DESIGN OF A UNIFIED CONTROL SYSTEM API

N.Malitsky, R.Casella, K.Lally, S.Peng, J.Smith, BNL, Upton, USA
D.Gurd, LANL, Los Alamos, USA

Abstract

This paper presents the design of a unified control system API.

1 MOTIVATION

The accelerator control client-server environment is built after the ISO reference multi-layer model where each layer provides a service interface to the layer above and protocols for communicating with the corresponding layers in other systems. There is a mismatch between object-oriented application semantics and a low-level control interface. Several accelerator tools address this task by providing uniform generic Application Programming Interfaces (API) to heterogeneous accelerator devices. A generic interface is a very effective approach for integrating open mutable diverse underlying systems. However, its highly adaptive architecture introduces significant overhead for application and system developers requiring them to explicitly support and employ proprietary API-specific meta-facilities and communication requests. Also, a generic service does not specify application domain data types and can not act as an object-oriented framework for high-level accelerator applications. The modern industrial technologies address this task by providing the object-oriented approach in developing distributed systems. They not only automate many routine procedures, but also create a basis for integrating accelerator application design patterns and frameworks into the control system environment. The next sections describe how to apply these technologies for developing a unified control system API.

2 SELECTION OF SOFTWARE TECHNOLOGIES

2.1 Programming Language

The choice of programming language is determined by several criteria: (1) support of the modern methodology; (2) the quality and quantity of associated libraries, extensions, and products; and (3) its popularity. At this time, most object-oriented applications are written in C++ or Java. The recent release of the second version of the Java Development Kit (JDK 2) and the specification of the Java 2 Enterprise Edition Platform (J2EE) has shifted the balance between two competitive languages. The Java 2 not only solves the C++ portability problem and implements the latest ANSI/ISO C++ Standard features, but also offers a rich set of integrated technologies and tools that significantly facilitate developing distributed scientific and control applications. For example, a meta-facility takes an essential part of the modern C++ accelerator control system frameworks and the Java reflection mechanism allows software developers to avoid the maintenance of proprietary accelerator-specific meta-languages (such as ADO .rad files or the CDEV Device Definition Language). Also, the previous Java version, JDK 1.1, limited Java developers to the client part of the multi-tier distributed systems. The Enterprise JavaBeans (EJB) specification and Java 2 performance enhancements (e.g. JIT compilers) have opened new possibilities for developing Java-based client-server systems. To compare C++ and Java compilers we run on the Sun Ultra (Solaris 2.7) computer a simple test performing multiplication on the elements of a large array of doubles (see Table 1).

Table 1: Average time of multiplication and copy on the 1000000 elements of an array of doubles.

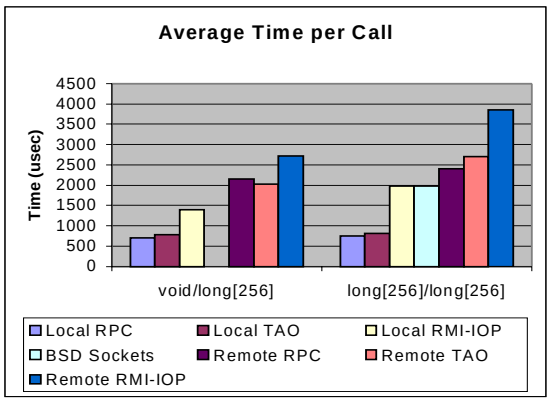
Language	Optimization	Avg. Time (ms)
Sun CC 4.2	-fast	51
egcs-2.91.66	-O3	75
Java		85

The next generation of Java compilers, the HotSpot Performance Engine, aims to provide additional performance enhancements to the Java platform. The HotSpot compiler is based on the adaptive optimization technology that identifies and precompiles the most frequently used methods and statements of long running systems. We plan to benchmark its power on real-size prototypes of accelerator applications.

2.2 Middleware

The accelerator control systems have to rely on the robust and high-performance infrastructure. The efficient communication between distributed components can be achieved with the low level transport mechanisms, such as BSD sockets, Windows NT named pipes, or Sun remote-procedure call (RPC) libraries. However, these mechanisms lack type-safe, portable, reentrant, and extensible interfaces, and require the additional adapters for being embedded into object-oriented applications. To solve these and other problems, the Object Management Group consortium has introduced the Common Object Request Broker Architecture (CORBA), an industrial

standard for developing scalable interoperable object-oriented distributed systems. CORBA not only automates the process of marshaling domain objects into transport layer, but also offers a rich set of supporting services, such as Naming, Event, Transaction, and many others. The recent implementation of the CORBA Internet Inter-ORB Protocol (IIOP) in the Java Remote Method Interface (RMI) brings CORBA technologies to the Java conventional environment. Figure 1 shows the performance comparison of the Sun RMI-IIOP middleware with BSD sockets, Sun RPC, and TAO (C++ high-performance real-time ORB). The remote roundtrip times were measured between Ultra-Enterprise 400 MHz machines running Solaris 2.7 and located on different subnets.



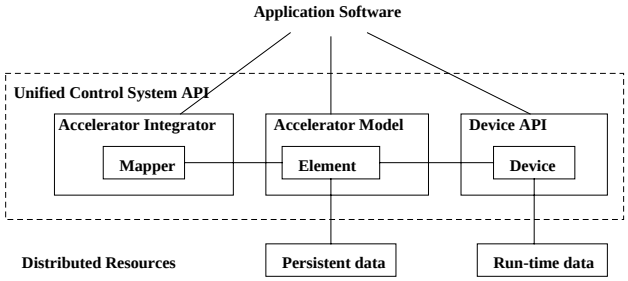
2.3 *Component-Oriented Framework*

The design and implementation of the application domain frameworks involve significant efforts in evaluating, developing, and integrating reusable collaborative components. The modern component-oriented frameworks, such as Sun Enterprise Java Beans (EJB), CORBA Component Model (CCM), and Microsoft COM, facilitate and direct the software development process by providing a standard-based horizontal platform, a factory of specialized vertical domain environments. At this time the EJB architecture is the most advanced, mature, and promising technology. It takes a central part of the Java 2 Platform, Enterprise Edition (J2EE), and the CORBA CCM specification defines the consistent interoperable mechanism for deploying EJB objects in the future CCM environment.

EJB architecture divides all domain objects into two categories: Entity Beans and Session Beans. Entity Bean represents a persistent coarse-grained identified domain object that can be shared among multiple clients. Session Bean can be a service or a conversational session with a particular client. All EJB objects are executed in the EJB container's run-time environment that provides a unified standard interface to the common industrial services, such as lifecycle management, persistence, transactions, and many others. In the next section, we show the relationship between the EJB architecture and accelerator framework patterns.

3 CONTROL SYSTEM API

A control system API unifies the three components: Accelerator Model, Device API, and Accelerator Integrator (see Fig. 2).



3.1 *Accelerator Model*

The description of accelerator structures is a key part of accelerator programs. Accelerator combines many elements of different physical types with heterogeneous attributes, all organized in a nested hierarchical structure. The complexity of this organization prompts a variety of project-specific views and implementations of accelerator models. The diversity of different models prevents the comparison, selection, and integration of reusable algorithms and applications. The recent Accelerator Description eXchange Format (ADXF) proposal aimed to provide a uniform, complete, and extensible approach in the definition of the accelerator state. The concepts described in this proposal are based on the Standard Input Format (SIF), Standard Machine Format (SMF), Standard eXchange Format (SXF) and experience with actual accelerator applications, such as RHIC, SNS, LHC, CESR, FNAL Main Injector, and others.

The original version of the ADXF proposal has been mapped into the Extensible Markup Language (XML), an industrial standard for processing Web documents and application-neutral data. In this paper, we have implemented its object model into two other representations: relational database tables and Java interfaces.

The ADXF database schema is described in Table 2. The *Accelerator Node* table keeps references to all accelerator nodes. *Beamline* is an intermediary linking table that represents the many-to-many relationships between elements and sequences of elements. Node attributes are distributed in the different element buckets. Each bucket has its own table to accommodate a fixed set of element attributes. The *Multipole Bucket* table illustrates the use of embedded virtual arrays in an Oracle table with knl and ktl fields. It is a new feature of Oracle 8i and is supported in SQL 3 and JDBC 2.0 specifications. Each accelerator element may also include insertions, other accelerator nodes (e.g. detector

solenoid). This many-to-many relationship is represented by the *Element Insertions* table.

Table 2: ADXF schema

Table	Columns
Accelerator Node	ID, NODE_TYPE, DESIGN_ID
Beamline	ID, POSITION, NODE_ID
Element Buckets	ID, BUCKET_INDEX, BUCKET_TYPE
Element Insertions	ID, POSITION, NODE_ID
Basic Bucket	BUCKET_INDEX, BUCKET_TYPE
Multipole Bucket	BUCKET_INDEX, KNL, KTL
Other Element Buckets . . .	

The ADXF model has been evaluated also in the Java RMI-IIOP-based distributed environment. Client and server communication stubs for accelerator nodes have been generated by the Sun rmic compiler from the AcceleratorNode remote interface. The interface between application server objects and Oracle is based on the standard Java JDBC technology.

3.2 Accelerator Device API

Accelerator Device represents an identified accelerator physical entity which actual run-time parameters can be accessed, controlled, and monitored by accelerator applications. In most control software packages, the structure of the Accelerator Device is described by the following model:

- Accelerator Device contains a collection of Parameters.
- Parameter has a value and Properties. There are several types of Parameters that are characterized by fixed sets of associated Properties.

This Accelerator Device and the ADXF Element have the same two-level structure: a *dynamic* collection of *statically* defined data sets. This structure facilitates selection and classification of well-defined concrete data types. On the other hand, it provides a consistent extensible mechanism for describing and integrating new element parameters. Also, Accelerator Device and ADXF Element models complement each other and can be connected according to the following formula: *a value of Accelerator Device parameter is an attribute of the corresponding Element.* It means that Accelerator Device may act as a service for comparing or synchronizing the Accelerator Model attributes with actual run-time data.

Employing the CORBA technology significantly facilitates the implementation of the Accelerator Device model in the distributed control environment and provides the following benefits: simultaneous support of generic and object-oriented interfaces to remote accelerator devices and extension of communication data types by objects. On the other hand, the CORBA is a highly adaptable technology and permits the incremental integration of its components with existing low–level control system interfaces, such as EPICS Channel Access and ADO classes.

3.3 Accelerator Integrator

The Accelerator Integrator is a computational engine that simulates diverse beam dynamic processes and transforms accelerator data into accelerator physics modeling abstractions (such as Twiss parameters, Taylor maps, *etc*.) for high-level applications. Accelerator physics (as any other scientific domain) is characterized by a variety of different algorithms and approaches. Usually, a choice of the optimal solution is a difficult tradeoff among many project-specific factors. The UAL framework infrastructure addresses this task by defining a universal mechanism for assembly and reuse of independently developed accelerator algorithms.

The mechanism is based on the Element-Algorithm Probe analysis pattern that introduces three collaborative classes: Element (e.g. Quadrupole), Algorithm (e.g. QuadrupoleTracker), and Probe (e.g. Bunch). This structure is very similar to the EJB architecture and makes straightforward its implementation in the EJB environment. The Accelerator element is an identified persistent object that corresponds to the EJB Entity Bean semantics. The Algorithm does not have a persistent state and can be implemented as a stateless or stateful EJB Session Bean. The Probe is used for data exchange between different Algorithms and fits to a Java serialized object.

The implementation of the Element-Algorithm-Probe analysis pattern is based on the Mutable Class design pattern. A Mutable Class divides a class into two parts, a Type class and Instance class, and delegates an Instance's behavior to Type. The Type object is implemented as a singleton, that serves as a Manager (factory and finder) of Instance objects. For accelerator elements of the same type (e.g. sector magnet) there is a separate manager (e.g. SBendManager) and associated algorithm managers (e.g. SBendTrackerManager) The Type-Manager singletons can be grouped together in the uniform collection and be controlled by the separate object, a Type Registry (finder). One can consider an Algorithm Registry as an alternative approach to a Visitor for open configurable systems. In the EJB environment, Element and Algorithm Managers correspond to EJB Homes that are automatically generated for each element types.

Architectural principles of the Element-Algorithm-Probe framework have been tested with the BullSoft freely available open source implementation of the EJB specifications.

We thank S.Sathe, J.Song, and J.Wei for many useful discussions.