

SANDIA REPORT

SAND96-8557 • UC-405

Unlimited Release

Printed August 1996

A CORBA-Based Manufacturing Environment

C. M. Pancerella, R. A. Whiteside, P. A. Klevgard

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of the contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors or subcontractors.

This report has been reproduced from the best available copy.

Available to DOE and DOE contractors from:

Office of Scientific and Technical Information
P. O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from:

National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.
Springfield, VA 22161

A CORBA-Based Manufacturing Environment

C. M. Pancerella and R. A. Whiteside
Distributed Systems Research Department
and
P. A. Klevgard
Information Based Manufacturing Systems Department
Sandia National Laboratories

Abstract

A CORBA-based distributed object software system was developed for Sandia's Agile Manufacturing Testbed (SAMT). This information architecture supports the goals of agile manufacturing: rapid response to changing requirements; small lot machining; reduction in both time and cost of the product realization process; and integration within a heterogeneous, wide-area networked enterprise. Features of the resulting software-controlled manufacturing environment are:

- Easy plug-and-play of manufacturing devices.
- Support for both automated and manual operations.
- Information flow both into and out of manufacturing devices.
- Dynamic task sequencer.

Each of the heterogeneous physical objects (lathe, milling machine, robot arm, *etc.*) has a corresponding software object that supports a common IDL interface called IDevice. This interface provides operations for material processing, material movement, status monitoring, and other administrative tasks. CORBA objects allow for the encapsulation of a machine tool, its controller, and the network interface to the controller.

Both manual and automated operations are supported by the software system. If an IDevice object receives a request for a non-automated operation, it uses an associated Console object to affect the operation by communications with a human machinist. A design goal of the Console object for a machine is to provide an information-intensive environment for the machinist, rather than just the transmittal of instructions to be carried out.

In addition to the flow of information into manufacturing devices (e.g., control and NC code), our software architecture supports the easy extraction of data (e.g., sensor data or inspection reports) back out of the machine and into the broader information processing environment.

The task sequencer object dynamically locates devices, accepts jobs, and dispatches tasks in the manufacturing cell. A job script captures setup operations, material movement, and processing. Though the task sequencer coordinates cell activities, many operations can be accomplished intelligently by the devices. For example, all material transfer is performed as peer-to-peer object interaction, independent of the supervisory control of the task sequencer.

Ongoing work involves integrating this Testbed management system with process planning, integrated design, and inventory control systems.

Table of Contents

1.0	Introduction	7
1.1	Agile Manufacturing	7
1.2	Sandia Agile Manufacturing Testbed	8
1.3	SAMT Cell Management	10
1.4	Distributed Objects	11
1.5	Common Object Request Broker Architecture (CORBA)	12
1.6	OMG Interface Definition Language (IDL)	13
2.0	Interfaces for Manufacturing Devices	15
2.1	The IDevice Interface	16
2.2	The IBaseDev Interface	17
2.3	The IRunDev Interface	18
2.4	The IMovePart Interface	20
2.5	Automated and Manual Operation in Idevice	21
3.0	IDevice Implementations: Cdevice	22
4.0	Integration of Devices in a Manufacturing Cell	24
4.1	Cell Controller Graphical User Interface	25
4.2	Cell Sequencer	26
4.3	Persistent Objects for Device Setup Instructions and Cell Results	29
5.0	Conclusions and Future Work	29
6.0	Acknowledgments	31
7.0	References	31

1. INTRODUCTION

The last decade has seen increasing awareness of the critical nature of software in the manufacturing automation process [7]. In February 1996, the *Communications of the ACM* devoted an entire volume to the role of computer science in manufacturing [13]. An agile manufacturing facility imposes special requirements including support for human interaction with machining devices, plug-and-play of manufacturing devices with little to no downtime in service, ease of upgrading machine controllers, and information flow into and out of manufacturing devices. Our experience has shown that a distributed computing environment is a particularly effective way of achieving these ends.

In the next sections we discuss agile manufacturing, introduce our agile manufacturing testbed, and give an overview of CORBA, an industry standard for distributed object systems. This serves as background for a more detailed description of our software design and implementation.

1.1 AGILE MANUFACTURING

Increasing competitiveness in manufacturing worldwide has spurred efforts to increase production flexibility and permit rapid setup for smaller lot sizes. *Agile manufacturing* [10] must support changes in product mix, batch size, manufacturing processes, customer requirements, and technology and at the same time provide cost-effectiveness, reduced cycle times, and high quality and accuracy. An agile manufacturing process spans the product realization life cycle: conception, design, analysis and simulation, planning (specifically, process, assembly, and inspection planning), fabrication, inspection, delivery, and possibly maintenance.

An implication of an agile manufacturing environment is that the design and manufacturing facilities though perhaps physically distributed, are networked and well-integrated within an enterprise. The enterprise is most often heterogeneous with respect to computer platforms, operating systems, network capabilities, and custom and commercial software. Further, it may be geographically distributed across a wide area, possibly internationally. The information architecture itself must be scaleable, interoperable, and reconfigurable. Internet technology, electronic data exchange, and industry standards for interoperability will be core to the infrastructure.

1.2 SANDIA AGILE MANUFACTURING TESTBED

The *Sandia Agile Manufacturing Testbed (SAMT)* [11] is a manufacturing research facility at Sandia National Laboratories in Livermore, California. The project objective is to develop agile manufacturing processes for various machined and welded products. Figure 1 shows the product realization cycle in the SAMT environment: design, planning, cell management, and fabrication. During design, engineers use computer-aided design (CAD) tools, simulation tools, and analysis tools to design parts that meet the customers' requirements. The planning phase includes planning for fabrication, assembly and inspection; fixturing, tooling, and analysis tools assist the process engineer during this stage. The cell management activities, described in greater detail in the next section, include scheduling, tracking, and job dispatching to the shop floor. This cycle is by no means sequential, but rather iterative; for example, incomplete designs may be planned to guarantee that a part is manufacturable. Furthermore, information from planning and fabrication must be stored, analyzed, and accessible to designers at a later time.

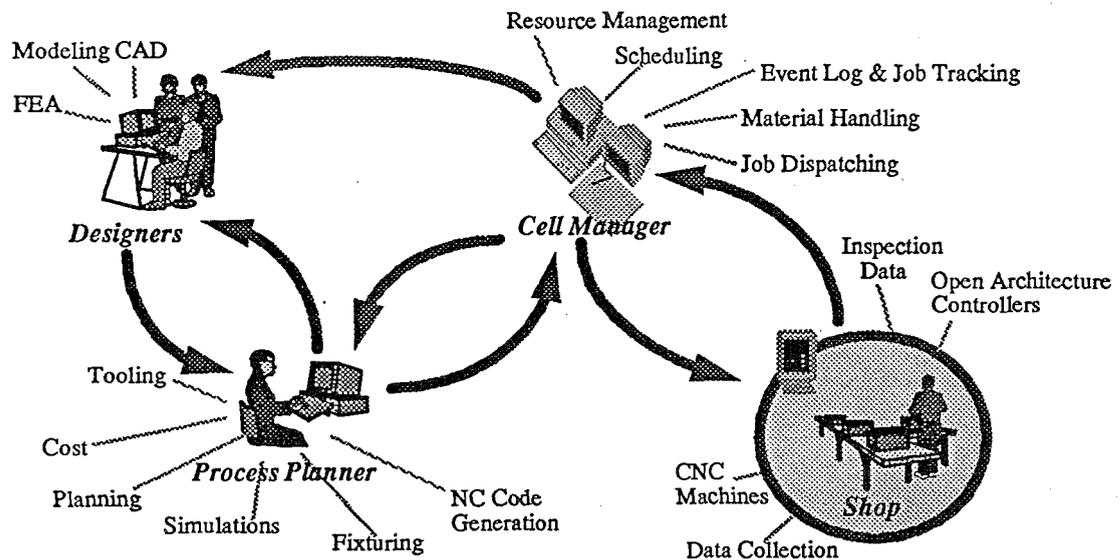


Figure 1. SAMT Product Realization Process.

The physical component in the SAMT is a networked manufacturing cell (Figure 2.) containing a conventional milling machine (Cincinnati 4-axis mill), a milling machine (Haas 4-axis mill) equipped with an open architecture controller, a lathe, a gas tungsten arc (GTA) welder, a coordinate measuring machine (CMM), various storage devices, and a Stäubli

robot which services some of the manufacturing and storage devices. Each of the manufacturing devices (lathe, milling machines, CMM, welder) has a computer—sometimes shared—which has two functions:

- as a network connection to the machine controller, allowing programs and control commands to be downloaded and executed in an automated fashion, and
- as an operator console for electronic data to be delivered to the shop floor in support of manual operations, e.g., machine setup, or as a *guardian interface* [12].

The PC and Sun Sparc 20 are used for cell management software components. Figure 2 shows the configuration of manufacturing devices, computers, and networks in the manufacturing cell.

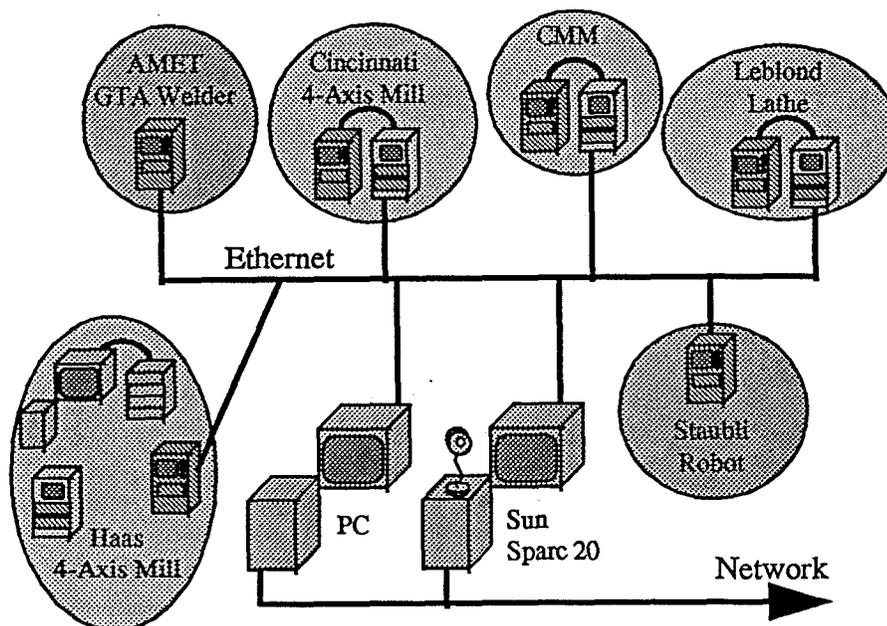


Figure 2. Agile Manufacturing Cell Configuration.

One of the goals of the SAMT project is to explore a narrow, vertical slice of the product realization process. This concept is illustrated in Figure 3. Throughout the product realization cycle (seen on the vertical axis of Figure 3), the interactions will be studied and optimized as manufacturing processes and product lines vary.

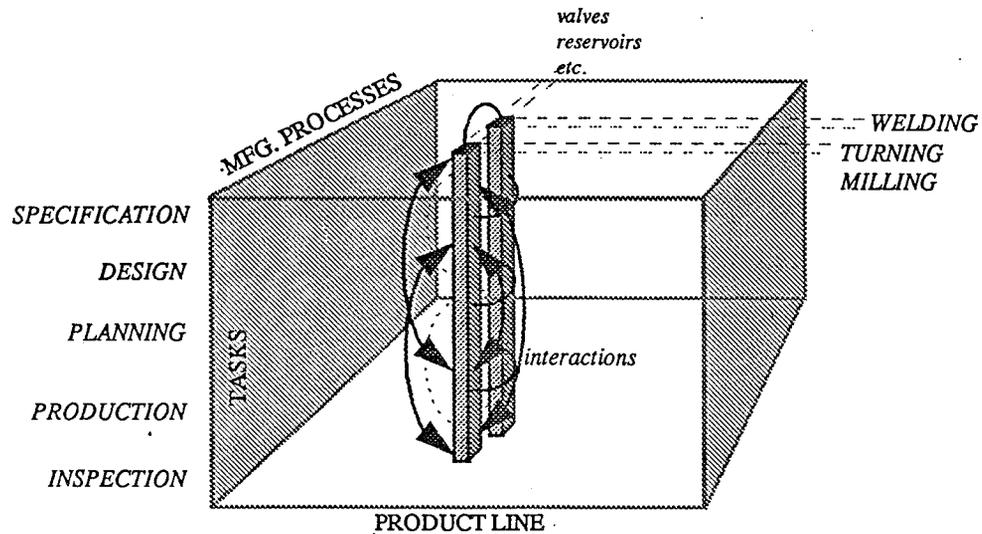


Figure 3. Narrow, Vertical Slice of Product Realization Process.

Given the project goals and the heterogeneous computing resources, there are several demands placed on software in the SAMT. Specifically, the software must support a variety of platforms, operating systems, and programming languages. The software must facilitate information flow through the product realization cycle and from the manufacturing activities to the designers and process engineers. It must support rapid and easy customization, integration, and reconfiguration. Finally, the manufacturing cell must be easily integrated into the extended enterprise such that manufacturing process data (for example, inspection reports and on-machine measurements) can be stored in a database for use by designers and process engineers in the future.

1.3 SAMT CELL MANAGEMENT

The focus of this paper is on software development for the cell management activities, specifically the interfaces to manufacturing devices and the dispatching of jobs to devices. We define cell management in the SAMT as the software components which are responsible for the following tasks:

- entering process plans into the cell from the SAMT process planning node or external source
- directing the development of a production plan from a process plan (*i.e.* producing a physical realization of a process plan)
- assigning production plans to be scheduled
- maintaining cell schedules, both long-term and short-term

- dispatching jobs to machines in the cell
- coordination of manufacturing devices in the cell
- event-logging of all cell activities and storage of all cell data (on-machine inspection [6], CMM inspection, etc.)
- gathering data and statistics on machining processes (tool utilization, for example)
- interface to planner for replanning of machining based on sensory input
- interface to material handling system
- interface to inventory system.

An underlying requirement of the cell management software is to first automate the flow of information to facilitate all those processes which precede and follow the actual machining of a part. Where it makes sense, the cell management software will also permit the automation of the machining itself.

1.4 DISTRIBUTED OBJECTS

Distributed object technology [8] allows computing systems to be integrated such that objects or components work together across machine and network boundaries. Examples of current distributed object or component technologies include CORBA[4], OLE[3], and OpenDoc[2].

A distributed object is not necessarily a complete application but rather a reusable, self-contained piece of software that can be combined with other objects in a plug-and-play fashion to build distributed object systems. A distributed object can execute either on the same computer or on another networked computer with other objects. Thus a client object may make a request of a server object and the operation proceeds unaffected by their respective locations. Following the principles of object-oriented design, a distributed object has a well-defined *interface*, describing the data and functionality it exposes to other objects.

There are four key principles in object-oriented programming, *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*, and each plays an important role in our CORBA-based distributed manufacturing testbed software. Abstraction is the process of refining away unimportant details of an object, so that only the appropriate attributes (data) and behaviors (methods) remain. Encapsulation means that an object publishes a public interface (of data and methods) that defines how other objects and applications can interact with it. The object's private implementation is encapsulated or

hidden from the public view. Inheritance allows a new class to be built by extending the data and methods of a previously defined class. A child class (the derived class) can inherit or receive data structures and functions from a parent class (the base class) and refine, customize, replace, or extend the simpler parent class. A chain of derived classes, each one defining its own variant of a method whose name is shared, permits polymorphic behavior: a class object will automatically invoke the correct variant of the shared method. This feature makes code modular and easy to modify.

1.5 COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

CORBA [4] is an industry middleware standard for building distributed, heterogeneous, object-oriented applications. CORBA is specified by the *Object Management Group (OMG)*, a non-profit consortium of computer hardware and software vendors. At this time, CORBA provides the best technical solution for integrating our manufacturing testbed; it is robust, heterogeneous, interoperable, multiplatform, and multivendor supported.

Figure 4 shows the CORBA reference model. The *Object Request Broker (ORB)* is the communication hub for all objects in the system; it provides the basic object interaction capabilities necessary for components to communicate. When a client invokes a method on a remote object, the ORB is responsible for marshaling arguments for the call, locating an object server, physically transmitting the request, unmarshaling the arguments in a format required by the server, marshaling return values or exceptions for the response, transmitting the response, and unmarshaling the response at the client end.

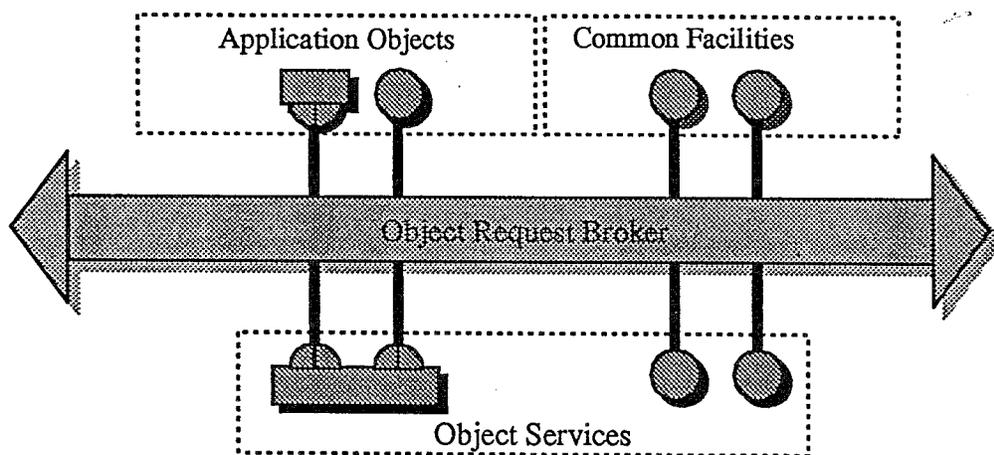


Figure 4. CORBA Object Model.

CORBA *object services* in Figure 4 are common services needed by a distributed object systems; these services add functionality to the ORB. CORBA object services include, for example, standards for object life cycle, naming, persistence, event notification, transactions, and concurrency.

Common facilities provide a set of general-purpose application capabilities for use by object systems, including accessing databases, printing files, document management, and electronic mail in a distributed system.

Finally, *application objects* in Figure 4 are the developed software which make use of the other three categories.

1.6 OMG INTERFACE DEFINITION LANGUAGE (IDL)

The OMG Interface Definition Language (*IDL*) is used to define interfaces in CORBA. An IDL interface file describes the data types, and methods or operations that a server provides for an implementation of a given object. IDL is not a programming language, but rather a language that describes only interfaces: there are no implementation-related constructs in the language. The OMG does specify mappings from IDL to various programming languages including C, C++, and Smalltalk. We will use IDL throughout this paper to show our interfaces to manufacturing devices, the task sequencer, and other CORBA objects in our system. Our particular manufacturing implementations have all been written in C++, though we have used other languages (Tcl and Visual Basic, for example) to write client-only applications which use our CORBA objects.

Though we will not give a detailed description of the OMG IDL language, it is fairly readable, and we present an informal introduction to IDL by describing a simple interface that is used in our environment called *INotify*. As its name suggests, this interface is used to communicate notifications of events. For example, when some asynchronous process is requested, success or failure may be later reported back through an object *INotify* interface. The OMG IDL for our *INotify* interface is as follows:

```
interface INotify
{
    readonly attribute string      str;
    readonly attribute boolean    isEmpty;
    readonly attribute long       exceptCode;
    readonly attribute string     exceptString;
    oneway void                   SetString(in string str);
    oneway void                   SetExcept(in long code, in string str);
}
```

```

oneway void          Clear();
INotify             New();
};

```

INotify is declared as an interface in the first line of the file. A description of the interface is provided in subsequent lines, between the curly braces. This interface consists of four *attributes* followed by four *operations*. Attributes correspond conceptually to state variables for the object. For example, the first attribute listed, *str*, is declared to be of type *string*. Thus each INotify object has a string of text called *str* that can be queried. The leading *readonly* keyword in the declaration of *str* indicates that its value cannot be directly modified. Without this keyword, the value of the attribute could be set as well as queried. Subsequent attribute declarations use some of the other basic types available in IDL. These include *long*, *short*, *boolean*, and *double*. New types can also be defined as structures, arrays, and sequences. Further, interfaces themselves represent types. Thus, an attribute might be of type *INotify*, for example.

The first of the operations declared in INotify is *SetString()*. The declaration indicates that this accepts an *in* argument of type *string*. The *InOut* mode of parameters in IDL operations must be one of *in*, *out*, or *inout*. The return value from *SetString* is declared to be of type *void*, but operations can in general return values of any type. The *oneway* keyword in the declaration for *SetString* indicates that use of the operation is non-blocking. Normally, invoking an operation causes the caller to wait (block) until the operation is completed. Declaring an operation *oneway* causes control to return immediately to the caller while the operation is performed asynchronously. *Oneway* may not return values.

The next two operations are fairly self-descriptive, declaring return types and required parameters. The *New()* operation, however, merits some comment. This operation creates a new instance of an INotify object and returns an object reference of type INotify as its value. One may then use this new object reference to call operations: *SetString()*, for example. This ability to use object references as types for calling arguments or for returned values is a very powerful feature of CORBA.

A typical use of the INotify interface is to register an event description in the data member *str* by means of the *SetString()* operation. For example, after a successful event, *str* may be set to "Done." The boolean attribute (*isEmpty*) indicates whether anything has been reported back and by examining this, one can differentiate between no

report and the reporting of an empty string. There are also attributes in the notification object to hold exception information (a string and a long) in case some error needs to be reported. The `SetExcept()` operation is used to set these values.

Note, however, that the IDL describes only an interface and that an implementation of `INotify` is free to do whatever it wants with its notifications. While the protocol requires that strings reported to the interface with `SetString()` be made available for subsequent queries via the `str` attribute, it may also do other things when `SetString()` is called. These strings sent to the interface may be written into a log file, displayed in a dialog box, or written into a Web page, for example. We have, in fact, used `INotify` implementations that did each of these.

The OMG IDL allows the developer to define *exceptions*, and any attribute or operation may raise an exception. There are also CORBA-defined exceptions as part of the standard. We make extensive use of exception handling in our software, but have omitted that feature in our IDL in this paper for simplicity.

2. INTERFACES FOR MANUFACTURING DEVICES

We have used OMG IDL to specify standard software interfaces to the various manufacturing devices in the SAMT. A goal of this effort is illustrated in Figure 5. At the bottom of the figure are the various machine tools. On the PC associated with each machine tool, we implement controlling software which provides the OMG IDL interface: by manipulating the software interface, a client program can control the corresponding machine. With this, then, we can write client software illustrated in Figure 5 as the "Cell Management Software Components" which controls the manufacturing activities in the SAMT.

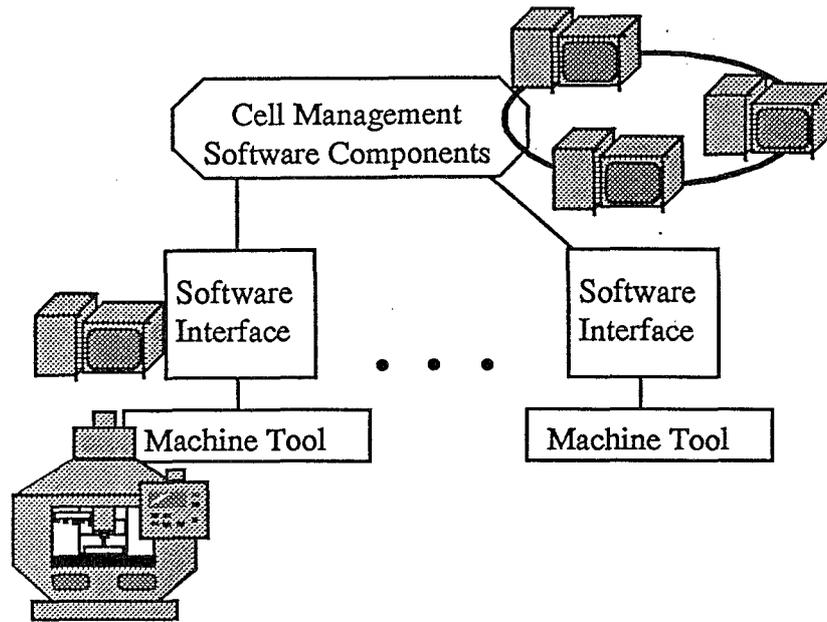


Figure 5. Cell Management Software.

Thus, each of the physical manufacturing objects in the agile manufacturing cell is controlled by a corresponding CORBA software object. In spite of the apparent differences among the various devices (lathe, robot, storage table, *etc.*), these objects all support the same software interface: an OMG IDL interface called `IDevice`. This section presents a description of our `IDevice` interface. For brevity and for presentation clarity, the IDL shown will be abbreviated. The full details can be found at <http://primetime.ca.sandia.gov/~raw/cell/index.html>

2.1 THE IDEVICE INTERFACE

`IDevice` interface presented below inherits from several interfaces and provides two functions, `GetProgDB()` and `GetConsole()`.

```
interface IDevice : IBaseDev, IAllocDev, IRunDev, IMovePart
{
    // Locate the program database for the device.
    IProgDB    GetProgDB();
    // Locate the operator console for the device.
    IConsole   GetConsole();
};
```

The first declaration line for interface `IDevice` illustrates the inheritance feature of OMG IDL. The declaration of `IDevice` as an interface is followed by a colon, then a list of other interface names, starting with `IBaseDev`. This syntax indicates that `IDevice` *inherits* all of the attributes and operations defined in the listed interfaces in addition to those explicitly listed in the `IDevice` body. `IDevice` therefore inherits operations and attributes from the following other interfaces:

- `IBaseDev` — Naming and operational status for the machine.
- `IAllocDev` — Controlling access to the machine.
- `IRunDev` — Running processing activities on the machine (*i.e.*, machining a part).
- `IMovePart` — Transferring material into and out of a machine.

The IDL body for `IDevice` indicates that the `GetProgDB()` operation returns a value of type `IProgDB`. This type corresponds to another IDL interface giving access to a database of manufacturing numerical control (NC) programs. Thus, `GetProgDB()` returns as its value an object reference which can be used to access the database of NC programs available for the device. Similarly, the `GetConsole()` function returns a reference to an `IConsole` object. This object can be used to access the operator's console for the device, enabling communication with a human operator. A brief description of the various base interfaces of `IDevice` is presented below.

2.2 THE IBASEDEV INTERFACE

The `IBaseDev` interface has attributes that give the "Class" of the device (*i.e.*, lathe, mill, *etc.*), a unique ID for the machine (*e.g.*, "Fanuc001"), and a version string for the software object.

```
interface IBaseDev
{
    // Attributes giving Class, unique ID and software version of this machine.
    readonly attribute string    DeviceClass;
    readonly attribute string    DeviceID;
    readonly attribute string    VersionString;

    // Status of the machine.
    readonly attribute string    Status;

    // Advise the provided interface of all changes to the machine state.
    // Return an "advise id". This id is used later to cancel an advise.
}
```

```

long      Advise(in INotify hnotify);
// Cancel advise for machine state.
boolean   UnAdvise(in long adviseID);
};

```

Also provided is a readonly string attribute called `Status`. This indicates whether the device is in an error state, running, or idle, *etc.* Thus the state of the machine can be monitored by polling the `Status` attribute of the device. The behavior of this attribute illustrates the fact that a readonly attribute is not necessarily constant: the value of the status attribute certainly changes over time, it is not, however, directly modifiable by client programs.

Another way to monitor the status of a device is through the `Advise()` operation. As indicated by the IDL, this operation accepts as an input argument a reference to an object of type `INotify` (described previously). A call to `Advise()` requests that all subsequent changes in the value of the `Status` attribute be posted to the `INotify` object with the `SetString()` operation. Thus, the calling program is notified of all status changes for the device. The implementation of the `SetString()` operation could, for example, write an entry into a log file, update a label in a dialog displaying the status, or modify the contents of an HTML page.

2.3 THE IRUNDEV INTERFACE

The `IRunDev` interface, another of the base interfaces of `IDevice`, provides operations for controlling the processing activities on a device.

```

interface IRunDev
{
// Pause, resume, or abort the current activity.
boolean   Pause();
boolean   Resume();
boolean   Abort();

... Other operations ...

// Run provided program. Return results as a string.
string    RunThisProgram(in string jobName, in string pgm);
// Start provided program. Get results later from the "results" attribute.
boolean   StartThisProgram (in string jobName, in string pgm,
                             in INotify hnotify);
};

```

This interface presents a number of operations for controlling processing operations in the device. In addition to a pause, resume, and abort functionality, several IDL operations are available to execute numerical control (NC) programs on a device. `RunThisProgram()` is such an operation, accepting two input string parameters, and returning a string as its value. The first input string is a name to assign to the job. This can be queried by other clients as the `JobName` attribute (not shown). The second input string parameter to `RunThisProgram()` is the text of the NC program to be downloaded into the machine and executed. It is the obligation of the `IDevice` implementation to do whatever is necessary to achieve this. In Section 2.5, we discuss how manual and automated tasks can both be implemented by `RunThisProgram()` in `IDevice`.

When the requested processing activity is completed, `RunThisProgram()` returns a string as its value. Not a status value as one might expect, the return value instead is the information output of the activity. For example, if the downloaded program performs a machining operation, then the output of the operation is the physical part machined, and the return value is a null string. However, the downloaded program may be one that causes a lathe, for example, to position a touch probe and make measurements of the part to test conformance to required tolerances while the part is still in the machine [6]. In this case, the principal output of the step is the measurement data, and the returned string value holds the probe data. This, then, is our mechanism for ensuring the easy extraction of process information out of manufacturing devices and into the broader information processing environment: the act of requesting an operation producing such data yields that data as the return value of the request.

A call to `RunThisProgram()` returns only when the operation is complete, but often an asynchronous request is more convenient so that the caller is not blocked during processing. This is provided in the `IRunDev` interface with the `StartThisProgram()` operation. This operation is very similar to `RunThisProgram()`, its first two string arguments being the same. However, an extra input argument of type `INotify` is required for `StartThisProgram()`. Instead of returning the string output as the return value of the operation, `StartThisProgram()` returns a boolean success/fail value immediately, indicating whether the operation was successfully initiated. The string output value that would have been returned by `RunThisProgram()` is instead posted to the provided `INotify` object when the operation is completed. In practice, we use this asynchronous mode of processing almost exclusively, although the synchronous

operation is occasionally convenient. Other operations in the IRunDev interface (not detailed here) provide for repeatedly running the same program, and for running programs by name, utilizing the program database associated with the machine.

2.4 THE IMOVEPART INTERFACE

The IMovePart interface provides for material movement between devices. An excerpt of this interface follows:

```
interface IMovePart
{
    // Partner in the material exchange.
    boolean    SetPartner(in IMovePart dev);

    // Enter/leave the "Access" state.
    boolean    Access (in boolean fAccess);
    // Enter/leave the "Hold" state.
    boolean    Hold (in boolean fHold);

    // Exchange the part.
    boolean    TakeFromPartner();
    boolean    GiveToPartner();
    ... Other operations and attributes ...
};
```

The interface is organized around three concepts:

- *partner* - Material movement is accomplished by a pair of devices which move material from one to the other. Each device object in this pair knows (*i.e.* has an object reference to) the other as its "partner" in the exchange.
- *access* - For processing devices, this indicates whether the material is accessible. For a machine with a door, for example, in the access state the door is open, while in the unaccess state, the door is closed. For a robot device, access refers to whether the robot is currently reaching into its partner's space.
- *hold* - Whether the device is currently gripping the part or material.

The first three methods in this interface provide operations for setting the partner in the exchange, and for manipulating the Access and Hold states of the device. To illustrate the utility of these operations, consider an example in which a robot arm holds a part that is to be put into a currently empty lathe. The sequence of steps required to accomplish this is presented in Figure 6. This illustrates how some controlling program, using the operations provided in the IMovePart interface, can effect material movement between devices.

Lathe	Robot	Comment
UnAccess, Unhold	UnAccess, Hold	Initial state: Robot holding part to be put into the currently empty, closed lathe.
Access		Open the lathe door.
	Access	Reach into lathe.
Hold		Both lathe and robot grip part
	Unhold	Robot releases part.
	UnAccess	Robot arm moves out.
UnAccess		Lathe door closes.
		Transfer accomplished.

Figure 6. Sequence of Operations to Transfer a Part from a Robot into a Lathe.

Although the preceding is certainly an effective transfer method, a more convenient means of material movement is provided with the operations `TakeFromPartner()` and `GiveToPartner()`. Using one of these operations, the entire sequence of access/hold operations is accomplished via direct device-to-device communications. Reconsider now the scenario in Figure 6. Since the robot device, for example, has an object reference to its partner in the exchange (set in a previous call to the robot's `SetPartner` operation), the robot object can manipulate the access and hold state of the lathe directly. Thus a call to the `GiveToPartner()` operation on the robot results in its assuming responsibility for the transfer, invoking operations on the Lathe interface as needed, and awaiting its replies. This kind of peer-to-peer interaction is very natural and easy to implement with distributed objects, and is a real strength of the CORBA technology.

2.5 AUTOMATED AND MANUAL OPERATIONS IN IDEVICE

In many cases, the operation of machines in the manufacturing cell is not completely automated. This may be because of limitations in the machine controller, or just because we still rely on the expertise of human machinists in various manufacturing processes. We want our software architecture to support automation where it makes sense, but the removal of humans from the manufacturing process is *not* a design goal of our system.

The architecture in place for the SAMT cell control supports both automated and manual activities. Consider, for example, the IDL operation `RunNamedProgram()` (in interface `IRunDev`). This operation accepts as an input string the name of the NC program to run in the manufacturing device, and it is the obligation of the `IDevice` object implementation to do whatever is necessary to carry out the requested machining operation. In the fully automated case, the software can carry out the task by itself. It looks up the provided name in the program database (using the `IProgDB`

interface) associated with the machine, downloads the resulting NC code into the machine, and runs the code on the machine. Some of the various techniques used to accomplish these tasks are described in Section 3 on IDevice implementations.

In the case that the machine (for whatever reason) does not support automated operation, we still want it to be a part of the information flow in the cell. Thus, even in this case we provide an IDevice object for the machine which implements, for example, the RunNamedProgram() operation. The implementation of this operation is obliged to do whatever is necessary to carry out the task. In this case, the implementation uses an IConsole interface (not presented here) to perform the operation. The IConsole interface provides operations needed to carry on a dialog with a human operator. The IDevice implementation uses these operations to request that the operator run the named NC program on the machine, for example.

3. IDEVICE IMPLEMENTATIONS: CDEVICE

One view of our implementations of IDevice objects is given in Figure 7. The "plug-in" jack at the top represents the IDevice interface itself. This is the network-visible interface that each manufacturing device in the cell is required to implement.

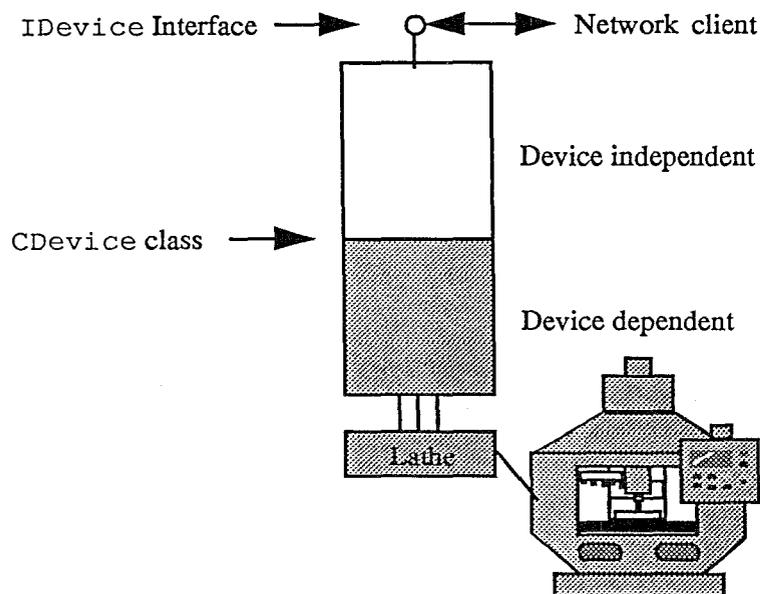


Figure 7. Implementation Layers for IDevice.

Below this is a largely device-independent layer that is common to all of our IDevice implementations. While dealing with issues like presenting the CORBA interface, threads, access-control, version strings, *etc.*, the functionality of this layer does not vary greatly for different manufacturing devices. However, the IDevice object is ultimately required to access the hardware level of a device, for example, to open a chuck or execute a block of NC code. The mechanisms for accomplishing this do vary greatly and the functionality required of this machine-dependent layer is captured in our standardized CDevice C++ class.

Note that IDevice implementations are not required to conform to this CDevice standard. Only IDevice is required by the framework. Utilization of CDevice is an implementation convenience: conforming to this CDevice standard permits reuse of the device-independent layer of our IDevice implementations, making it easier and faster to integrate new manufacturing devices into our software environment.

As a C++ class, CDevice abstracts the functionality of any manufacturing resource: mill, lathe, robot, storage system, *etc.* CDevice class, never instantiated, is used as an abstract base class from which a local and more specialized class is derived. Such a machine-specific derived class then serves as the local implementation class of CDevice. For example, a lathe might have its specific functionality implemented within a CLathe class which in turn derives from CDevice class.

Some member functions for CDevice, and hence its derived classes, have already been mentioned. A complete listing follows:

```
virtual BOOL    Init(...);                virtual char *  GetVersionString(...);
virtual char *  RunThisProg(...);          virtual char *  RunNamedProg(...);
virtual BOOL    Pause(...);               virtual BOOL    Resume(...);
virtual BOOL    SoftAbort(...);           virtual BOOL    EmergencyAbort(...);
virtual BOOL    Access(...);              virtual BOOL    UnAccess(...);
virtual BOOL    Hold(...);                virtual BOOL    UnHold(...);
```

The implementations classes derived from CDevice vary considerably over the shop floor. A brief description is offered to illustrate the variety of implementation methods supported by the CDevice class.

Part loading and automated transport operations depend upon a Stäubli RX 170 robot. The CDevice implementation code sends string commands over a serial line to the Stäubli controller which runs disk-stored programs to execute specific functions. One of the machine tools serviced by this robot is a Baron 25 LeBlond lathe driven by a Fanuc 11T

controller. The lathe and its controller have been modified to achieve fully automated operation including the loading and unloading of parts, the loading of NC code into the controller memory, and the retrieval of measurement data from a turret-mounted touch probe. Implementation code for the lathe and its controller actuates relay cards, digital I/O cards, and a serial port for the transfer of NC code in and measurement data out.

A coordinate measuring machine (Helmel MicroStar Model 430-201) is also serviced by the robot. Implementation code for this device sends command strings over a serial line to run specific programs stored by the controller. Our gas tungsten arc welder (Amet Advent-TPS) is not serviced by the robot. It is manually loaded and unloaded, and a human operator is always present during welding for safety reasons. However, the welding controller has TCP/IP socket communication with a CORBA server object which sends weld schedules to the controller, and receives weld process data in return. A similar scheme services the open architecture controller [14] of the Haas 4-axis mill. Command strings and NC code are sent as socket packets, and data is returned in the same way. Finally, a Cincinnati Milacron 4-axis horizontal milling machine is treated as a totally self-contained unit. It is manually operated and currently sends no data back. But a near-by rack-mount PC furnishes the operator with an interactive screen to both present and receive information via the IConsole object. In this way the machinist has potential access to a wide variety of information sources that relate to the part being machined.

In all of the above cases, the IDevice interface and CDevice class encapsulate the implementation details while allowing for a common network interface to the machine tool. Hence, clients can access manufacturing devices in the same manner; this plug-and-play feature allows for easy integration with other software components and user interfaces.

4. INTEGRATION OF DEVICES IN A MANUFACTURING CELL

As a research facility, SAMT has seen continual evolutionary change over a year and one-half, both in terms of machine tools and their controlling software. IDevice objects were among the first software modules created for this project. Our experience has been that the IDevice design and implementation is sufficiently flexible and adaptive to accommodate both the addition of new machine tools and the evolution of more sophisticated cell management software.

4.1 CELL CONTROLLER GRAPHICAL USER INTERFACE

In the first phase of development of this system, IDevice implementations were developed for all of the devices in the SAMT at that time: a lathe, a coordinate measuring machine, a Stäubli robot arm, and a cleaning station (as well as for several storage tables). Thus we had plug-and-play objects ready to plug into something that could direct them. At that first milestone, these components were controlled directly by a relatively simple client program presenting a graphical user interface (GUI) as illustrated in Figure 8.

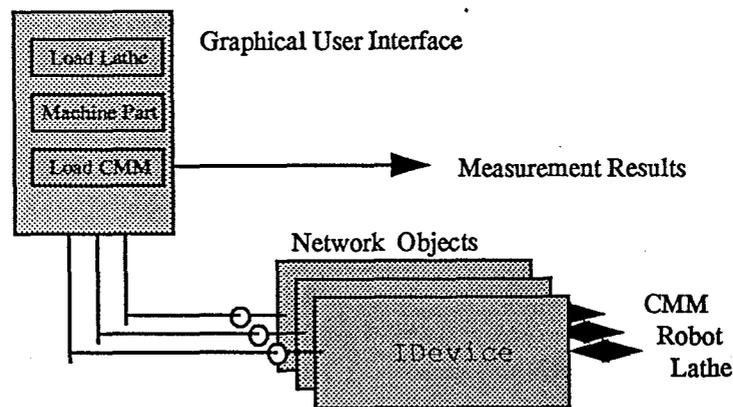


Figure 8. IDevice Objects Plugged into a GUI.

The client GUI was developed using an extension of the Tcl/Tk [9] programming language. The extension [1] to Tcl allows the use of distributed CORBA objects. One of the great strengths of Tcl/Tk is the ease with which X-Windows based user interfaces can be assembled, and this CORBA-based extension to Tcl was a very useful tool.

The client GUI presented a simple collection of buttons with labels like "Load Lathe", "Machine Part", "Load CMM", and "Measure Part". In response to the "Load Lathe" button, for example, the client program invoked operations in the various IDevice objects to pick up a raw stock part from the storage table and put it into the lathe. For "Machine Part", the lathe was instructed to cut the blank into the desired shape, and so on. When "Measure Part" was selected, the coordinate measuring machine probed the part, and sent back an inspection report which was presented to the user. This could be saved into a file as desired.

Thus, in milestone 1 of our project, these plug-and-play IDevice objects were plugged directly into a graphical user interface, that permitted operation of the cell by pushing buttons sequentially. In the next project milestone, these same IDevice objects were plugged instead into a Cell Sequencer component, itself a CORBA object with its own IDL.

4.2 CELL SEQUENCER

In the next phase of development of the manufacturing cell software, we developed a cell sequencer, or task sequencer. As a CORBA object itself, the cell sequencer is also a client to the IDevice objects. This configuration can be seen in Figure 9.

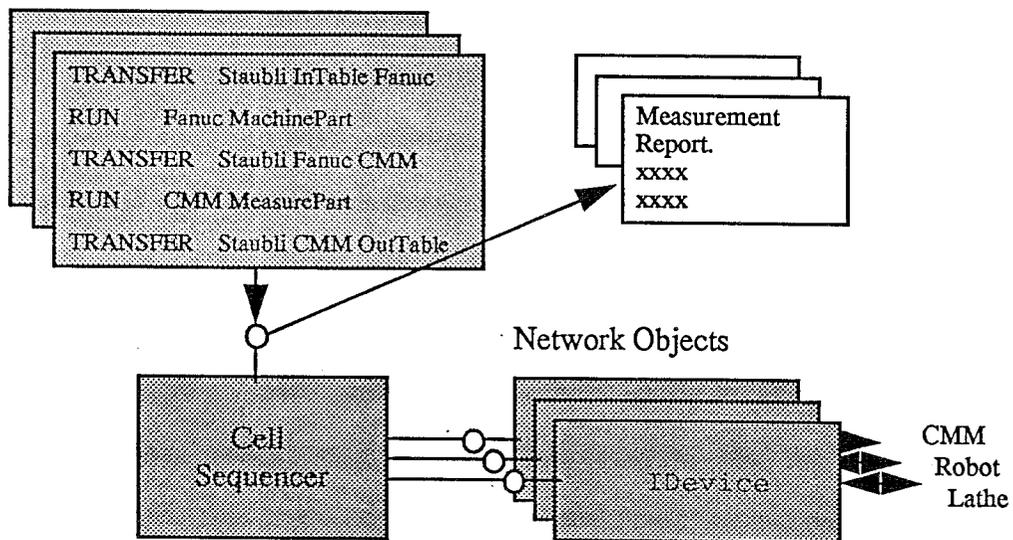


Figure 9. Cell Sequencer Manipulating Several IDevice Objects.

The cell sequencer dynamically attaches to devices, hence, there is no need to re-compile when a new machine tool comes on-line or a machine tool disappears. The cell sequencer accepts jobs, dispatches tasks in the cell, prevents deadlock situations, and guards against starvation of any single job. An abbreviated IDL for the sequencer, ICellSeq, is as follows:

```
interface ICellSeq
{
    readonly attribute string CellName;
    long      AddJob (in string JobName, in string JobText, in string PartName,
                    in INotify WhenDone);
    void      Pause (in long JobID);
    void      Resume (in long JobID);
    void      Abort (in long JobID);
}
```

```

boolean    DeleteJob (in long JobID);
long       Advise (in long JobID, in INotify inot);
boolean    UnAdvise(in long AdviseID);

boolean    AddDevice (in string DevName, in IDevice Dev);
boolean    AddRobotDevice (in string DevName, in IDevice Dev);
boolean    RemoveDevice (in string DevName);
IDevice    QueryDevice (in string DevName);

IIterCellSeqJobs IterateJobs (in long JobID);
};

```

The attribute `CellName` contains the name of the manufacturing cell; in the future several cell sequencers could be coordinated by a shop floor scheduler. A new job can be added to the sequencer with the `AddJob()` operation. This operation takes three input strings as arguments: `JobName`, a unique name within the cell; `JobText`, a "script" of high-level instructions to be accomplished in the cell; and `PartName`, which identifies the part to be manufactured. The `AddJob()` operation also takes an `INotify` object reference so that the sequencer's client can be notified of job completion or error conditions encountered. The return value of the `AddJob()` operation is of type `long`, indicating the assigned `JobID` given by the sequencer; this `JobID` can then be used to `Pause()`, `Abort()`, `Resume()`, or `Delete()` a job in the sequencer, even while the task dispatcher is operating. The task sequencer parses the `JobText`, written in a simple scripting language developed for our manufacturing cell. The language supports manufacturing setup operations, material transfer between `IDevice` objects, and program execution at `IDevice` objects. Though the cell sequencer coordinates cell activities, many operations can be accomplished intelligently by the devices. For example, we have mentioned that all material transfers are performed as peer-to-peer object interactions, independent of the supervisory control of the task sequencer.

The `Advise()` operation gives the sequencer an `INotify` object reference to call when the status of a job changes. If a `JobID` of "0" is given to the sequencer in the `Advise()` operation, the client will be notified of *all* status changes for *all* jobs. The return value of the `Advise()` is an `AdviseID`, which can then be used to cancel sequencer notification by calling the `UnAdvise()` operation with the corresponding `AdviseID`.

There are four operations in `ICellSeq` which allow a client to manipulate devices known to the sequencer: `AddDevice()`, `AddRobotDevice()`, `RemoveDevice()`, and `QueryDevice()`. Notice that there are two different

operations to add a device to the sequencer: the `AddRobotDevice()` is necessary to distinguish robot and transport vehicles from all other manufacturing and storage devices known by the sequencer. The sequencer must know if a device is a transport device in order to prevent certain deadlock conditions in the cell. By including operations for dynamically adding and removing devices in a cell sequencer, the sequencer will never have to be recompiled or restarted when a new `IDevice` object is available on the network. In theory, this architecture supports a cell sequencer remotely dispatching jobs to any `IDevice` object.

The final operation `IterateJobs()` in `ICellSeq` returns an iterator, such that a client can use this interface to iterate over jobs in a sequencer. If specific non-zero `JobID` is given to `IterateJobs`, the iterator will be initialized to point to the identified job. The IDL for `IIterCellSeqJobs` is this:

```
interface IIterCellSeqJobs
{
    boolean    Reset();
    long       Next();

    readonly attribute string    JobName;
    readonly attribute string    JobText;
    readonly attribute long      CurrentLineOfJobText;
    readonly attribute string    LastError;
    readonly attribute long      LastErrorCode;
    readonly attribute string    JobStatus;
};
```

By using the `IIterCellSeqJobs` interface, current status information about one or all jobs can be easily acquired.

The current task dispatcher is quite simple, with no scheduling optimization criteria. This task dispatcher will be used by a smart scheduling object (`ICellSched`). This elaborate scheduler will call `ICellSeq` to dispatch jobs once it has optimized on "time", "cost", and "priority" values on jobs. We expect the current interface and implementation of `ICellSeq` to remain as described above. Allowing for this type of growth is a strength of using this distributed object architecture.

4.3 PERSISTENT OBJECTS FOR DEVICE SETUP INSTRUCTIONS AND CELL RESULTS

In the configuration in Figure 9, a manufacturing production script is an input to the AddJob operation in the cell sequencer, while the cell results are an output of this operation, written to an INotify object. Some of the manufacturing steps in the production script will be setup operations, *e.g.*, tooling and fixturing, possibly extracted from a previous process plan or database. The cell results will likely be stored in a database for future analysis or access. We have developed persistent CORBA objects to hold the device setup instructions (IDevSetup) and the cell results (ICellResults). Hence, we pass CORBA object references for setup information and cell results storage to the cell sequencer. These objects are stored in a database, which is located on a networked computer. The IDL for these objects is available on-line at <http://primetime.ca.sandia.gov/~raw/cell/index.html>.

5. CONCLUSIONS AND FUTURE WORK

We have presented a distributed object CORBA framework for management of a manufacturing cell. This architecture is robust, allowing for easy addition, deletion, and updating of manufacturing devices in a plug-and-play manner. Further, this architecture supports not only manufacturing automation, but human integration by providing console interfaces to manufacturing devices. We have shown several different implementations of our CORBA IDevice interface and the corresponding C++ class, CDevice. Our well-defined common interfaces hide implementation details, including the machine controller and the communications to the machine tool, from the cell management software. The IDevice interface also supports automated and manual machine operations and/or material transport. The CDevice class allows us to encapsulate a wide range of machining, storage, or transport devices.

We have defined and implemented a CORBA interface to a cell sequencer. The cell sequencer manipulates IDevice objects and coordinates manufacturing jobs on a shop floor. Our software development and integration strategy, described in detail in Section 4, allow for the incorporation of more advanced cell management components in the future. CORBA enhances the system integration because it is an industry-standard for interoperable, distributed objects across heterogeneous hardware and software platforms. In time, as commercial software vendors provide CORBA interfaces to various software components, it will be easy to integrate them with our developed manufacturing software. The resulting architecture is scaleable across a wide-area enterprise.

Our future cell software development includes adding support for a more *information-intensive* environment on the shop floor. Intelligent cell management software components must have access to design and planning information in order to make well-informed decisions about scheduling jobs and managing manufacturing devices. At the same time, design, planning, and fabrication information should be available to human operators at machines. To do this we propose implementing an *electronic traveler* as a collection of persistent CORBA objects that interface to databases and product data management (PDM) systems at Sandia. A traveler is the collection of routing forms, part drawings, production scripts and other fabrication information that "travels" with the part as it is being fabricated. This electronic traveler will replace the paper travelers of the past and allow on-line access by both designers and manufacturers, possibly at remote sites. We are also integrating our manufacturing cell with a part inventory system for tooling and fixtures, and a state-of-the-art process planning environment being developed in the SAMT.

Finally, we are currently evaluating the CORBA IDL manufacturing framework defined by Sematech [5]. A key concern we have is the degree to which Sematech architecture supports information-driven manufacturing with human integration.

6. ACKNOWLEDGMENTS

The authors express gratitude to a number of people at Sandia National Laboratories. Jim Costa is a program manager for this work. Hisup Park is the project lead of the SAMT project and the author of the vignettes that became Figures 1 and 3. Robert Hillaire (who provided us with Figure 2), Jon Baldwin, and Tony DeSousa are responsible for various implementations of CDevice.

7. REFERENCES

1. G. Almasi, Suvaiala, A., et. al., "TclDii: A TCL Interface to the Orbix Dynamic Invocation Interface", Concurrent Engineering Research Center, West Virginia University, Morgantown, West Virginia, 1994.
2. Apple Computer, Inc., *OpenDoc Programmer's Guide for the Mac OS*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
3. K. Brockschmidt, *Inside OLE 2*, Second Edition, Microsoft Press, Redmond, Washington, 1995.
4. *The Common Object Request Broker: Architecture and Specification*, OMG Technical Document PTC/96-03-04, Object Management Group, Framingham, Massachusetts, July 1995.
5. "Computer Integrated Manufacturing (CIM) Application Framework", Specification 1.1, Sematech Technology Transfer 93061697D-ENG, August 1994.
6. A. J. Hazelton, "On-Machine Acceptance of Machined Components", *Proceedings of the 10th Annual Meeting of the American Society of Precision Engineering*, Austin, Texas, October 1995.

7. A. W. T. Jones, and McLean, C. R., "A Cell Control System for the AMRF", *Proceedings of the 1984 ASME International Computers in Engineering Conference*, August 1984.
8. R. Orfali, Harkey, D., and Edwards, J., *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, Inc., New York, New York, 1996.
9. Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
10. R. Nagel, and Dove, R., editors, *21st Century Manufacturing Enterprise Strategy, An Industry-Led View*, Iacocca Institute, Lehigh University, Bethlehem, Pennsylvania, 1991.
11. H. Park, "Sandia Agile Manufacturing Testbed", Sandia National Laboratories, Livermore, California, January 1996.
12. M. K. Senehi, Barkmeyer, E., et. al., "Manufacturing Systems Integration Initial Architecture Document", NISTIR 91-4682, National Institute of Standards and Technology, Gaithersburg, Maryland, September 1991.
13. M. J. Wozny, and Regli, W. C., guest editors, "Computer Science in Manufacturing", *Communications of the ACM*, Vol. 39, No. 2, February 1996.
14. P. K. Wright, "Principles of Open-Architecture Manufacturing", Engineering Systems Research Center, ESRC 94-26, University of California at Berkeley, October 1994.

UNLIMITED RELEASE

INITIAL DISTRIBUTION:

0521 G. L. Laughlin, 1567
0342 K. W. Mahin, 1807
9405 D. Lindner, 1809
0505 D. L. Eilers, 2304
0661 G. E. Rivord, 4012
0863 M. A. Tebo, 4012
0807 R. Detry, 4918
9001 T. O. Hunter, 8000
 Attn: J. B. Wright, 2200
 E. E. Ives, 5200
 M. E. John, 8100
 L. A. West, 8200
 W. J. McLean, 8300
 R. C. Wayne, 8400
 P. N. Smith, 8500
 T. M. Dyer, 8700
 P. E. Brewer, 8800
9214 E. Friedman-Hill, 8117
9420 L. A. West, 8200
9405 R. E. Stoltz, 8202
9430 L. N. Tallerico, 8204
9405 M. Rogers, 8220
9133 J. M. Baldwin, 8220
9405 T. De Sousa, 8220
9405 B. V. Hess, 8220
9405 R. G. Hillaire, 8220
9101 P. A. Klevgard, 8220 (3)
9405 S. Leonard, 8220
9405 S. Marburger, 8220
9405 R. Mariano, 8220
9405 H. Park, 8220
9405 J. Schwegel, 8220
9405 J. R. Smith, 8220
9405 T. R. Walker, 8220
9430 A. J. West, 8240
9408 J. O'Connor, 8414
9003 D. L. Crawford, 8900
9011 R. E. Palmer, 8901
9011 P. W. Dean, 8910
9011 W. T. Strayer, 8910
9012 J. E. Costa, 8920 (2)
0806 R. L. Davis, 8920
9011 J. A. Friesen, 8920
9012 J. N. Jortner, 8920
9012 C. M. Pancerella, 8920 (6)
9012 R. A. Whiteside, 8920 (3)
1003 R. W. Harrigan, 9602
1010 M. E. Olson, 9622
1004 M. J. Griesmeyer, 9661
1006 P. Garcia, 9671

0863 T. M. Stephens, 14307
0863 N. A. Lapetina, 14309
0320A K. Thomas, KCP (2)
0320A G. Brace, KCP (2)
9021 Technical Communications Department, 8815, for OSTI (10)
9021 Technical Communications Department, 8815/Technical Library, MS 0899, 13414
0899 Technical Library, 13414 (4)
9018 Central Technical Files, 8940 (3)