Journal of Systems Integration, 1, 7-34 (1991)

Developing Control and Integration Software for Flexible Manufacturing Systems*

NEJIB BEN HADJ-ALOUANE
Advanced Technologies Laboratory, The University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109-2110

JARIR K. CHAAR
Advanced Technologies Laboratory, The University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109-2110

ARCH W. NAYLOR
Advanced Technologies Laboratory, The University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109-2110

Received October 10, 1990. Revised January 10, 1991.

Abstract. The slow growth of computer-integrated manufacturing is attributed to the complexity of designing and implementing their control and integration software. This article expands on a methodology for designing and implementing this software that was introduced in [16]. The goal of this methodology is to build flexible and reusable control and integration software for computer-integrated manufacturing systems. It hinges upon the concepts of software/hardware components, their assemblages, a distributed common language environment, formal models, and generic controllers. Major sources of flexibility are obtained by decoupling process plan models from the model of the factory floor and by using a generic controller. Reusability is achieved by building self-contained software/hardware components with general, possibly parametrized, interfaces. The interplay between simulated and actual hardware internals of software/hardware components is used as the basis of a testing strategy that performs off-line simulation followed by on-line testing.

The methodology has been applied in designing and implementing the control and integration software of an actual Prismatic Machining Cell. The article also reports on the details of this implementation.

Key words: computer-integrated manufacturing systems, automation, software components, models.

1. Introduction

Not too many years ago there was considerable enthusiasm for computer-integrated manufacturing systems, but in recent years this enthusiasm has died down, indeed almost disappeared in some companies, as more and more difficulties have been revealed. At one extreme are all the mundane problems of just being able to connect things together, at the other are the cultural problems of changing company organization to accommodate major increases in levels of automation. In between is the daunting problem of designing and implementing the required software for such systems. This problem is perhaps the major reason for the slow growth in computer-integrated manufacturing, and is addressed here. In particular, it is the general problem of developing control and integration software for real-time distributed systems.

*The names of the authors appear in alphabetical order.

Our starting point is the claim—that many would agree with—that current practices for such software are archaic and often result in a high cost, and extended development period, and extremely inflexible systems. We argue—as many have [4]—that the key is software reusability based on the careful design of software components and their assemblages. However, we go beyond these customary ideas to argue that developing reusable software for the *real-time distributed control of computer-integrated manufacturing systems* requires additional concepts and approaches. In particular, we claim that:

1. *Control and integration* software need to be *segregated*,
2. *Generic* control algorithms are necessary to reduce the amount of labor that is involved in reusing control software, and *formal models* are needed to provide the necessary support for the design and implementation of these algorithms,
3. The first concept is best achieved by developing control software on a single computing platform and then *distributing* it a later stage,
4. *Simulation* that captures the real-time aspects of manufacturing devices is needed for the testing of control and integration software of manufacturing systems.

Computer-integrated manufacturing systems differ from other, more familiar, distributed computer systems in a fundamental way: In addition to the software that constitutes traditional distributed computer systems, computer-integrated manufacturing systems involve manufacturing devices and *mechanical interactions* among such devices. This difference is behind the need for the four concepts listed above.

The structural diagram of Figure 1 illustrates our view of a typical computer-integrated manufacturing system. A and B are two software components driving mechanical devices (referred to later in this article as software/hardware components); these two components are assembled to form a larger third component. Within the assembly, the devices of A and B can interact mechanically; moreover, the nature of this interaction depends on the way these devices are configured on the factory floor.[1] More importantly, the nature of the top-level control software of the assembly (illustrated in Figure 1 by a box marked "control software") in turn depends on the nature of the interactions that occur between the devices of A and B, and, hence, on the factory floor configuration. Consequently, if this configuration is changed, the top level control software of the assembly may need to be modified in order to maintain the original functionality of the system.

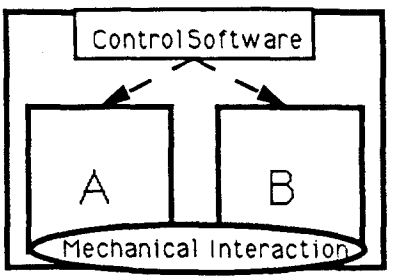

Figure 1. An assembly.

Computer-integrated manufacturing software is typically distributed and involves integration as well as control software; integration software handles the communication between the factory floor computers and the interfaces with the device controllers. Without proper care, control software and integration software may be entangled. This entanglement may present a hindrance to the reuse of computer-integrated manufacturing software (i.e., identifying appropriate control sofware and modifying it when the target factory floor configuration differs from the original one can become a considerably difficult task). Hence, the need for the first concept.

The second concept—generic control algorithms—can be used to eliminate the need for control software modifications whenever computer-integrated manufacturing software is reused. These algorithms are supplied with a formal model of the underlying system and take control decisions by consulting this model. This idealistic view, however, has not yet been fully achieved. Instead, a restricted, and yet powerful, version has been implemented and is presented in this article.

In our current implementation, generic algorithms take process plan models as data. Hence, control and integration software can be easily and quickly reconfigured to make new products, a much-needed feature in flexible computer-integrated manufacturing systems. These systems are designed to produce a wide variety of products that are not all known in advance. Switching to a new set of products is usually made on a short-term notice and must happen within a short time span.

The third concept advocates developing control software on a single computing platform and distributing it at a later stage. This approach has three main advantages. First, it provides for the segregation of control software and integration software, since the two are developed separately. Second, it facilitates the software development process since it rids the programmer from having to think across processor boundaries, and instead concentrate on the real programming task. Third, this approach can benefit from automatic program distribution tools such as the Ada-distributed translator developed at The University of Michigan [19].

Although it is widely recognized that testing distributed software is a complex task due to the interspersed nature of this software, this complexity is further augmented in testing computer-integrated manufacturing control and integration software by the presence of manufacturing devices and their mechanical interactions. On-line testing of control and integration software is often not practical because of expense and danger. This process is expensive because it prolongs the factory idle time associated with it. It is dangerous because the bugs in the software may cause harmful system behaviors. Our remedy consists developing a real-time simulation for each manufacturing device and testing the control and integration software *off-line* by interfacing it to these simulations. Once a reasonable level of confidence in the correctness of the control and integration software has been gained, the real manufacturing devices are incrementally incorporated in place of their simulations. Furthermore, alternating between the simulations and the actual devices is easily done and follows naturally from our use of software/hardware components.

The concepts of our methodology have been presented in [2, 11, 16]. Furthermore, these concepts have been validated by applying it to the design and implementation of the control software of several systems [2, 3]. This article reports on the application of this methodology to the design and implementation of the control and integration software of

a full-fledged real manufacturing system: a *Prismatic Machining Cell* of a major automobile manufacturer. A highly-efficient implementation of this software has been carried out in the Ada programming language. Some implementation details proved to be important issues. In particular, adequate solutions to the one-way naming and multiple inheritance problems (not supported by Ada) have proven necessary to designing and implementing reusable software components and are discussed in this article.

The article is organized as follows. The next section reviews the relevant work in the literature. This is followed by elaborating upon the various concepts of our methodology. Next, a description of the Prismatic Machining Cell, including devices, layout, and functionality, is provided. The last section reports on the design and implementation of the control and integration software of this cell. We conclude by outlining the important concepts and future research directions.

2. Related Work

Research in the area of manufacturing systems is heavily concentrated in modeling; very few articles are actually concerned with control and integration software, and even fewer are concerned with making this software reusable and flexible. The most popular models of manufacturing systems are based on extensions of Petri Nets [6, 8, 12–14, 17, 18]. Among those that actually implement control software are [12], GRAFCETs [18] and PROT nets [6, 8].

Crockett et al. [12] is based on Petri net hierarchies. Places in these nets are either simple or macro places. Macro places are themselves Petri nets. The controller, developed in C, associates a C procedure with every place in the net and acts as a Petri net interpreter by executing this procedure whenever a token arrives in the place.

A similar approach is adopted by Thomas and McLean [18]. GRAFCET, an extension to Petri nets is used. In a GRAFCET, a place is associated with an action, a macro place is itself a GRAFCET and a transition is associated with a condition. Both conditions and actions are coded as C expressions and procedures, respectively.

A methodology based on Process Translatable (PROT) nets that supports the specification, rapid prototyping, and simulation of manufacturing systems is reported in [6, 8]. PROT nets are extensions to Petri nets that associate attributes with tokens and model hierarchy by enclosing other PROT nets in net transitions. PROT nets are translated into Ada program structures to be used as the basis for control software prototyping [5, 6, 8]. The same nets can be translated into OPS5 rules to derive a production schedule for the system [7, 9, 10].

A major shortcoming of the above approaches is the intermix in their models of process plans, control, and cell operation. This intermix results in a rigid cell controller (i.e., a change in a process plan or the control strategy calls for major changes in the controller). In addition, software reusability is not a major issue, and the issues of simulation and online testing are not considered.

3. The Design and Implementation Methodology

To achieve the goal of developing *reusable* control and integration software our methodology introduces the concept of software/hardware components. A software/hardware component is a generalized version of a software component [16]. A software component is an object-oriented construct that is characterized by a set of public *software interfaces* and a *body* (*internals*) [4].

A software interface specifies a set of services that can be performed by this component. Users can capture a distinct view of the component by accessing the services in a subset of its software interfaces. The internals of the component implement the serivces listed in its interfaces. The structure of software components promotes the well-known principle of information hiding. This is achieved by separating the interfaces from the internals and making the internals inaccessible to the users.

A software/hardware component generalizes the concept of a software component by allowing the internals of the component to enclose hardware [20]; that is, these internals are interfaced to and drive hardware devices. Moreover, the specification of a software/hardware component provides a software interface, in the common implementation language, to the hardware it drives. As an example, the Ada specification of a robot software/hardware component is given as follows. This specification lists the main functions performed by the robot.

```
WITH Cell_Definitions;
PACKAGE Robot IS
   TYPE R_Status IS (Moving, Idle); -- Status of the robot
   TYPE G_Status IS (Opened, Opening, Closed, Closing); -- Status of the gripper
   SUBTYPE Acknowledgement IS Cell_Definitions.Acknowledgement;
   SUBTYPE Location IS Cell_Definitions.Location;

   FUNCTION Pick_Up RETURN Acknowledgement;
   FUNCTION Put_Down RETURN Acknowledgement;
   FUNCTION Move(Source, Destination: IN Location) RETURN Acknowledgement;
   FUNCTION Robot_Status RETURN R_Status;
   FUNCTION Gripper_Status RETURN G_Status;
END Robot;
```

In addition to the software interface, a software/hardware component presents its users with a hardware interface. This hardware interface consists of the portion of the component's hardware that is accessible to and interacts with the external environment. A hardware interface can be as simple as the gripper of a robot, such as in the above software/hardware component, or, as complex as a set of automatically guided vehicles operated by a material transport system. Figure 2 is a schematic diagram of a software/hardware component. Both software and hardware interfaces are windows to the component. Each can be independently used by the external environment to control and monitor the component's operation. However, they usually provide different functionalities and views of their component.

Assembling software/hardware components involves, from a software perspective, interconnecting their software interfaces and, from a hardware perspective, interconnecting their hardware interfaces. Manufacturing devices are designed as software/hardware components. Manufacturing cells are assemblages of software/hardware components. Cells together with their controllers are in turn assembled into factories. The decisions

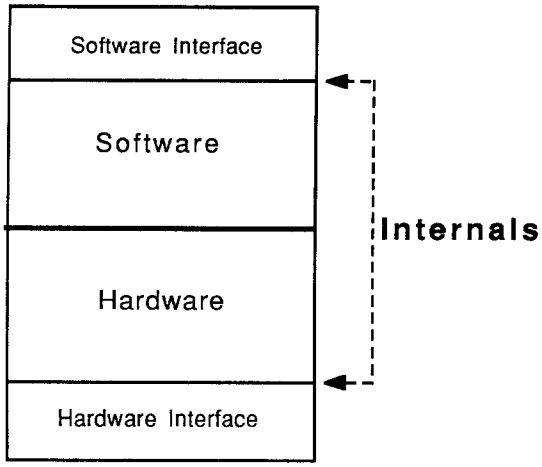

Figure 2. A software/hardware component.

involved in designing and assembling software/hardware components into reuseable control and integration software are discussed next.

3.1. Designing Reusable Software/Hardware Components

Our methodology identifies two main features that are essential in promoting the reusability of a software/hardware component. First, the software interface of the component should be general and parametrized whenever possible. Second, the software/hardware component should be self-contained. The software interface of the foregoing robot software/hardware component is general because it does not reveal either the type of robot used or the details associated with implementing the services of this robot. These services can be carried out by a large class of robots. Hence, any one of these robots could be enclosed by the component without affecting its users' software; only the internals of the component need be changed to interface with the enclosed robot.

Self-containment is the decoupling of a component from its potential users' programs. This means that the component shall not request any services from any other component (not part of its internals) while other components, its users, can use its services. Hence, the component should not be aware of any name outside its domain; a situation we refer to as *one-way naming*. In reality, however, many systems require interacting with their external environments. As a result, it may seem that their software/hardware components cannot be made reusable. Fortunately, this is not the case. A solution enforcing one-way naming while still allowing components to interact with their environments is provided in [3]. This solution is detailed as follows due to the surprisingly important role one-way naming plays in designing and implementing reusable software/hardware components.

The nature of the one-way naming problem is a function of the interaction of a software/hardware component with its environment. Two different situations can be identified.

First, a component cannot avoid requesting services from its user; this is exemplified by a cell software/hardware component requesting the factorywide material transport system software/hardware component to remove pallets at its output dock. The cell component is intended to be operated in conjunction with a multitude of factorywide material transport systems. The problem is that both name and specifics of the remove operation may differ from one material transport system to the other. The solution consists of creating an interfacing software component between the cell component and the factorywide material transport system component—call it the intermediate component. The interface of this intermediate component lists the services required by the cell from the material transport system component, among which is the remove pallet from output dock operation. The internals of this intermediate component are assigned depending on the specifics of the material transport system in use. Hence, a change of the material transport system coupled with the cell requires modifying only the internals of the intermediate component.

In the second situation, the component operates in a multi-user environment where each user can request services from the component by calling a procedure listed in the component's interface. Furthermore, the nature of the request may necessitate that the component notifies the originator of the request of the results whenever they become available. The problem is that these results may not become available until long after the called procedure has completed. Hence, the component must call a procedure of the user component to deliver these results. This is not a satisfactory solution because it requires that the component be tailored to the specifics of its environment (e.g., know the names of some procedures of the user's component). Instead, a solution involving the use of a mailbox system is adapted as follows: a mailbox is associated, at run-time, with each user of the component. A user's mailbox is used to deposit the results of a given user request. These results are then retrieved by the threads of control spawned by this user.

Although dealing with the preceding issues can be considered essential to designing reusable software in general, additional concepts and approaches are definitely required in developing reusable control and integration software for computer-integrated manufacturing systems, discussed as follows.

3.2. Building Generic Controllers

Adopting a hierarchical control structure is key to enchancing the reusability of software/hardware components; this structure blends naturally with our use of software/hardware components and their assemblages and also simplifies considerably the control scheme implemented by the control and integration software.

Figure 3 illustrates this hierarchical control structure. A set of software/hardware components, whether enclosing manufacturing devices, cells, or factories, is assembled and coupled with a control strategy to form another software/hardware component; the constituents are considered at a lower level than their assembly. The software implementing the control strategy of the assembly component plans, executes, and monitors the operations of its constituent software/hardware components.

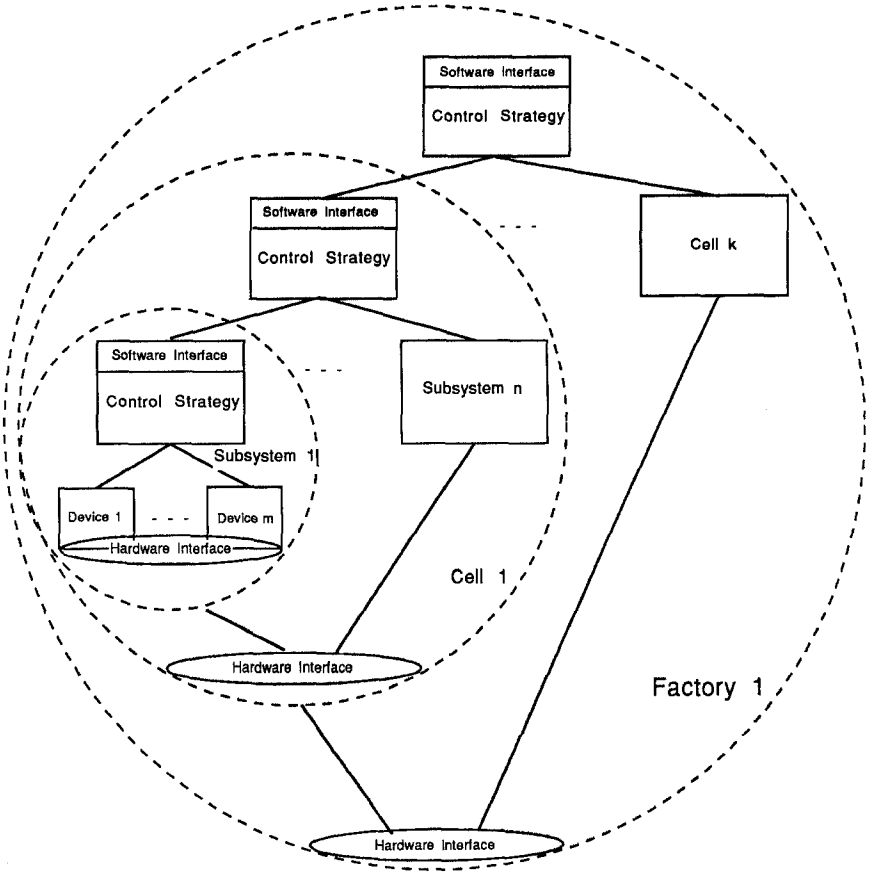

Figure 3. A hierarchical control structure.

An assembly software/hardware component can be viewed as a subsystem (see Figure 3) of a computer-integrated manufacturing system; this subsystem is operated through its software and hardware interfaces; its internals enclose the control strategy (adopted by the subsystem) implemented. In particular, when Ada is the implementation language, the body of an assembly component package (labeled Control Strategy in Figure 3) encloses and is completely devoted to implementing its application-specific control strategy; therefore, this package is the only Ada unit that needs to be modified and recompiled whenever the control strategy of an assembly component is modified.

Modifying the software that implements the control strategies of assembly components, whenever they are reused, may involve major efforts especially when these components are sizable (e.g., a cell component). Our ultimate goal is to develop a generic controller that can be used in controlling any set of manufacturing software/hardware components [16]. Figure 4 illustrates our perception of this generic controller. The role of the generic controller is to generate the control strategy for the assembly software/hardware component.

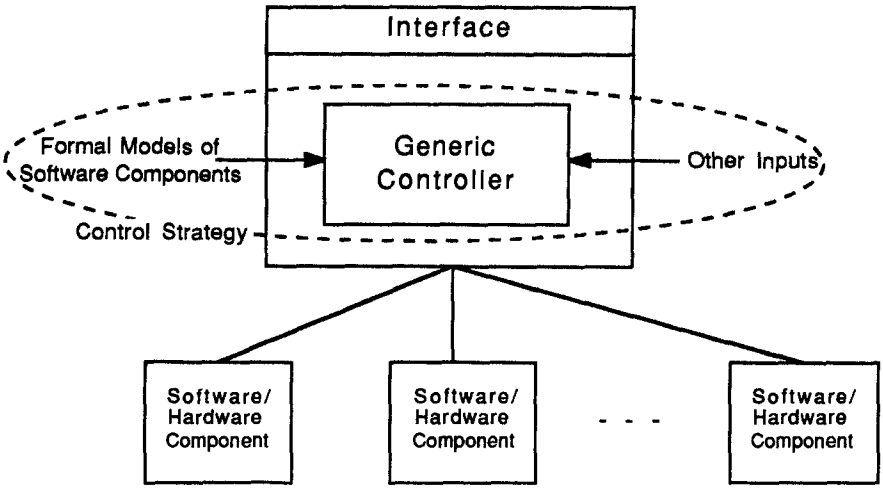

Figure 4. The generic controller of a software/hardware component.

Our generic controller is a software component that accepts as inputs the formal models of the constituent components of an assembly together with directions on how to assembly them and executes the orders received through the interface of the assembly component. The other inputs that may be needed specify, for example, models of process plans and any control objectives that need to be achieved.

Our implementation of the Prismatic Machining Cell partly achieved the above goal of building generic controllers by developing a simplified version of the generic controller of Figure 4. The structure of this controller is shown in Figure 5. This controller is

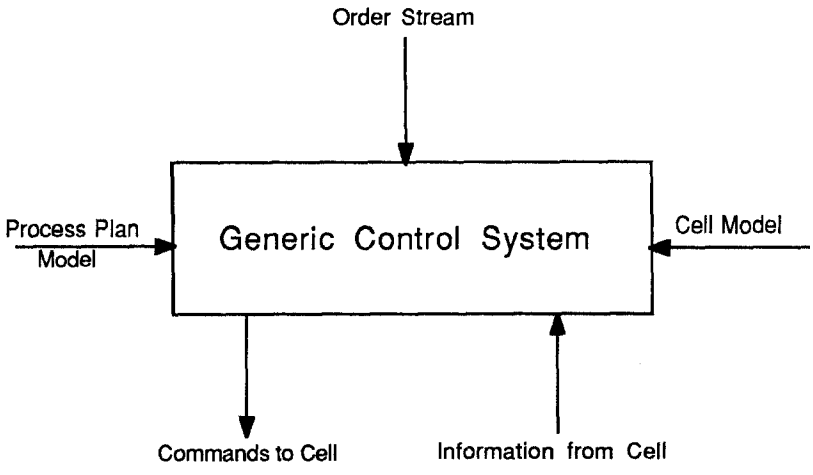

Figure 5. The prismatic machining cell generic controller.

presented with a model of the cell together with a model of the process plans and a stream of orders. These inputs are used to determine, based on the current status of the cell and orders, the appropriate commands to be executed by the cell software/hardware component. The workings of this controller are described in the next section. In the meantime, we describe the general structure of the formal models [15] used in building this cell controller.

For our purpose, the formal model of a software/hardware component captures, at the logical level [15], its functionality as viewed through both its software [16] and its hardware interfaces. The logical level is, as the name suggests, largely concerned with logical conditions and transformations of logical conditions. A typical condition might be stated as "The robot is at the machining center." Transformations of conditions involve actions such as moving the robot from one place to the other. The logical level is captured by a simple first-order logic rule-based model. In this model, the state consists of a set of predicates (relations) and constants (names). Rules, reminiscent of the familiar Artificial Intelligence paradigm, are used to describe the state transitions. A rule consists of a set of preconditions followed by a set of postconditions and is associated with a logical variable (input) and a time delay. Whenever the preconditions are satisified by the current state of the system and the rule is commanded by enabling its logical variable, its postconditions are satisfied by the state of the system after the specified time delay.

3.3. Performing Simulation

An added advantage that is gained from casting manufacturing hardware into software/hardware components is the ability to alternate their simulation and on-line testing. This alteration reduces considerably the idle time of the manufacturing devices that can be manually operated while the control and integration software of the system is being independently developed and tested. Furthermore, the complexity of the task of testing the control and integration software is reduced by eliminating hardware-related errors from the software testing process.

Simulation is performed by assigning simulated internals to the hardware-dependent components of a manufacturing system. This constitutes the first step in testing the control and integration software of the system. The thoroughness of the testing process depends on how well the simulated internals capture the mechanical interactions of the hardware devices of the system. The next step gradually replaces each simulated internal by an interface to the real hardare device. On-line testing is completed whenever the whole system is fully operational.

Our simulation is performed in real-time, as opposed to GPSS-like event-driven simulations, by using the concurrency and timing constructs offered by the implementation language—in this case Ada. To further illustrate this point, let us consider a simulation of the internals of the robot software/hardware component previously specified. An Ada task is used to simulate the movement of the robot while a second task emulates the operations of its gripper. These two tasks execute in parallel because the robot can move and operate its gripper at the same time. The duration of a given operation is simulated by a delay statement that has the effect of blocking the execution of its associated task for a specified period of time and then resuming normal execution afterwards.

3.4. Distributed Common Language Environment

The run-time environment for control software is inevitably distributed. It is composed of a number of computers and device controller that are connected via a communication network. Hence, a portion of the control software will be executing on several nodes of the network.

Each portion, together with the required interprocessor communication, could be designed and implemented independently of the other portions. This approach has many disadvantages. First, partitioning the control software at an early stage of the design and development process across processor boundaries might not result in the most logical partition, and, more importantly will seriously affect the reusability of this software; a change in the underlying network architecture, which is most likely to happen when the software is ported to another system, might require the complete redesign of this software. Second, coupling the design of the control software with the design of the communication software runs the risk of entangling them together and further affects the reusability of the control software. Third, it is well known that testing distributed software is much harder and more complex than testing nondistributed software. One of the factors that contribute to this complexity is the inability to perform compile-time error and data-consistency checking across processor boundaries.

We opted for designing and implementing the control software of computer-integrated manufacturing systems as a single program written in a single high-level language. Distributing this software across the network is performed after the implementation and testing stages are completed; the appropriate communication software is inserted either by hand or automatically by means of a distributed translator [19]. This approach avoids all the problems previously stated; in addition, the programmer can concentrate on the task of implementing the control software without being concerned with its distributed aspects.

Nevertheless, the single-platform approach is typically associated with two main disadvantages; however, with proper care their effects can be considerably reduced, if not totally averted. Communication overhead is the most important concern when dealing with the foregoing approach: Distributing a single program (in our case, the control software) across a network involves transforming some of the local procedure and function calls into remote ones; usually, the total communication delay entailed by the execution of these remote calls is greater than the delay involved had the program been developed in a distributed form. Although the above assertion is true, its significance can be considerably reduced by the choice of an efficient network and communication protocol ([16, 19]). Moreover, critical areas, where communication traffic is particularly heavy, should be identified and the corresponding software should be appropriate tuned. The second concern involves the transition from a sequential environment to a distributed one: for obvious reasons, the correct execution of a distributed program does not usually imply the correct execution of a distributed version of this program. Although the foregoing statement is true in general, it certainly is false in the case of a program consisting of assemblages of software components destined for multiuser environments.[2]

The high-level language used must support the development of software components and must easily interface with the variety of device-specific languages used by manufacturing

device controllers. There are several languages that meet these requirements such as Ada, C^{++}, and Modula-3. We opted for Ada because, in addition to meeting the previous requirements, it provides language constructs for expressing concurrency and other real-time features that faciliate the task of simulation of the internals of software/hardware components.

Nonetheless, Ada proved to be deficient in a very important aspect of the assembly process of software/hardware components. During this process, the software designer should be able to designate one or more data objects, that are defined in the constituent components of the assembly, as the same object in the assembly component. Ada does not unfortunately support this feature. Our solution, using the current Ada features, consists of avoiding the use of *private* data types in the packages that implement the various software/hardware components of a system, and declaring the data objects that are common to a set of components in the environment package of these components.

4. The Prismatic Maching Cell

The Prismatic Machining Cell, as its name suggests, is intended for making a wide variety of prismatic parts. Many parts are clamped on a standardized pallet and are machined together. The cell layout is shown in Figure 6. This cell is composed of the following devices: a Cincinnati Milacron T-10 Machining Center, a Cincinnati Milacron Shuttle, a Brown and Sharpe 1057 PCR Coordinte Measuring Machine, a GMF S-400 six-axis pedestal robot, and three Load/Unload stations. A brief description of the functionality of each device follows.

- The Machining Center is capable of milling, drilling, boring, reaming, and tapping metal workpieces. It is equipped with a single spindle, a machining table, an automatic tool changer, and a 45-tool belt. The tool mounted on the spindle can operate on a set of workpieces grouped on a pallet that is automatically clamped to the machining table. The automatic tool changer can exchange the tool in the spindle with another tool from the tool belt. The Machining Center is operated through a Cincinnati Milacron Acramatic 950 CNC controller.
- The Coordinate Measuring Machine is equipped with a probe that is used to inspect a set of workpieces grouped on a pallet that is automatically clamped to a rotary table. The Coordinate Measuring Machine is operated through a DEC Microvax II computer.
- The Shuttle is a two-slot rotating turntable. Each slot can hold one pallet and is equipped with an automatic load/unload mechanism. The rotation angle of the turntable is an integer multiple of 90°. Hence, the shuttle can service up to four surrounding tables. The shuttle is operated through an Allen-Bradley Programmable Logic Controller.
- The Robot is used to transfer pallets among the shuttle, the coordinate measuring machine, and the accessible load/unload stations. The robot is operated through a built-in controller.
- The Load/Unload station are standard tables used to hold the pallets in the cell. They are equipped with sensors that can detect the presence/absence of a pallet. The Prismatic Machining Cell is equipped with a Manual Load/Unload (L/U) station and a Robot Load/Unload (L/U) station.

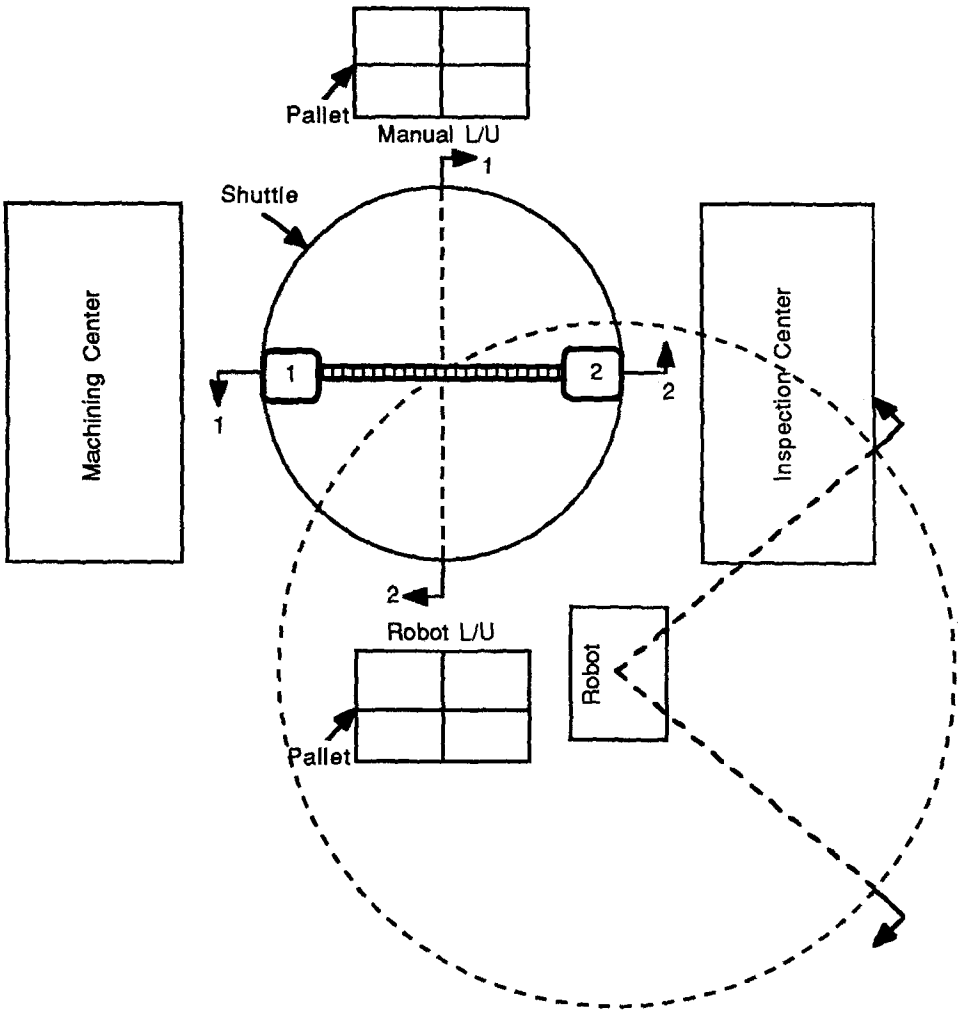

Figure 6. Prismatic machining cell layout.

The Prismatic Machining Cell is controlled through an IBM PC/AT computer (the cell controller) that is connected with the various device controllers of the cell through a MAP carrier band network. The control software executing on the IBM PC/AT issues appropriate commands to each device controller via the network. These device controllers can, in turn, issue requests and report status information, via the network, to the cell control software. The Prismatic Machining Cell control software operates the cell as follows.

The cell operator specifies a part type, the number of parts to be manufactured, and the process plan to be used in manufacturing these parts. Based on this information, the control software requests the appropriate raw materials, usually on pallets, from a factorywide (manual or automated) material transport system. Pallets enter the cell through either one of the L/U stations shown in Figure 6. The operations performed on these

pallets within the cell are determined by the steps of the corresponding process plan. A process plan step may specify a machining or an inspection operation; the details pertaining to such operations (e.g., part programs, inspection programs, and tools) are enclosed in the process plan step. Upon completing the operations specified by a process plan on a particular pallet of parts, the finished product is removed from the cell via an L/U station. The later operation is carried out by the factorywide material transport system.

Our implementation of the Prismatic Machining Cell control software allows the simultaneous processing of several pallets of parts. Moreover, the current set of pallets need not use the same process plan (i.e., several batches of parts can be processed in parallel).

5. The Cell Control Software

A hierarchical structure diagram of the control and integration software of the Prismatic Machining Cell is shown in Figure 7. As mentioned previously, this structure results from assembling together the constituent components of the cell. First, the software/hardware

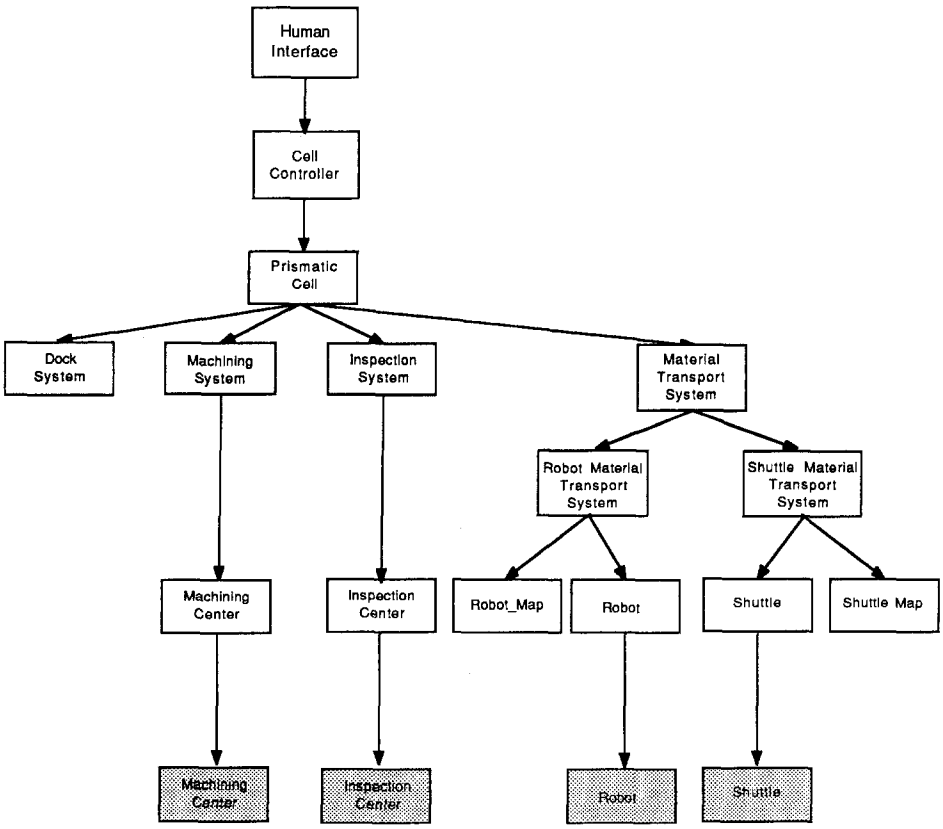

Figure 7. Prismatic machining cell control hierarchy.

components that directly enclose the physical devices (shown as shaded boxes) of the cell are created and named to reflect the devices they enclose (i.e., *Machining Cener, Inspection Center, Robot,* and *Shuttle*). Next, the *Robot_Map* and *Shuttle_Map* software components are formed, implementing the databases enclosing the road maps for the robot and the shuttle, respectively. This is followed by assembling both Robot_Map and Robot into the *Robot Material Transport System* software/hardware component and both Shuttle_Map and Shuttle into the *Shuttle Material Transport System* software/hardware component. These two material transport systems are then assembled into a single *Material Transport System* software/hardware component. At the same time, the *Dock System* software/hardware component is created to directly enclose the two Load/Unload stations in the cell, and, a more abstract view of both Machining Center and Inspection Center is also presented by the *Machining_System* and *Inspection_System* components, respectively. The Dock System, Machining System, Inspection System, and Material Transport System components are next assembled to form the *Prismatic Cell* software/hardware component. The latter component, together with the cell control strategy, form the *Cell Controller* software/hardware component. Finally, the *Human Interface* software component presents the users of the cell with a software interface. The software assembly steps performed during the design of the cell software/hardware components result in the foregoing hierarachy where the components that present similar functional views can be thought of as belonging to the same level in the hierarchy. We describe next the major components of Figure 7.

5.1. The Machining System Software/Hardware Component

The Ada specification of the Machining System software/hardware component is shown as follows. This component offers its users the capabilities of executing a set of part programs and querying for the status of the Machining Center. The center will be *busy* when executing a part program and *idle* otherwise. The details associated with the execution of a part program are not of interest to the user of the component, and, are hidden in the component's internals.

```
WITH Machining_Center;
PACKAGE Machining_System IS
   TYPE Machining_System_Status IS (Busy, Idle);
   SUBTYPE Part_Program IS Machining_Center.Part_Program;
   SUBTYPE Acknowledgement IS Machining_Center.Acknowledgement;

   FUNCTION Execute_Part_Program(Part_Program_name: Part_Program)
    RETURN Acknowledgement;
   FUNCTION Query_Machining_System_Status RETURN Machining_System_Status;
END Machining_System;
```

The set of part programs that can be executed by the Machining Center is stored in a database. A code segment and a data segment are associated with each part program. These are declared as private types to hide their internal data structures from the user of the database and the database is implemented as a generic package. Both the Machining Center and

the Factory are users of the database. The Machining Center is provided with a read-only view of this database. It can query for the availability of and download part programs into its local memory. On the other hand, the Factory is presented with the complete view of the database. It is responsible for populating it with new part programs and deleting or updating old part programs.

While a part program is executing, it can request the use of a set of tools from the machining center tool belt. A local tool database keeps track of the set of tools on the tool belt together with their associated data. Again, this database is implemented as a generic Ada package and private types are used to hide the details of the tool records. The Machining Center requests the Factory component to bring tools to or remove tools from the belt; this is an example of a first situation one-way naming problem. The tool database is updated accordingly whenever these requests are executed.

To execute a part program, the availability of this part program is checked with the database. If available, the program is loaded into the local memory. This operation is followed by loading the set of tools to be used by the part program on the tool belt of the Machining Center. If a tool is not loaded on the belt, a request is issued to the factory to bring it to the belt. The new tool is either placed in an empty slot on the belt or swapped with a tool from the belt that is not to be used by the current part program. All these complicated operations are simulated by Ada tasks that rendezvous with each other to perform the requested operations. The actual execution of the part program can only be started when the program is loaded into the controller, all the required tools are loaded on the belt and the first tool is mounted on the spindle of the Machining Center.

5.2. The Inspection System Software/Hardware Component

The Ada specification of the Inspection System software/hardware component is very similar to the specification of the Machining System software/hardware component and is shown as follows. This component offers its users the capabilities of executing a set of inspection prorams and querying for the status of the Coordinate Measuring Machine. This Machine will be *busy* when executing an inspection program and *idle* otherwise. The details associated with the execution of an inspection program are not of interest to the user of the component and are hidden in the component's internals.

```
WITH Inspection_Center;
PACKAGE Inspection_System IS
   TYPE Inspection_System_Status IS (Busy, Idle);
   SUBTYPE Inspection_Program IS Inspection_Center.Inspection_Program;
   SUBTYPE Acknowledgement IS Inspection_Center.Acknowledgement;

   FUNCTION Execute_Inspection_Program(Inspection_Program_name:
     Inspection_Program) RETURN Acknowledgement;
   FUNCTION Query_Inspection_System_Status RETURN Inspection_System_Status;
END Inspection_System;
```

The set of inspection programs that can be executed on the Coordinate Measuring Machine is stored in a database identical to that of the Machining Center. The execution of an

inspection program is similar to that of a part program with the exception that no tools are needed and a single inspection probe is used.

5.3. The Material Transport System Software/Hardware Component

The Material Transport System software/hardware component is an assemblage of the Robot Material Transport System component and the Shuttle Material Transport System component. As their names suggest, the robot material transport system encloses the robot and the shuttle material transport system encloses the shuttle. A shuttle is characterized by its number of slots, hence, this number is specified as a parameter of the generic package implementing the Shuttle software/hardware component, a subcomponent of the Shuttle Material Transport System. Each of these two material transport systems is responsible for transferring pallets among a subset of the locations of the cell; a single transfer can take place at any given time within either robot or shuttle material transport system. The two subsets covered by the robot and shuttle material transport systems are labeled the shuttle map and the robot map, respectively. Each map is implemented as a database. The topology of the cell can be easily changed by appropriately updating the map databases of either robot or shuttle material transport systems.

The Ada specification of the Material Transport System software/hardware component is shown as follows. It offers its users the capabilities of transferring pallets between the various locations of the cell. The user of this component can check the feasibility of a pallet transfer operation and obtain an estimate of the time it takes to perform this operation. He or she can also query for the status of an ongoing transfer and the time remaining for this transfer to complete. In accordance with our philosophy, both the details of the map and the details associated with executing a feasible pallet transfer between a pair of locations of this map should not be revealed by the interface of the Material Transport System component. In our case, however, the full adoption of this strategy would result in a nondeterministic software interface for the Material Transport System Component; this nondeterminism can complicate the design of the cell controller. Hence, the Material Transport System interface reveals enough details as to eliminate any nondeterministic aspects (the structure of the cell material transport system map is revealed as a union of robot and shuttle material transport system maps). This decision is a typical example of the trade-off encountered when dealing with abstraction.

```
WITH  Cell_Definitions;
WITH Robot_Material_Transport_System;
WITH Shuttle_Material_Transport_System;
PACKAGE Material_Transport_System IS
   TYPE Move_Type IS PRIVATE;
   TYPE Move_Status IS (In_Progress, Done);
   SUBTYPE Acknowledgement IS Cell_Definitions.Acknowledgement;
   SUBTYPE Location IS Cell_Definitions.Location;
   SUBTYPE Pallet_Type IS Cell_Definitions.Pallets;
   SUBTYPE Status_of_Map1_Moves IS
    Shuttle_Material_Transport_System.Shuttle_Material_Transport_System_Status;
   SUBTYPE Status_of_Map2_Moves IS
    Robot_Material_Transport_System.Robot_Material_Transport_System_Status;
```

```
    PROCEDURE Move_Pallet(Source_Station, Destination_Station: IN Location;
     P_Name: IN Pallets; P_Type: IN Pallet_Type; Started: OUT Acknowledgement;
     M_Id: OUT Move_Type);
    PROCEDURE Estimate_Move_Time(Source_Station, Destination_Station: IN Location;
              Estimated_Time: OUT Duration; Feasible: OUT Acknowledgement);
    PROCEDURE Status_Of_Move(M_Id: IN Move_Type; Remaining_Time: OUT Duration;
     M_Status: OUT Move_Status);
    FUNCTION Status_of_Shuttle_Material_Transport_System
     RETURN Status_of_Map1_Moves RENAMES
    Shuttle_Material_Transport_System.Status_of_Shutle_Material_Transport_System;
    FUNCTION Status_of_Robot_Material_Transport_System
     RETURN Status_of_Map2_Moves RENAMES
    Robot_Material_Transport_System.Status_of_Robot_Material_Transport_System;

    PRIVATE
      -- The implementation of the above private types.
END Material_Transport_System;
```

In order to transfer pallets from a source location to a destination location, both locations are reserved and access to the pallet on the source location is secured for the transport vehicle responsible for carrying out the pallet transfer operation. Next, the pallet is loaded on the vehicle, the source location is freed, and the transfer is started. After a predetermined travel time, the vehicle reaches the destination location where the pallet is unloaded, access to it is cleared, and the destination location freed. The scheduling algorithm associated with the Material Transport System gives priority for the use of the robot over the shuttle in transferring pallets between locations that are common to both maps.

5.4. The Dock System Software/Hardware Component

The Ada specification of the Dock System software/hardware component is as follows. It offers its users the capabilities of bringing a raw pallet of a specified type into the cell and removing a finished pallet from the cell. These services are carried out in collaboration with the factorywide material transport system. The services of this material transport system are invoked from within the internals of the Dock System component. When these services are completed (i.e., the specified pallet has been brought or removed), the factorywide material transport system notifies the Dock System component by calling either Pallet_Brought or Pallet_removed. Moreover, the user of this component can locate a given pallet within the cell or check the status of the locations of this cell by testing the values of the sensors of these locations.

```
WITH Cell_Definitions;
WITH Sensors;
PACKAGE Dock_System IS
    SUBTYPE Pallet_Type IS Cell_Definitions.Pallet_Type;
    SUBTYPE Pallets IS Cell_Definitions.Pallets;
    SUBTYPE Location IS Cell_Definitions.Location;
    SUBTYPE Acknowledgement IS Cell_Definitions.Acknowledgement;
    SUBTYPE Sensor_Status IS Sensors.Sensor_Status;
```

```
    FUNCTION REmove_Pallet(Pallet: IN Pallets; Source: IN Location)
     RETURN Acknowledgement;
    FUNCTION Bring_Pallet(Pallet: IN Pallet_Type; Source: IN Location)
     RETURN Acknowledgement;
    FUNCTION Removing_Pallet(Source: IN Location; Pal_Name: IN Pallets;
     Pallet: IN Pallet_Type) RETURN Acknowledgement;
    PROCEDURE Removed_Pallet(Source: IN Location);
    FUNCTION Bringing_Pallet(Destination: IN Location) RETURN Acknowledgement;
    PROCEDURE Brought_Pallet(Destination: In Location; Pallet: IN Pallet_Type);
    PROCEDURE Locate_Pallet(P_Name: IN Pallets; Station: OUT Location;
      Found : OUT Acknowledgement) RENAMES Sensors.Locate_Pallet;
    PROCEDURE Query_Station(Station: IN Location; P_Name: OUT Pallets;
      P_Type: OUT Pallet_Type; State : OUT Sensor_Status)
      RENAMES Sensors.Query_Station;
END Dock_System;
```

5.5. The Cell Controller Software/Hardware Component

The structure diagram of the Cell Controller component is shown in Figure 8. This component controls the operation of the Prismatic Cell software/hardware component, a simple union of the Dock System, Machining System, Inspection System, and Material Transport System component (i.e., the interface of the Prismatic Cell component is the union of the interfaces of its constituents, and its internals are also the union of the internals of its

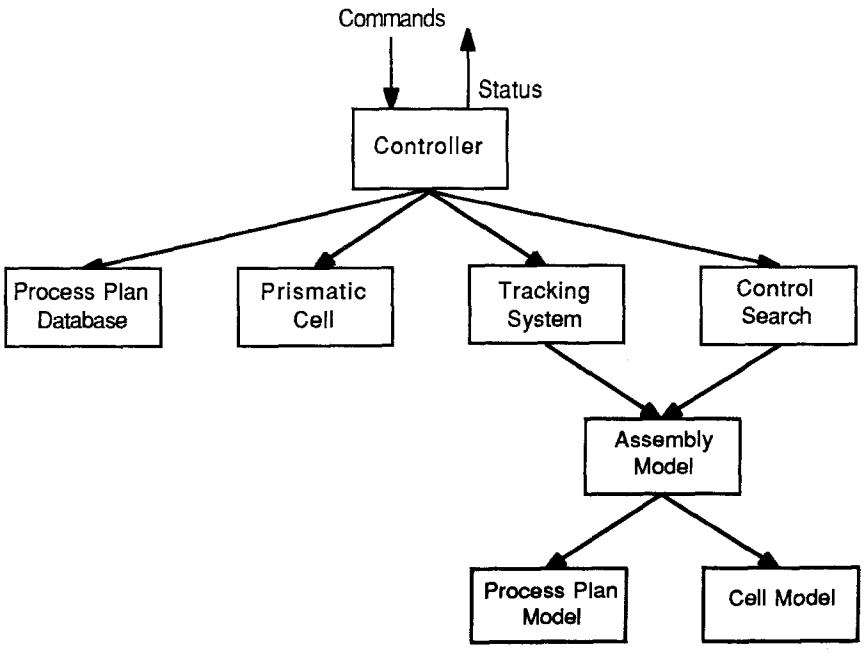

Figure 8. The structure diagram of the prismatic machining cell controller.

constituents). The Cell Controller accepts commands, through the Human Interface component (Figure 7), to make a batch of pallets of a certain type. Each pallet type is associated with a unique process plan that describes the making of this type of pallets and is stored in the process plan database shown in Figure 8. Status information, including the current state of the cell and its devices, is constantly reported by the Cell Controller component to the cell operators.

The internals of the Cell Controller component implement a generic control strategy for planning, executing and monitoring the sequence of actions performed by the Prismatic Machining Cell. This control strategy is based on the models of both the Prismatic Cell software/hardware component and the process plans. First, a brief and informal discussion of these models is given. This is followed by outlining the generic control strategy.

5.5.1. The Prismatic Cell Model. In a manner analogous to the construction of the Prismatic Cell software/component, its model of behavior, as observed through its software interface, is constructed as the simple union of the models of its constituent components (i.e., the set of predicates and constants that form the state of this model is the union of the corresponding sets of the constituent models; moreover, the set of rules of this model is the union of the corresponding sets of the constituent models). These constituent models, in turn, capture the behavior of their software/hardware components, as observed through the software interfaces of these components. They are presented in the following sections.

Machining System Model. The predicates and constants of the Machining System model are:

- MACHINING_CENTER, IDLE, EXECUTE_PROGRAM, and COMPLETE_PROGRAM are the constants of the model.
- Part_Program(.), Pallet(.), and Location(.) are unary predicates true of all part programs, pallets, and locations, respectively. These predicates are referred to as *static* predicates because the relations they denote do not change as the system evolves in time.
- $I_{execute(.,.)}$ is a binary predicate input to the model true when the rule is commanded. This predicate is *dynamic* because the relation it denotes is allowed to change with time. Moreover, some of these changes may be caused by the controller of the model.
- CT(.,.) is a binary predicate that captures the general notion of contact between entities of the model, physical or otherwise.

The software interface of the Machining System software/hardware component is composed of two functions. One function is a query; the dynamics of its execution are not relevant for the purpose of controlling the component and, hence, are omitted from the model. Note, however, that the result of this query is captured by a well-formed formula whose truth of falsity can be checked within the context of the current state. On the other hand, the dynamics of the Execute_Part_Program function are captured by the following rule:

- Execute_Part_Program(*pp*, *p*) executes part program *pp* to machine the pallet *p* currently on the machining table. This rule is executed whenever it is commanded by the controller, pallet *p* is an MACHINING_CENTER, MACHINING_CENTER is IDLE, and

program *pp* can be executed. The execution of this rule occurs in two stages. The first stage takes place instantaneously and marks both MACHINING_CENTER and *pp* as busy. The second stage occurs t_{pp} time units later and frees MACHINING_CENTER and *pp*.

$$\left\{\begin{array}{l} I_{execute(pp,p)} \\ Part\_Program(pp) \wedge Pallet(p) \\ (\forall l)(Location(l) \wedge CT(MACHINING\_CENTER,\ l) \rightarrow CT(p,l)) \\ CT(MACHINING\_CENTER,\ IDLE) \\ CT(pp,\ EXECUTE\_PROGRAM) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{execute(pp,p)} \\ \neg CT(MACHINING\_CENTER,\ IDLE) \\ \neg CT(pp,\ EXECUTE\_PROGRAM) \end{array}\right\} (t),$$

$$\left\{\begin{array}{l} CT(MACHINING\_CENTER,\ IDLE) \\ CT(pp,\ COMPLETE\_PROGRAM) \end{array}\right\} (t + t_{pp})$$

Inspection System Model. The model of the Inspection System software/hardware component is analogous to that of the Machining System model. Hence, only the dynamics of the Execute_Inspection_Program rule are presented.

- Execute_Inspection_Program(*ip*,*p*) executes program *ip* to inspect the pallet *p* currently on the inspection table.

$$\left\{\begin{array}{l} I_{execute(ip,p)} \\ Inspection\_Program(ip) \wedge Pallet(p) \\ (\forall l)(Location(l) \wedge CT(INSPECTION\_CENTER,\ l) \rightarrow CT(p,l)) \\ CT(INSPECTION\_CENTER,\ IDLE) \\ CT(ip,\ EXECUTE\_PROGRAM) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{execute(ip,p)} \\ \neg CT(INSPECTION\_CENTER,\ IDLE) \\ \neg CT(ip,\ EXECUTE\_PROGRAM) \end{array}\right\} (t),$$

$$\left\{\begin{array}{l} CT(INSPECTION\_CENTER,\ IDLE) \\ CT(ip,\ COMPLETE\_PROGRAM) \end{array}\right\} (t + t_{ip})$$

Material Transport System Model. The Material Transport System has two sets of locations $Map_1 = \{L_1, L_2, L_3, L_4\}$ and $Map_2 = \{L_3, L_4, L_5\}$. Pallets can be moved between any two locations within each map. L_3 is common to both maps, and is used to move pallets between locations in different maps. Only one move can be progressing within a given map at any given time. The dynamics of Move_Pallet are captured by three rules. The first rule applies when both source and destination are located in Map_1 and are not

common to both maps. The second rule applies when both source and destination are located in Map_2 and are not common to both maps. The third rule applies whenever a transfer of pallets takes place between locations common to both maps.

- Map_1 Moves. Move_Pallet(s, d, p) models the transfer of pallet p from location s to location d. Both s and d are in Map_1. s or d can be in Map_2 but not both.

$$\left\{\begin{array}{l} I_{move(s,d,p)} \\ Map_1(s) \wedge Map_1(d) \\ \neg(Map_2(s) \wedge Map_2(d)) \\ CT(p, s) \wedge \neg(\exists y)(Pallet(y) \wedge CT(y, d)) \\ \neg CT(s, RESERVED) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_1, ON\_GOING) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{move(s,d,p)} \\ CT(s, RESERVED) \\ CT(d, RESERVED) \\ CT(MOVE\_Map_1, ON\_GOING) \end{array}\right\} (t),$$

$$\left\{\begin{array}{l} \neg CT(p,s) \\ \neg CT(s, RESERVED) \end{array}\right\} (t + t^1_{sd}) \left\{\begin{array}{l} CT(p,d) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_1, ON\_GOING) \end{array}\right\} (t + t^2_{sd})$$

- Map_2 Moves. Move_Pallet(s, d, p) models the transfer of pallet p from location s to location d. Both s and d are in Map_2.

$$\left\{\begin{array}{l} I_{move(s,d,p)} \\ Map_2(s) \wedge Map_2(d) \\ CT(p, s) \wedge \neg(\exists y)(Pallet(y) \wedge CT(y, d)) \\ \neg CT(s, RESERVED) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_2, ON\_GOING) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{move(s,d,p)} \\ CT(s, RESERVED) \\ CT(d, RESERVED) \\ CT(MOVE\_Map_2, ON\_GOING) \end{array}\right\} (t),$$

$$\left\{\begin{array}{l} \neg CT(p,s) \\ \neg CT(s, RESERVED) \end{array}\right\} (t + t^1_{sd}) \left\{\begin{array}{l} CT(p,d) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_2, ON\_GOING) \end{array}\right\} (t + t^2_{sd})$$

- Common Moves. Move_Pallet(s, d, p) models the transfer of pallet p from location s to location d. Both s and d are in Map_1 and Map_2.

$$\left\{\begin{array}{l} I_{move(s,d,p)} \\ Map_1(s) \wedge Map_1(d) \\ Map_2(s) \wedge Map_2(d) \\ CT(p, s) \wedge \neg(\exists y)(Pallet(y) \wedge CT(y, d)) \\ \neg CT(s, RESERVED) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_1, ON\_GOING) \\ CT(MOVE\_Map_2, ON\_GOING) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{move(s,d,p)} \\ CT(s, RESERVED) \\ CT(d, RESERVED) \\ CT(MOVE\_Map_1, ON\_GOING) \end{array}\right\} (t),$$

$$\left\{\begin{array}{l} \neg CT(p,s) \\ \neg CT(s, RESERVED) \end{array}\right\} (t + t^1_{sd}) \left\{\begin{array}{l} \mathrm{CT}(p,d) \\ \neg CT(d, RESERVED) \\ \neg CT(MOVE\_Map_2, ON\_GOING) \end{array}\right\} (t + t^2_{sd})$$

Dock System Model. The dynamics of Bring_Pallet and Remove_Pallet of the Dock System model are captured as follows:

- Bring_Pallet(t, l) loads a pallet of type t of raw parts at input location l of the cell. Predicate AE(.) marks the parts that are currently being processed within the cell. Type_off(.,.) indicates the type of a pallet. t_b is a random variable that depends on when the factorywide material transport system honors the request to bring a pallet.

$$\left\{\begin{array}{l} I_{bring\_pallet(t,l)} \\ Pallet\_Type(t) \\ Location(l) \wedge \neg CT(l, RESERVED) \\ (\forall x)(Pallet(x) \wedge \neg CT(x, l) \\ (\exists y)(Pallet(y) \wedge Type\_of(y, t) \wedge \neg AE(y)) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{bring\_pallet(t, l)} \\ AE(y) \\ CT(l, RESERVED) \end{array}\right\} (t), \quad \left\{\begin{array}{l} \neg CT(l, RESERVED) \\ CT(y, l) \end{array}\right\} (t + t_b)$$

- Remove_Pallet(p, l) unloads a pallet p of machined parts from output location l of the cell. Predicate AE(.) marks the parts that are currently being processed within the cell. t_r is a random variable that depends on when the factorywide material transport system honors the request to remove a pallet.

$$\left\{\begin{array}{l} I_{remove\_pallet(p,\ l)} \\ Pallet(p) \wedge Location(l) \\ CT(p,\ l) \wedge \neg CT(l,\ RESERVED) \end{array}\right\}(t),\ \mapsto$$

$$\left\{\begin{array}{l} \neg I_{remove\_pallet(p,\ l)} \\ CT(l,\ RESERVED) \end{array}\right\}(t),\ \left\{\begin{array}{l} \neg CT(l,\ RESERVED) \\ \neg AE(p) \\ (\forall x)(\neg CT(x,\ p)) \end{array}\right\}(t + t_r)$$

5.2.2. Process Plans Model. Traditionally, the set and sequence of operations that need to be followed when manufacturing a product are specified by a process plan. In our view, the model of such a process plan consists in part of an acyclic directed graph (the fixed part of the model). The nodes of this graph represent process plan steps, and the arcs indicate the precedence relation that exists among these steps. Each step specifies a manufacturing operation; in the Prismatic Machining Cell, the specification of such an operation involves naming either a part program or an inspection program. In addition, a process plan model tracks the current status of each part being manufactured in accordance with this plan. Moreover, the model captures the dynamics involved in carrying a part through the process plan (traversing a path of the graph).

A single model is required to capture all the process plans currently being used in the cell. The unary predicates Process_Plan(.) and Process_Plan_Step(.), and the binary predicates Predecessor(.,.) and CT(.,.) represent the fixed part of the model. Process_Plan and Process_Plan_Step are true, respectively, of all process plans and process plan steps currently in use. CT indicates which steps belong to which process plan (e.g., CT(pp, pps) means that *pps* is a step of process plan *pp*). Predecessor indicates the precedence relation among the steps of each process plan (e.g., Predecessor(pps1, pps2) means that step *pps*1 must be executed prior to executing *pps*2).

The dynamics of the processing plan model are, in turn, captured by the following rules:

- Start_Process_Plan(*pp*, *pps*, *p*) starts the execution of the first step *pps* of process plan *pp* on pallet *p*.

$$\left\{\begin{array}{l} I_{start\_execute(pp,pps,p)} \\ Pallet(p) \wedge \neg CT(p,\ COMPLETED\_PLAN) \\ Process\_Plan(pp) \wedge Process\_Plan\_Step(pps) \wedge CT(pp,\ pps) \\ (\exists z)(Pallet\_Type(z) \wedge Type\_of(p,\ z) \wedge \neg CT(z,\ p)) \\ (\forall x)(Process\_Plan\_Step(x) \wedge \neg CT(x,\ p)) \\ (\forall y)(Process\_Plan\_Step(y) \wedge \neg Predecessor(y,\ pps)) \end{array}\right\}(t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{start\_execute(pp,pps,p)} \\ CT(p,\ pp) \\ \neg CT(p,\ COMPLETED\_STEP) \end{array}\right\}(t)$$

- Start_Process_Step(*pp*, *pps*, *p*) starts the execution of a step *pps* of process plan *pp* on pallet *p*. Process plan step *pps* should not be the first step of process plan *pp*.

$$\left\{\begin{array}{l} I_{start\_execute(pp,pps,p)} \\ Pallet(p) \wedge CT(p, COMPLETED\_STEP) \\ Process\_Plan(pp) \wedge Process\_Plan\_Step(pps) \wedge CT(pp, pps) \\ (\exists y)(Process\_Plan\_STEP(y) \wedge Predecessor(y, pps) \wedge CT(y,p)) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{start\_execute(pp,pps,p)} \\ CT(p, y) \wedge CT(p, pps) \\ \neg CT(p, COMPLETED\_STEP) \end{array}\right\} (t)$$

- Complete_Process_Step(*pp*, *pps*, *p*) completes the execution of a step *pps* of process plan *pp* on pallet *p*. Process plan step *pps* should not be the last step of process plan *pp*.

$$\left\{\begin{array}{l} I_{complete\_execute(pp,pps,p)} \\ Pallet(p) \wedge Process\_Plan(pp) \wedge Process\_Plan\_Step(pps) \\ CT(pp, pps) \wedge CT(p, pps) \wedge \neg CT(p, COMPLETED\_STEP) \\ (\exists y)(Process\_Plan\_STEP(y) \wedge Predecessor(pps, y)) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{complete\_execute(pp,pps,p)} \\ CT(p, COMPLETED\_STEP) \end{array}\right\} (t)$$

- Complete_Executing_Last_Process_Plan_Step(*pp*, *pps*, *p*) completes the execution of the last step *pps* of process plan *pp* on pallet *p*.

$$\left\{\begin{array}{l} I_{complete\_execute(pp,pps,p)} \\ Pallet(p) \wedge Process\_Plan(pp) \wedge Process\_Plan\_Step(pps) \\ CT(pp, pps) \wedge CT(p, pps) \wedge \neg CT(p, COMPLETED\_STEP) \\ (\forall x)(Process\_Plan\_Step(x) \wedge Predecessor(pps, x)) \end{array}\right\} (t) \mapsto$$

$$\left\{\begin{array}{l} \neg I_{complete\_execute(pp,pps,p)} \\ CT(p, COMPLETED\_STEP) \\ CT(p, COMPLETED\_PLAN) \end{array}\right\} (t)$$

A process plan database is used to store the graph structure (and other pertinent information) associated with all the process plans that may be used during the course of operation of the Prismatic Machining Cell. A database management system is used to add, delete, and update process plans and perform other useful operations on this database. Whenever a process plan is required by the Prismatic Cell controller, this database is queried and the appropriate process plan is added to the current process plans model (the predicates Process_Plan, Process_Plan_Steps, Predecessor, and CT are set accordingly). This

process plan is removed from the process plans model whenever it is no more required by the controller of this cell; this process plan, however, may still exist in the process plan database).

5.5.3. The Control Strategy. In our implementation, an assembly of the Prismatic Cell model and process plans model (Figure 8) is presented to the controller as a set of procedures, each implementing a rule of the model; the details of the process used to assemble these models are described in [1].

The Control Search component (Figure 8) consults the Assembly Model to determine the current set of "useful" rules; this set is then reported to the controller. A "useful" rule is defined as one that is executable, given the current state of the cell (as indicated by the Tracking System), and whose execution can eventually lead to executing a process plan step. The set of "useful" rules is determined by searching a tree of depth n, the maximum number of rules needed to start a process plan step. The nodes of this tree are states of the Assembly Model. A node S_p is a predecessor of a node S_s if and only if the state captured by S_s can be reached from the state captured by S_p by executing a rule of the assembly model. The search is performed in a depth-first manner. In order to improve the efficiency of the search procedure (performed on-line), the data structure enclosing the search tree is built off-line (once and for all) and is simply updated each time it is traversed, whenever a new set of "useful" rules is to be determined.

The Controller selects from a given set of "useful" rules a subset of "nonconflicting" ones. Two rules are labeled "conflicting" if they cannot be executed in parallel (their postconditions can be satisfied at the same time); this can occur, for example, if the execution of these rules requires th same resource. The commands corresponding to the subset of "nonconflicting" rules are issued by the controller to the cell component. The state of the tracking system is then updated in accordance to the resulting state of the prismatic machining cell. The control search component is then instructed to generate a new set of "useful" rules by consulting the current state of the Tracking System (Figure 8).

6. Conclusion

In this article, the concepts of a methodology for designing and implementing the control and integration software of computer-integrated manufacturing systems are presented. The goal of this methodology is to build flexible and reusable software. Software flexibility is obtained by decoupling the process plan models from the factory floor model and using a generic control algorithm. Reusability is achieved by building self-contained software/hardware components with general, possibly parametrized, interfaces. These reusable components can be used to populate manufacturing software libraries. Off-the-shelf components can then be assembled into manufacturing systems. Moreover, the interplay between simulated and actual hardware internals of software/hardware components is used as the basis of a testing strategy that performs off-line simulation followed by on-line testing.

The application of the methodology to the design and implementation of the control and integration software of a Prismatic Maching Cell is also reported. A highly efficient implementation of this software has been carried out in the Ada programming language and is currently fully operational. Surprisingly, some implementation details proved to be important issues. In particular, adequate solutions to the one-way naming and multiple inheritance problems (not supported by Ada) are necessary for designing and implementing reusable software components.

Our planned future work includes three main directions. First, the development of software tools for specifying and cataloging software/hardware components. Second, building fully generic controllers for computer-integrated manufacturing systems; the goal of building generic controllers requires the design and implementation of a language for expressing our formal models together with its associated compiler. Third, developing user-friendly human interfaces to computer-integrated manufacturing software; we believe that these interfaces, whenever designed and implemented, can be easily coupled with our current control and integration software.

Notes:

1. For instance, two robots sharing a common work space have the possibility of crashing into each others. This possibility can be eliminated, however, by placing them further apart.
2. Carefully note that proper distribution of such a program is an underlying assumption of the statement. For example, consider the assembly of Figure 1, with A and B as multiuser components; a proper way of distributing the assembly involves A, B, and the top-level control software of the assembly executing on three different machines; on the other hand, splitting the top-level control software can have undesirable consequences.

References

1. N. Ben Hadj-Alouane, Ph.d. thesis proposal, "Assembly of the Formal Models of Software/Hardware Components in Flexible Manufacturing Systems," technical report, RSD-TR-11-90. Robot Systems Division, The University of Michigan, 1990.
2. N. Ben Hadj-Alouane, J. K. Chaar, and A. W. Naylor. "The design and implementation of the control and integration software of a flexible manufacturing system," in *The First Int. Conf. Systems Integration*, April 23–26, 1990.
3. N. Ben Hadj-Alouane, J. K. Chaar, A. W. Naylor, and R.A. Volz, "Material handling systems as software components: An implementation," technical report RSD-TR-10-88, Robot Systems Division—The University of Michigan, May 1988.
4. G. Booch, *Software Components with Ada: Structures, Tools, and Subsystems*, Menlo Park, CA: Benjamin/Cummings, 1987.
5. G. Bruno, "Using Ada for discrete event simulation," *Software Practice and Experience*, 14(7) pp. 685–695, July 1984.
6. G. Bruno and A. Balsamo, "Petri net-based object-oriented modeling of distributed system," in *OOP-SLA'86: Object-Oriented Programming Systems, Languages and Applications Conf. Proc.*, September 1986, pp. 284–293.
7. G. Bruno and A. Elia, "Operational specification of process control systems: Execution of prot nets using ops5," in H. J. Kugler, ed., *Information Processing (IFIP) 86*, 1986, pp. 35–40.
8. G. Bruno and G. Marchetto, "Process-translatable petri nets for the rapid prototyping of process control systems," *IEEE Trans. Soft. Eng.*, SE-12(2) February 1986, pp. 346–357.
9. G. Bruno and M. Morisio, "Petri-net based simulation of manufacturing cells," in *1987 IEEE Int. Conf. Robotics and Automation*, March 1987, pp. 1174–1179.

10. G. Bruno and M. Morisio, "The role of rule based programming for production scheduling," in *1987 IEEE Int. Conf. Robotics and Automation*, March 1987, pp. 545–550.
11. J. K. Chaar and R. A. Volz, "On the Ada implementation of a component-oriented rule-based specification language for manufacturing syste ms control software, in *Proc. Fifth Annual Conf. on Artificial Intelligence and Ada (AIDA-89)*, November 1989, pp. 39–50.
12. D. Crockett, A. Desrochers, F. DiCesare, and T. Ward, "Implementation of a petri net controller for a machining workstation," in *Proc. 1987 IEEE Int. Conf. on Robotics and Automation*, March 1987, pp. 1861–1867.
13. 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*, March 1986, pp. 312–317.
14. Y. Narahari and N. Viswanadham, "Modeling flexible manufacturing systems with map/1," in *Proc. the First ORSA/TIMS Special Interest Conf. Flexible Manufacturing Systems: Operations Research Models and Applications*, The University of Michigan, Ann Arbor, August 15–17, 1984, pp. 346–358.
15. A. W. Naylor and M. C. Maletz, "The manufacturing game: A formal approach to manufacturing software," *IEEE Trans. Systems, Man, and Cybernetics*, SMC-16(3) May/June 1986, pp. 321–334.
16. A. W. Naylor and R. A. Voltz, "Design of integrated manufacturing system control software," *IEEE Trans. Systems, Man, and Cybernetics*, SMC-17(6) November/December 1987, pp. 881–897.
17. R. Ravichandran and A. K. Chakravarty, "Decision support in flexible manufacturing systems using timed petri nets," *J. Manuf. Systems* 5(2), pp. 89–101, 1986.
18. B. H. Thomas and C. McLean, "Using grafcet to design generic controllers," in *Proc. 1988 Int. Conf. Computer Integrated Manuf.*, Rensselaer Polytechnic Institute, Troy, New York, May 23–25, 1988, pp. 110–119.
19. R. A. Voltz, P. Krishnan, and R. Theriault, "An approach to distributed execution of Ada programs," in *NASA Workshop on Telerobotics*, May 1987.
20. R. A. Voltz and T. N. Mudge, "Robots are (nothing more than) abstract data types," in *Proc. SME Conf. Robotics Research: The Next 5 Years and Beyond*, August 1984, p. MS84–493, pp. 1–16.