Preprint
UCRL-JC- 137092

Java Based Open Architecture Controller

G. Weinert

This article was submitted to
World Automation Conference, Maui, HI, June 11-16, 2000

January 13, 2000

U.S. Department of Energy
Lawrence Livermore National Laboratory

Java Based Open Architecture Controller

George Weinert
Lawrence Livermore National Laboratory, Livermore, California, USA

ABSTRACT

At Lawrence Livermore National Laboratory (LLNL) we have been developing an open architecture machine tool controller. This work has been patterned after the General Motors (GM) led Open Modular Architecture Controller (OMAC) work, where we have been involved since its inception. The OMAC work has centered on creating sets of implementation neutral application programming interfaces (APIs) for machine control software components.

In our work at LLNL, we were among the early adopters of the Java programming language. As an application programming language, it is particularly well suited for component software development. The language contains many features, which along with a well-defined implementation API (such as the OMAC APIs) allows third party binary files to be integrated into a working system. Because of its interpreted nature, Java allows rapid integration testing of components.

However, for real-time systems development, the Java programming language presents many drawbacks. For instance, lack of well defined scheduling semantics and threading behavior can present many unwanted challenges. Also, the interpreted nature of the standard Java Virtual Machine (JVM) presents an immediate performance hit. Various real-time Java vendors are currently addressing some of these drawbacks.

The various pluses and minuses of using the Java programming language and environment, with regard to a component-based controller, will be outlined.

KEYWORDS: Java, Component Software, Control Systems.

INTRODUCTION

Current machine tool controls (a.k.a., CNCs) are embedded, firmware based computer systems. Although many of these allow end users and system integrators to add sensors and actuator, it is very difficult for them to add algorithms that would allow them to integrate their process knowledge into machining and manufacturing processes. Further

it is difficult to get the builders of these controllers to create "specials", customized versions with their customer's software added to them.

Even if this software is added to one manufacturer's current generation of controllers, there is no guarantee that this software will be portable to newer generations of controller hardware, or to another manufacturer's controllers. Without a standard framework to implement these process-specific algorithms, the end user's process knowledge is lost with each change in control system hardware.

Open Modular Architecture Controls (OMAC) Initiative

In 1994 General Motors, Ford Motor Company, and Chrysler Corporation released their requirements for an Open Modular Architecture Controller [1]. This document outlined the need for standardized, modular, component based control system architecture.

The approach outlined in the document involved breaking a controller into several modules (i.e., components) defined by their functionality and by the software interface that they provided.

Work commenced in early 1995 to begin defining precisely how many modules there were, what their functionality and states they encompassed, and specifying well-defined application programming interfaces (APIs) for these modules. In addition this API working group, which I have been an active member in since its inception, was tasked with defining an architectural framework in which these modules would function.

In defining this architecture, one of the overriding requirements was allowing one component to be swapped with another component. In other words allowing one implementation of a module to be replaced with another implementation. This required that the APIs specify how the results of a computation are accessed, but not how the calculation is carried out.

Component Software Requirements and Definition

The OMAC Requirements Document [1] specified that the controller be open, modular, and scaleable. The openness and modularity requirements were addressed by breaking the controller into several replaceable pieces, defining the state behavior for those pieces, and specifying their APIs.

Scalability, which was defined as "enabling easy and efficient reconfiguration to meet specific application needs, from low to high end", has several dimensions to it. One of these is the ability to extend the APIs of the modules (for more demanding, unanticipated needs), while still allowing backwards compatibility with existing components. This was addressed by treating the module APIs as Objected Oriented entities, which had no implementation. Inheritance may be used to extend any given API, and therefore any given module.

A software component, for purposes of OMAC, is required to implement one, or more, well defined API sets, have a well-defined state behavior (which is reflected in the individual API sets), and be easily integrated into a controller or exchanged with another compatible component. Ideally these components could be shipped as binary code, rather than source code, allowing software producers to protect their proprietary knowledge.

OMAC Component Defined

An OMAC Component is an implementation of a given OMAC module API, which may be integrated into an OMAC controller. Any given component must implement at least one of the Module APIs, but may actually implement more. In general, there may be more than one instance of any given component in an OMAC controller. An OMAC controller is an integrated collection of OMAC components.

Since building a controller means integrating many individual components, it is desirable to have some automated assistance in performing this task. In addition to the module APIs, each component must also implement Component APIs. These Component APIs can provide an integration tool with a wealth of information about a component. This information includes the Module APIs implemented, and other the other Module API implementations required. For example a component may implement the Axis Module API, and require a connection to another component implementing the Control Law Module API. A Unified Modeling Language (UML) model of an OMAC Component is shown in Figure 1.

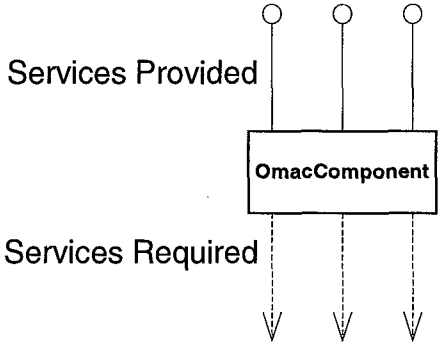

Figure 1. Omac Component

The Component APIs allow a system integrator to pull in the implementations required for his application, and then specify connections between components. These connections may be checked for type and completeness using the information provided by the component, thereby reducing time spent by the integrator and increasing safety. Code may then be generated to produce the system specified by the integrator.

In future we anticipate additional information to be provided by components, which will increase system correctness and reliability. This information could include component characteristics such as timing and memory constraints.

THE JAVA LANGUAGE AND OMAC REQUIREMENTS

The OMAC API specifications were designed to be language independent, and were specified using the Interface Definition Language (IDL). This allowed the APIs to implementation language agnostic.

As stated previously, OMAC goals required that the software components be capable of being distributed as binaries and integrated using automated tools.

In addition to its component features, Java has several other important features as related to control systems. Unlike C or C++, Java includes a threading model and the concept of mutual exclusion at the language level. This allows a software author to safely specify concurrent activities. Java also specifies a rich, standardized class library of utility objects such as hash tables and object vectors.

Java Component Features

The Java language has many features which enable the use of component based software. These include late binding and the ability to distribute objects as compiled class files. Java also defines an "interface", which corresponds to the module APIs. In addition, features added in Java 1.1 to support the Java Beans specification provide ways for tools to "look inside" an object and see what it implements. These features are referred to generically as "introspection".

All code written for our system that requires the use of an OMAC module API is written using Java interfaces. Instead of specifying a particular implementation of an API, code is written referencing the interfaces only. Later, a system integrator will specify which implementation of that interface is to be used.

Late binding allows the Java run-time environment to incrementally load classes and components, and only linking as the need arises. This allows a system to be built and tested piece-wise by an integrator.

The introspection capabilities of Java allow an integration tool to find the interfaces (APIs) that a component implements, the data attributes that may be queried or set, as well as events that may be generated or caught by an object. These abilities are already in widespread use in readily available Java development tools.

All of the above abilities come at no cost to the programmer, and are supported by the Java compiler.

OMAC extensions to Java Component Features

Even though Java provides a rich set of component features, it is not rich enough to support all of the functionality needed by an OMAC component. Specifically, a standard Java component cannot advertise which other modules it needs to operate.

We have extended the objects that we are using with additional interfaces that may be queried to find which other modules are required (see Figure 1). Other interfaces have been added that allow a component to be connected to another component's services. These connections are made using ordinary Java references.

Any additional future information that an integration tool requires would be made available through new extended interfaces.

Java and Real-Time

While Java provides good component facilities, it does not provide the best real-time characteristics. Good real-time characteristics would be defined as predictable response times and execution times. There are three primary areas which affect real-time performance in a Java based system: garbage collection, thread management, and interpreted execution.

Garbage Collection. The primary failing of Java in a real-time environment is Java's extensive use of garbage collection as a means of memory management. Garbage collection can be both a blessing and a curse.

In a component-based system built out of pieces supplied by many third parties, it is difficult (if not impossible) to assign responsibility, in an enforceable way, for the lifetime and memory management of any particular object. In other programming

languages (notably C++), this can lead to memory leaks (un-reclaimed memory) and the reclamation of memory still in use. Java has solved this problem by using garbage collection to automatically manage storage.

Garbage collection can be implemented in many ways [2]. The most common approach is "mark and sweep". While this approach is simple, it has one major drawback: it must freeze all threads in the system and run until it has recovered all inactive memory. This can take anywhere from a few milliseconds to a few seconds. In a system that has many control loops running at millisecond repetition rates, this can be disastrous.

Fortunately, several vendors are starting to produce Java run-time with "real-time" garbage collection characteristics. Although these algorithms are not yet standardized, they allow for incremental, prioritizable, interruptible garbage collection to be performed. This allows a system to retain its real-time characteristics. We anticipate that as Java makes more inroads into the embedded/real-time market that the availability of these product offerings will increase. Additionally, industry lead groups such as the Real-Time Java Experts Group (the J Consortium) are addressing these problems. [3]

Thread Management. Java provides standardized thread semantics, which include creation, prioritization, and mutual-exclusion. However, the actual behavior of different priority threads is not defined (e.g., are threads cooperative or preemptable?). Java also does not provide any definition of the order that threads waiting at the same synchronization block will be freed (e.g., FIFO or priority?).

Once again, various real-time java vendors and the J Consortium are addressing these concerns.

Interpreted Execution. Finally, Java by its nature is made to run on a virtual machine, whose instruction set is defined by a set of byte-codes. Normally, these byte-codes are interpreted by software running on the target machine. This process is anywhere from 10 to 20 times slower than native code execution. While slower execution speeds do not necessarily translate into the violation of real-time constraints, in general an open system will be more useful if it has more free resources available for user code.

To solve this speed limitation, the Java Virtual Machines (JVMs) on most popular platforms use a just-in-time (JIT) compiler, which transforms the byte codes to native code the first time the code is executed. While this speeds up subsequent executions of the code, it does lead to long, unpredictable delays the first time the code is executed.

JIT compilation is not an acceptable solution for real-time systems. Once again, several solutions now exist from various vendors. These include ahead-of-time (AOT) compilers, and compile-on-load solutions. The AOT compilers essentially take Java byte codes and compile them into an executable image in native code. All class loading is pre-resolved before run time. Compile-on-load solutions work similarly to JIT compilers, but instead of waiting for the code to be executed, it is compiled to native as soon as it is loaded. Classes used by a loaded class are immediately loaded and compiled as well. In essence, an entire system will be loaded and compiled before any execution takes place with either AOT or compile-on-load. When compiled to native, Java code runs approximately 10 to 20 times faster, at the expense of either longer preparation or loading times.

SUMMARY

We have implemented an extensible machine tool controller using Java. This controller is based on well-defined OMAC module APIs and component APIs. In addition, we have implemented rudimentary integration tools that take advantage of the component APIs, generating application code and checking for system consistency.

We are currently fielding different test configurations of this controller, and working on various extensions to the original work.

The pros and cons of Java for this work are summarized in Table I.

Table I. Java Pros and Cons for Component-Based Control Systems

Feature	Pros	Cons
Garbage Collection	Avoids memory leaks and use of reclaimed memory	Improper GC algorithm may cause unpredictable system pauses
Component APIs	Many APIs for components already exist and are transparently supported by the compiler and development tools	Insufficient for OMAC use – must be extended
Interfaces	Semantics well defined for module interfaces	None
Thread Management	Semantics for multi-threaded applications are well defined, and standardized across all Java platforms, easing implementation	Not all threading behaviors with regard to priorities and synchronization are fully defined, leading to potentially incompatible implementations
Byte-Code Interpreted Execution	Faster debugging and prototyping. Easer to distribute components as compiled binary code	10 to 20 times slower execution speed compared to native binaries

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES

1. "Requirements of Open, Modular Architecture Controllers for Applications in the Automotive Industry Version 1.1", December 13, 1994. http://www.arcweb.com/omac/docs&nrs/omacv11.htm
2. Richard Jones, Rafael D. Lins "Garbage Collection: Algorithms for Automatic Dynamic Memory Management", John Wiley & Son Ltd., 1996.
3. Real-Time Java Experts Group. http://www.rtj.org/public/