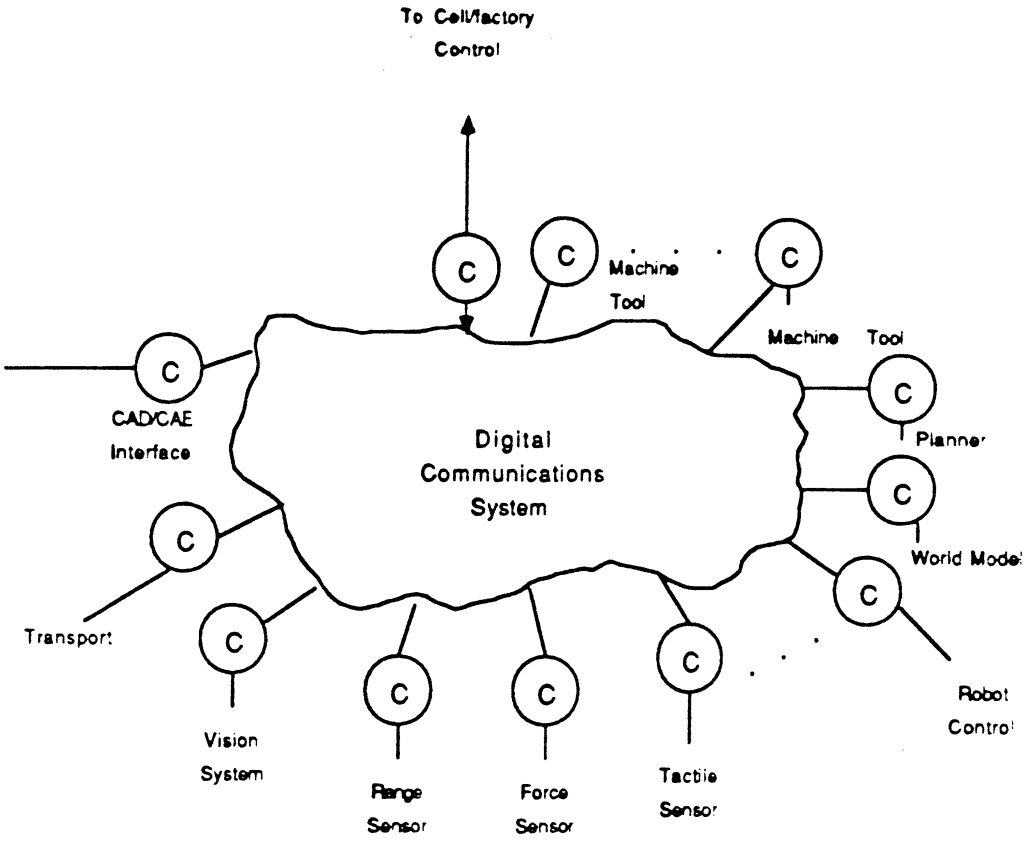

Figure 9:

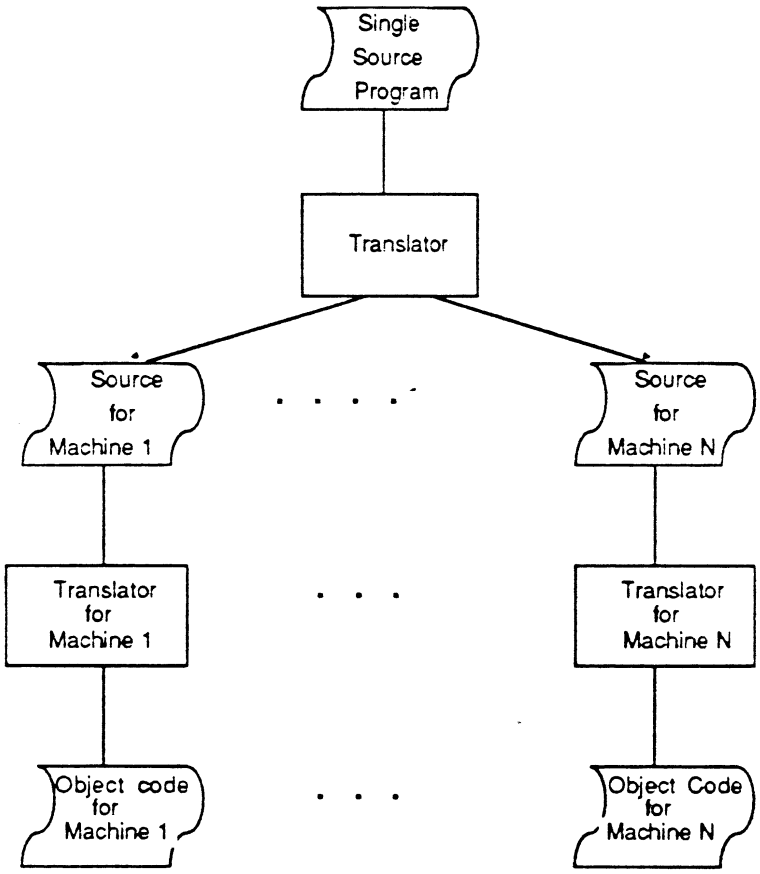

Figure 10:

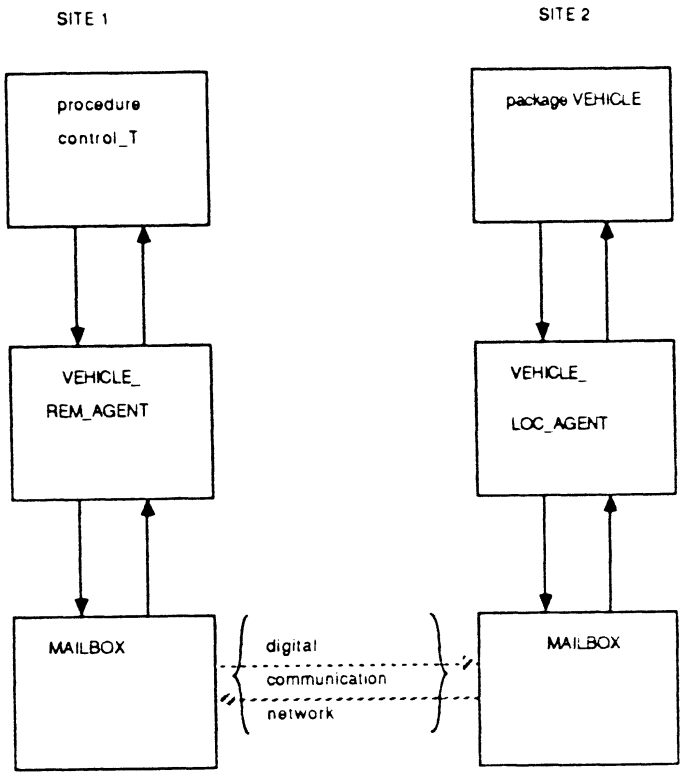

Figure 11:

The operation of our translator is illustrated in Figures 9, 10 and 11. We presume that our computers are interconnected by a communication network, as shown in Figure 9. Figure 10 illustrates the operation of our translator. It translates our single Ada program into a set of independent Ada programs which include library modules of our design to effect communication among processes. Each of the individual programs can thus be compiled by an existing Ada compiler. This approach simplifies the translation process considerably since our pre-translator is much less complex than a full Ada compiler.

The communication is managed by a set of *agents* [62] created for units that can be remotely referenced and an underlying process-to-process mailbox routine. Roughly speaking, the relevant entities and flow of data are as shown in Figure 11. CONTROL_T is the translation of CONTROL with the references to objects in VEHICLE replaced with calls into VEHICLE_REM_AGENT.

The operation of this system can be modeled in terms of the run time overhead associated with various kinds of remote references. From the tests performed in [63] we know that task rendezvous times exceed procedure call times by one and a half to two orders of magnitude. We can also reasonable expect the network communication times to be sizable. For example message end-to-end times for repeat MAP are on the order of 100ms, [29], for the Intel hypercube, a few milliseconds, and for the NCUBE hypercube, several hundred microseconds to a millisecond. Thus, we neglect all local procedure and function call times, and model our overhead in terms of the number of messages and

local rendezvous required. Let t_m and t_r be the times to complete a message transfer and local rendezvous, respectively, and let n_m^o and n_r^o be the number of messages and local rendezvous required for a remote operation of type o. Then, the time to complete a remote operation is

$$n_m^o \cdot T_m + n_r^o \cdot t_r$$

For task rendezvous we have $n_m^r = 2, n_r^r = 6$ and the remote rendezvous time is $2 \cdot t_m + 6 \cdot t_r$. For example, if we have a slow network in which t_m = 100ms and an Ada compiler that produces 0.500 ms rendezvous times (several such compilers now exist for Motorola 680x class processors), we would have a remote rendezvous time of 203 ms, obviously, heavily dominated by the message passing time. For a fast network with, say, a t_m of 500 μ sec, we would have a rendezvous time of 4 ms., which is close to single machine rendezvous times of a year ago [63].

Clearly, within the message passing times achievable with today's technology, a reasonable distributed language capability is feasible. For a more complete description of the distributed Ada translator system, see [62].

7 Example

A preliminary version of a generic factory floor controller using simulation instead of real components is operational on a pair of VAX's. The control component is on one VAX and a simulation of the factory floor on the other, as shown in Fig. 12. The factory floor controller sends commands to the factory floor simulator on VAX 1. Each time something changes in the simulator the controller is notified, and the calls for the same change in the factory floor tracking model in VAX 1. Except for a small time delay, the configuration of the tracking model is supposed to be the same as that of the factory floor simulator. The controller uses the process plan tracking model to keep track of where parts are in their process plans. There is no comparable model on VAX 1 because a model of the factory floor says and knows nothing about process plans. The connection between movement on the factory floor and progress through the process plan must be made on VAX 2. The search models on VAX 2 are used to select the next commands to send to the factory floor simulator. At the beginning of a command-selection cycle the configuration of the search model is made the same as that of the tracking model. Then the controller uses the search model to build a tree of various future scenarios. A few simple pruning rules keep the search manageable. The most desirable path through the tree is selected, and its initial part is used to select the next command(s). This next command(s) is sent to the factory floor, and the cycle begins again.

Carefully note that factory floor models, process plan models, and the controller are separate in this system. In principle, but not completely in practice, one should be able to change any one of them. This is a major source of flexibility.

Also note that communication is in dashed boxes, that is, the agents. This is done

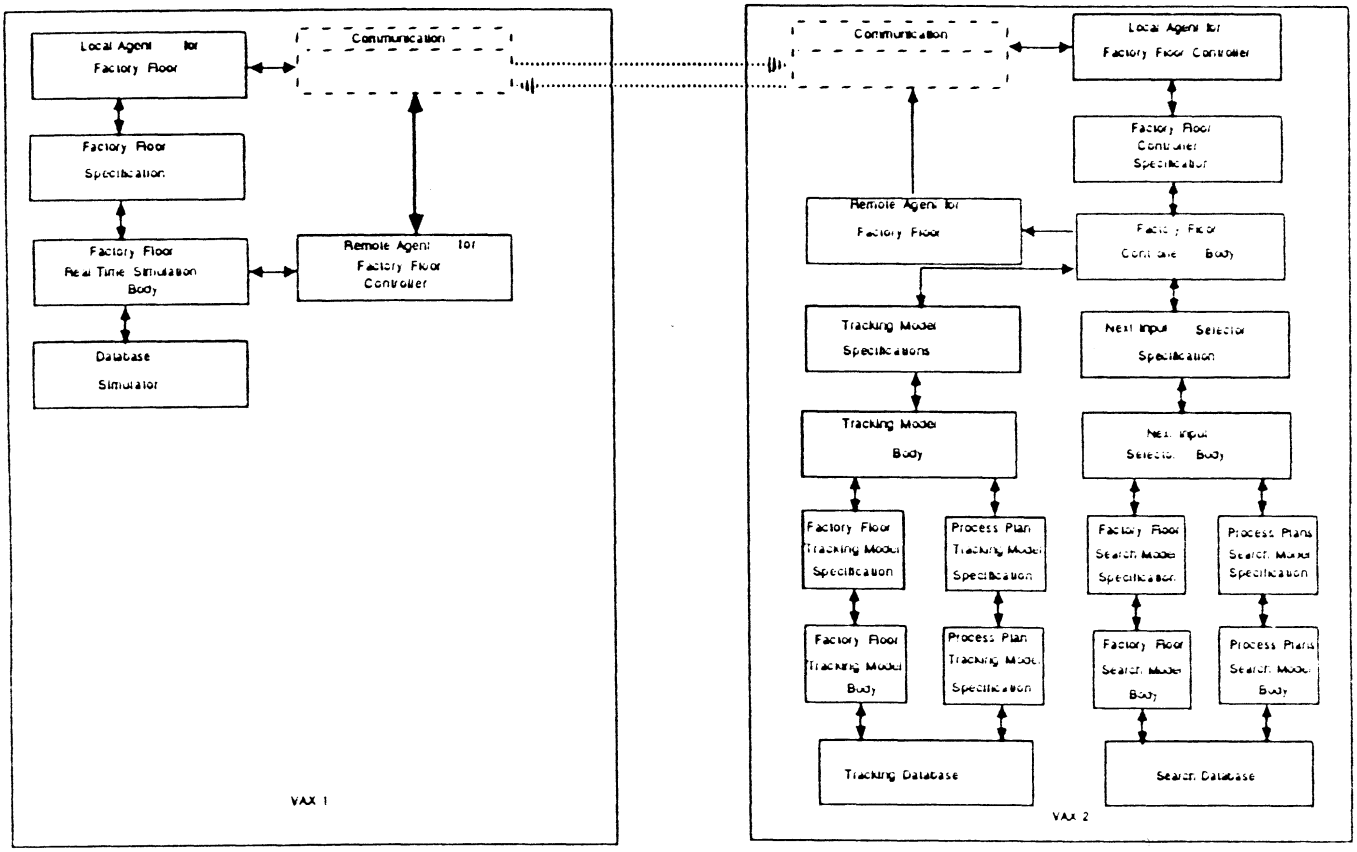
Local Agent for
Factory Floor
Communication
Factory Floor
Specification
Factory Floor
Real Time Simulation
Body
Remote Agent for
Factory Floor
Controller
Database
Simulator
VAX 1
Communication
Local Agent for
Factory Floor Controller
Factory Floor
Controller
Specification
Remote Agent for
Factory Floor
Tracking Model
Specifications
Next Input Selector
Specification
Tracking Model
Body
Factory Floor
Tracking Model
Specification
Process Plan
Tracking Model
Specification
Factory Floor
Search Model
Specification
Process Plans
Search Model
Specification
Factory Floor
Tracking Model
Body
Process Plan
Tracking Model
Specification
Factory Floor
Search Model
Body
Process Plans
Search Model
Body
Tracking Database
Search Database
VAX 2

Figure 12:

to indicate that communication is invisible to the programmer. In particular, the FACTORY_FLOOR component on VAX 1 "thinks" that it is connected directly to the FACTORY_FLOOR_CONTROLLER on VAX 2 and vice versa. Our approach to a distributed language, discussed in Sec. 6, is that the programmer should not have to be concerned with this communication software or the extra specifications. This is the job of the language translation system.

8 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 [64]).

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] R.A. Volz, T.N. Mudge, A.W. Naylor, and B. Brosgol. Ada in a manufacturing environment. In *Proc. Fifth Annual Control Eng. Conf.*, pages 433–440, Rosemont, IL, May 1986.

[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] J.A. Simpson, R.J. Hocken, and J.S Albus. The automated manufacturing research facility of the national bureau of standards. *Journal of Manufacturing Systems*, 1(1):17–32, 1982.

[8] D. Maier. *The theory of Relational Databases.* Computers Science Press, 1983.

[9] A. S. Tanenbaum. *Computers Networks.* Prentice Hall, 1981.

[10] J.C. Cleaveland. *An Introduction to Data Types.* Addison-Wesley, Reading, Mass, 1986.

[11] 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.

[12] 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.

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

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

[15] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[16] Ada Joint Program Office, Department of Defense, OUSD(R&D). *Ada Programming Language (ANSI/MIL-STD-1815A)*, Washington, D.C., January 1983.

[17] B. Stroustrup. An overview of C++. *ACM Sigplan Notices*, 21(10):7–18, October 1986.

[18] R.E. Strom and S. Yemini. Nil: an integrated language and system for distributed programming. In *Sigplan '83 Symposium on Programming Language Issues in Software Systems*, pages 73–82, June 1983.

[19] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[20] C. Antonelli and R.A. Volz. Hierarchical decomposition and simulation of manufacturing cells using ada. *Simulation*, 46(4), April 1986.

[21] 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.

[22] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[23] M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[24] R.E. Milne. *The formal semantics of computer languages and their implementations.* PhD thesis, Univ. of Cambridge, 1974.

[25] A.J.R.G. Milner. An approach to the semantics of parallel programs. In *Proc. Convegno di Information Teorica*, Istituto Di Elaborazione della Informazione, Pisa, 1973.

[26] J.A. Goguen and J.W. Thatcher. Initial algebra semantics. In *Proc. 15th IEEE Symp. on Switch. and Auto. Theory, New Orleans*, October 1974.

[27] S.S. Yau and M.U. Caglayan. Distributed software system design representation using modified petri nets. *IEEE Trans. on Soft. Eng.*, SE-9:733–745, Nov 1983.

[28] E. Rich. *Artificial Intelligence.* McGraw-Hill, NY, 1966.

[29] C. Green. Theorem proving resolution as a basis for question-answering system. *Machine Intelligence*, 1969.

[30] R.E. Fikes and N.J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *In Artificial Intelligence*, 1971.

[31] Oper. Res. Soc. of Amer. and The Inst. of Manag. Sci. *Proc. 1st ORSA/TIMS Spec. Int. Conf. Flex. Manuf. Sys.*, Aug 1984. held at The Univ. of Mich.

[32] R. Suri. An overview of evaluative models for flexible manufacturing system. In *Proc. First ORSA/TIMS Conf. on FMS*, Aug. 1984.

[33] J.A. Buzacott and J.G. Shanthikumar. Models for understanding flexible manufacturing systems. *Trans. AIIE*, 12(4):339–350, December 1980.

[34] J. Solberg. Analytic performance evaluation for the design of flexible manufacturing systems. In *Proc. Conf. Dec. and Contr.*, 1979.

[35] C. Whitney. Control problems in flexible manufacturing. In *Proc. 1984 Conf. Dec. and Contr., San Diego*, 1984.

[36] R. Suri and R.R. Hildebrant. Modelling flexible manufacturing systems using mean value analysis. *Jour. of Manuf. Sys.*, 3(1):27–38, 1984.

[37] S. Gershwin, R. Akella, and Y. Choong. Short term production scheduling of an automated manufacturing facility. In *Proc. 23rd Conf. Dec. and Contr.*, Dec 1984.

[38] S. Gershwin. Manufacturing systems modeling and control. In *Proc. IEEE Conf. on Robotics and Automation*, April 1986.

[39] J. Kimemia and S. Gershwin. An algorithm for the computer control of a flexible manufacturing system. *IIE Trans.*, 15(4):353–362, Dec 1983.

[40] C.L. Beck. *Modeling and Simulation of Flexible Control Structures for Automated Manufacturing Systems*. Technical Report, Robotics Institute, Carnegie-Mellon University, 1985.

[41] D. Dubois and K.E. Stecke. Using petri nets to represent production processes. In *Proc. 22nd IEEE Conf. Dec. and Contr.*, pages 1062–1067, December 1983.

[42] M. Kamath and N. Viswanadham. Applications of petri net based models in the modelling and analysis of flexible manufacturing systems. In *Proc. 1986 IEEE Int. Conf. on Robotics and Automation*, pages 312–317, April 1986.

[43] G. Cohen, D. Didier, J.P. Quadrat, and M. Viot. A linear-system-theoretic view of discrete-event processess and its use for performance evaluation in manufacturing. *IEEE Trans. Automatic Control*, March 1985.

[44] M.S. Fox, B.P. Allen, S.F. Smith, and G.A. Strohm. *ISIS: A Constraint-Directed Reasoning Approach to Job Shop Scheduling*. Technical Report CMU-RI-TR-83-8, Intelligent Systems Lab., The Robotics Institute, Carnegie-Mellon University, 1983.

[45] E. Denert. *Trends in Information Processing Systems. 3rd Conference of the European Cooperation in Informatics*, chapter Software Engineering: Experience and Convictions, pages 16–35. Springer-Verlag, October 1981.

[46] S.N. Woodfield, H.E. Dunsmore, and V.Y. Shen. The effect of modularization and comments on program comprehension. In *5th International Conference on Software Engineering*, pages 215–23, March 1981.

[47] L. Varga. Specifications of reliable software. *Tanulmanyok Magy. Tud. Akad. Szamitastech. And Autom. Kut. Intez. (Hungary)*, (113):309–25, 1980.

[48] P.B. Hansen. *The Architecture of Concurrent Programs*. Prentice Hall, 1977.

[49] J. Van DenBos, R. Plasmeijer, and J. Stroet. Process comm. based on input specifications. *ACM Trans. of Programming Languages & Systems*, 3:224–250, July 1981.

[50] G.R. Andrews. Synchronizing resources. *ACM Trans. of Programming Languages & Systems*, 3(4):405–430, October 1981.

[51] J.A. Feldman. High level programming for distributed computing. *Comm. of the ACM*, 22(6):353–367, June 1979.

[52] R. Taylor and P. Wilson. Process-oriented language meets demands of distributed processing. *Electronics*, 55(24):89–95, November 1982.

[53] M. Tsukamoto. *Language Structures and Management method in a distributed real-time environment*, pages 103–13. Pergamon, August 1981.

[54] W.V. Ruggiero and G. Bressan. A programming model for distributed computing. In *Actas del la segunda conferencia international in ciencia de la computacion*, pages 97–111, August 1982.

[55] R.C. Holt. A short introduction to concurrent Euclid. *Sigplan Not.*, 17(5):60–79, May 1982.

[56] M.T. Liu and Chung-Ming Li. Communicating distributed processes: a language concept for distributed programming in local area networks. In *Local Networks for Computer Communications, IFIP Working Group 6.4, International Workshop on Local Networks*, pages 375–406, August 1980.

[57] T. Christopher, O. El-Dessouki, M. Evens, H. Harr, H. Klawans, P. Krystosek, R. Mirchandani, and Y. Tarhan. Salad-a distributed compiler for distributed systems. In *1981 International Conference on Parallel Processing*, pages 50–7, August 1981.

[58] P.E. Lauer and M.W. Shields. Cosy-an environment for development and analysis of concurrent and distributed systems. In *Symposium on Software Engineering Environments*, pages 119–56, June 1980.

[59] P.B. Hansen. Edison—a multiprocessor language. *Software—Prac. and Exper.*, 11(4):325–361, April 1981.

[60] R.P. Cook. *mod—a language for distributed programming. *IEEE Trans. Software Eng.*, SE-6(6):563–71, November 1980.

[61] T.W. Mao and R.T. Yeh. Communication port: a language concept for concurrent programming. *IEEE Trans. Software Eng.*, SE-6(2):194–204, March 1980.

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

[63] 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.

[64] 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.