Building an Object-Oriented Environment for Distributed Manufacturing Software

Martin C. Sturzenbecker

Manufacturing Research Department
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Integration of distributed systems for manufacturing automation is a difficult task which would be facilitated by a standard object-oriented language which can bridge heterogeneous processor architectures and operating systems. In this paper we discuss the characteristics of a programming environment for distributed manufacturing software based on an open workcell architecture. We consider C++ as a base language for this environment and describe extensions to the language which would make it more suitable for distributed systems integration. The main problem stems from insistence on static (compile-time) binding and type-checking and the necessity to introduce dynamic binding for convenient encapsulation of concepts such as remote procedure call.

Introduction

Software development for complex manufacturing tools has been dictated by the closed workcell architectures of the past. The result is slow, expensive, inflexible software systems. Software costs constitute as much as 90% of the cost of engineering an advanced workcell application. [3] As progress is made towards more open architectures, development environments which can capitalize on the advantages of these architectures are needed for the rapid design and implementation of flexible manufacturing software.

Recent projects at IBM Manufacturing Research have attempted to address this need.

One such project was the “fourth generation” manufacturing programming language AML/X (A Manufacturing Language/eXtended). [9] The goal of AML/X was to promote better integration between product-design (CAD) systems and manufacturing systems; in other words, to be a programming language for CIM. Other projects addressed the need for programming environments that could cope with distributed real-time systems based on heterogeneous processor architectures. [6,8] The aim of these projects was to bring tool and process control more readily into the CIM domain, as well as to accelerate the process and enhance the flexibility of software design for automated manufacturing.

AML/X was a very eclectic language which integrated concepts from a variety of other languages into a remarkably coherent whole. It offered the class-inheritance structuring methods of object-oriented programming, and contained some significant innovations in the area of exception handling. [10] However, AML/X evolved from previous “robot” or “manufacturing” languages which emphasized interactive use and were executed interpretively. Run-time efficiency was not a primary concern in the implementation of AML/X, and no compiler was developed. Consequently, AML/X was not an appropriate vehicle for the development of software which had to satisfy real-time constraints. Although AML/X could be used for simulation [7], a different language (typically C) was used for real-time software.
Programming environments such as the one developed for GPAC [12] were based on this "mixed language" approach. Experience quickly showed that this made the environment difficult to learn and use. There was too little commonality between high-end and low-end program design, and it was hard for a single person to comprehend the system in its entirety.

The problem of dealing with multiple languages and the need for a single systems integration language which can bridge the heterogeneous processor architectures and operating environments in distributed workcells has been observed previously. [15] In an attempt to select a single language which would support all levels of workcell application programming, as well as furnish the "glue" to bind various system components together, we have focused on C++ [4,11], which is developing a considerable following in industry as an object-oriented language for systems programming.

The advantages of C++ as a base language for flexible manufacturing systems have been discussed elsewhere. [2,13] The primary design goal of C++ is to enable an object-oriented approach to software development while at the same time preserving run-time efficiency. The performance implications for every language construct can be easily understood and controlled, which is an important consideration in the development of real-time software. Another advantage of C++ as a candidate language for automated manufacturing is that it can be ported easily and makes few, if any, assumptions about the capability of the run-time system underneath it.

However, implementation of a programming environment for manufacturing should be concerned with more than run-time efficiency and portability. Interactive development, rapid prototyping, and the capability to integrate distributed systems quickly are also important goals. In attempting to achieve these goals, we have come up against the limitations of C++, but we have also observed that with a few extensions to the language these problems can be overcome. These extensions are quite easy to implement and do not severely impact either efficiency or portability, therefore we feel they do not undermine the primary design criteria for C++.

**Integrating Object-Oriented Manufacturing Systems with RPC**

Remote procedure call (RPC) has proven to be an effective concept for integrating distributed automation systems, due in part to its underlying efficiency and easily understood semantics. [1] It combines with the object-oriented approach to yield a specialization through inheritance concept, in which the fundamental method of interaction between nodes of the CIM hierarchy is independent of the function or role of each node. In this approach, communication between nodes of a system assumes one of a few general-purpose forms, such as RPC. Specific protocols and technologies for implementation of this communication are associated with a particular base class of system components. For example, we can build a base class called real-time system, which is an abstraction of a multiprocessor, shared-memory real-time engine devoted to sensor I/O, device control, or real-time analysis functions. Methods of this class provide the fundamental communication protocols for RPC and other structuring concepts. Subclasses of this base class (for example, robot controller or imaging system) build interfaces to specialized systems using the general methods of the base class.

The key to an open workcell architecture is the ability to construct such broadly applicable abstractions without undue loss of efficiency. In a predominantly closed architecture, where each piece of equipment defines a unique set of messages that it can understand, it is difficult to realize abstractions such as this. One approach is based on virtual messages, or a functional mapping from a generic message set to the set of specific messages understood by each machine or other system component. This scheme benefits from the fact that the functional
delineation of the set of basic capabilities corresponds closely to the view which the application engineer has of the equipment. In contrast, notions of real-time system and remote procedure call seem more in the domain of the systems engineer.

However, there are significant drawbacks to the functional definition of a generic interface method. When the mapping from the virtual message space to the real message space of a device is fairly straightforward, all is well. However, this is generally not the case. Often, the advanced capabilities of sophisticated devices can only be accessed by circumventing the generic message set, so environments based on this approach must always provide a readily accessible escape from the basic scheme. Since the layers of translation involved in functional mappings lead to inefficiency, there is ample incentive for a programmer to use the escape mechanism frequently, which detracts from the ultimate objective of the method. There is a burden involved in constructing and maintaining functional mappings for each type of device, and this falls squarely on the systems integrator. Consequently, we feel there is much to be gained from the transition to a truly open workcell architecture based on communication methods such as RPC, both in efficiency and ease of system integration.

RPC Implementation Issues

We now turn to issues of RPC implementation in an object-oriented language which are affected by the language design. Those issues primarily relate to binding (e.g. of procedure references to remote entry points, or remote procedure arguments to local formal parameters), scope (i.e., the lifetime of non-local objects accessible to a remote procedure, and of the procedure activation itself), and type-checking. In a conventional, non-distributed implementation of a language, these issues are typically resolved statically; that is, either by the compiler or the linker, but seldom at run-time. In a distributed system, a dynamic method of resolution is much more desirable. The integrator of a manufacturing system will often find himself linking together equipment and software from a variety of vendors, and cannot count on having source code for every module which may be called remotely. To implement an RPC mechanism elegantly in C++, our main problem is to escape from the built-in bias toward static binding and type-checking. In C++, any new procedure to be accessed on a remote system will require its own interface on the host system, providing a point at which types of actual arguments can be checked and type-safe linkage performed. (This interface is sometimes called a stub, or a wrapper. [14] The host program calls the stub and the run-time system "unwraps" the call by performing the RPC and returning the result, which is of course transparent to the host program.) To code such an interface for every procedure to be called remotely can become quite cumbersome, especially in the prototyping phase. A stub compiler [1,5] can be used to automate the process but does not solve the problem of providing a basic, inheritable RPC method. It would be highly desirable if C++ supported some form of dynamic binding, particularly of procedure (function) argument types to formal parameter types, and of the type of object returned by a procedure. These dynamic binding features were supported by AML/X, which had no type declarations at all for subroutine formal parameters or return values.1

In addition to binding and type-checking issues, problems of scope must be addressed in any implementation of RPC. To do this, we must understand the context in which a remote procedure executes. In a rendezvous model, for instance, the procedure executes within the context of a server process which accepts the

---

1 One could optionally specify a range of acceptable types for a formal parameter in AML/X.
remote call. Thus, to enable an RPC, the code for the server process must be written, the process must be instantiated at run-time, and its identity must be communicated to possible callers.

The notion of process in most real-time systems is very lightweight; that is, the amount of context generated by a process tends to be small. The intent is obviously to minimize the amount of time required both to instantiate a process and to save and restore contexts when control of a processor passes from one process to the next. In such a situation, all processes in the system may run as independent threads within the same global context, and the lifetime of non-local variable references can transcend the lifetime of any particular process. This situation allows RPC to be implemented very simply by creating a new process on a system node for each call to that node. Process creation is done by an RPC-server which is provided as part of the underlying environment. The process has no life apart from the duration of the procedure activation, and there is no need to communicate the identity or state of the process to the caller. Since the scope of non-local references is independent of the process, there is no question or ambiguity introduced by the passing of these references as arguments. C++ has no built-in tasking model, so it can take full advantage of lightweight processes when they are available.

Passing references to objects across system boundaries is usually not meaningful unless all the processing nodes share a common memory. Thus, a call-by-reference argument binding scheme is often inappropriate. In many cases, call-by-reference can be converted into call-by-value with copyback\textsuperscript{2} if a reference exists on a system different from the RPC target. This works well for simple arrays and structures, but quickly becomes cumbersome and inefficient when dealing with objects which contain references to other objects.

One way to handle this problem is to use surrogate objects. A surrogate class is a class which mirrors the operation of another class that is defined and implemented remotely. Every surrogate object contains the handle of an object which may be instantiated on a remote system, and messages to the surrogate are relayed by the surrogate methods to the remote system where the message is processed by the real object. When a system receives the handle of a remote object, it instantiates a surrogate and performs all operations on the surrogate. Again, it would be desirable if all surrogate objects could belong to or inherit from a single surrogate class; otherwise, it would be necessary to create a new surrogate class for each class of objects to be manipulated remotely, and a message interface for the new class which explicitly mirrors the interface of the original class. Hierarchical system architecture also simplifies the problem, since one may assume that object handles are passed only to higher levels. Slaves know only about the objects which they manage, and have a narrower view of the world than the masters to which they report.

Implementation of dynamically-bound RPC and surrogate classes in C++ is hampered by the weak form of overloading which was adopted by the language designers. Overloading, as it is usually understood, involves the semantic extension of a particular syntactic form (e.g., an operator) to cover its use with user-defined types as well as built-in or predefined types. In languages with a starkly simple syntax, such as Smalltalk or Lisp, the concept of an operator as a special syntactic form does not really exist;

\textsuperscript{2} A form of argument passing in which the contents of a remote reference are copied into a location with scope local to the called procedure, and the contents of this local variable are then copied back into the remote location on procedure exit.
arithmetic operators like + or - are just messages or functions and so are readily redefinable for non-scalar or non-arithmetic types of operands. Languages such as C++ maintain the distinction, handed down by their precursors, between predefined and user-defined types, and in some cases have evolved special syntactic forms to denote the sending of messages to objects. Nowhere is this developed more extensively than in C++, where member references can be made directly or through bound pointers.3

The question naturally arises as to whether these syntactic forms can also be overloaded. In C++ the general answer is no: operators . and * cannot be overloaded, and operators -> and -> * can be overloaded but the semantics of an overloading is severely constrained. In other words, C++ regards the built-in mechanisms for member reference or function call (the C++ terminology for object messaging) as sacrosanct. Without the ability to overload these critical mechanisms, there is no way to implement a basic surrogate class in C++.

Extensions to C++ for Dynamic Binding

As indicated previously, we have addressed these problems by making some simple extensions to the language. The first provides a way of defining a multitype, or a type specification which is a finite set of distinct, previously defined types. A type specifier for a multitype is a list of "null declarations" of objects of each type, enclosed in square brackets. An object declared with this type specifier has, at any given time, a type corresponding to one of the types in the list. This type is set dynamically on initialization of or assignment to the object. An example:

```c
[ int, double, complex ] x[100];
```
declares a heterogeneous array, or

```c
[int, double] square([[int, double] x) { return (x*x); }
```
declares a function taking either int or double as an argument and returning either int or double accordingly. The multitype really pays off in this ability to dynamically bind the types of formal parameters or return values to procedures. In our generic RPC implementation, the type which may be returned by an RPC method falls into one of the following categories:

1. A scalar built-in type, such as int or double, if that is the return type of the remote procedure.
2. If the return type of the remote procedure is pointer to a scalar type, or an array of scalar types, the RPC method returns an object of the remote reference class. This object contains the type signature of the scalar type pointed to.
3. If the return type of the remote procedure is user-defined (i.e., a class or structure) or pointer to a user-defined type, the RPC method returns a surrogate object which is an instance of the surrogate class.

Using rules such as these, a wide variety of possible return types can be handled by a single multitype.

Multitypes do involve some hidden overhead, but in most cases this is minimal. Some ambiguities may arise in assignment to a multitype, which must be resolved with appropriate rules. But the concept is easy to understand, relatively easy to implement, and introduces dynamic type binding into the expressive range of the language, so it seems worth the expense.

---

3 A bound pointer is an object which points to an instance variable or method of a particular class, and is therefore "bound" to the structure of objects belonging to that class and can only be used in connection with such objects.
The second extension provides for stronger overloading of member-reference operators and the call operator. To make a member reference operator such as \texttt{.} a "first-class" operator, we must view the expression \texttt{x.y} as sending the message "y" to the object \texttt{x}. This is considered to be a binary operation with \texttt{x} as the first operand and a character string as the second. By allowing overloading of this operator we can replace the built-in member reference mechanisms of C++, which allows a universal surrogate class to be constructed.

It is also difficult in standard C++ to overload the call operator in a way which preserves semantics, since one may not view the argument list as a single entity. The operator may only be overloaded for specific combinations of number and types of actual arguments. In AML/X, the list of arguments passed to a call operator was accessible through the \texttt{actual_args} list object, and the call operator viewed essentially as a binary operation with this list as the second operand. We would not impose this interpretation on C++, for it places too many constraints on the implementation of argument passing; however, we would like to be able to introduce our own mechanisms for argument list building when appropriate. This can be done by defining three special member functions:

1. \texttt{arginit}, called with the number of actual arguments at the beginning of argument processing
2. \texttt{argnext}, called with the \textit{i}th actual argument as parameter after that argument has been evaluated
3. \texttt{argend}, called after all arguments have been processed

Extending call operator overloading allows us to implement RPC in a syntactically transparent way, without requiring the coding of stubs for every remotely callable function.

Again, these extensions could be criticized on the grounds that they make possible extensive amounts of hidden overhead, but no more so than the standard form of operator overloading. The only difficulty lies in accepting the notion that built-in mechanisms for member reference, etc. can be selectively replaced.

Dynamic binding as described here involves use of a run-time data base. For example, a instance of the \texttt{surrogate} class must have the type signature of the remote object that it represents; this signature is actually a pointer into a data base which contains, among other things, the accessible member functions of the type and the types of arguments expected. This data base is built as the system is configured and various programs are downloaded and/or enabled on subordinate nodes. When \textit{slave-to-slave} RPC is required among nodes having a common master, the data base residing on the master can be shared by the cluster. This is most tractable when the cluster is tightly-coupled, having either a shared memory or a high-bandwidth connection to the master node.

Thus, the requirements of a distributed programming environment impose certain constraints on the system architecture. Object-oriented environments in particular seem to have an affinity for hierarchically structured systems. The master node maintains the required global view of the system under construction, and subordinate nodes simply implement the internals of various classes.

Conclusion

We have developed a research prototype of a distributed C++ environment for manufacturing at the IBM T. J. Watson Research Center, and will be applying it to the development of real manufacturing systems in 1991. Interactive system configuration, prototyping, and debugging are supported through interpreted execution of C++ at the user-interface level.

C++ can be the implementation language for a distributed programming environment for automated manufacturing, but the static binding and type-checking bias of the standard language are an impediment in this regard. Two fairly simple
extensions to the language, multitypes and extended overloading, can provide C++ with the dynamic binding capability required to integrate distributed object-oriented systems effectively. As an object-oriented approach to programming manufacturing workcells becomes more popular, the need for open architectures and a standard language for system integration will become more critical. If standard C++ can evolve the necessary features for convenient encapsulation of concepts such as RPC and surrogate classes, it can establish a powerful presence in manufacturing programming environments of the future.

References