Towards a Common Object Model and API for Accelerator Controls

F. Di Maio[a], J. Meyer[b], A. Götz[b,c]
a) PS Division, CERN, 1211 Geneva 23, Switzerland
b) ESRF, European Synchrotron Radiation Facility, BP220, 38043 Grenoble, France
c) HartRAO, Hartebeesthoek Radio Astronomy Observatory, PO Box 443,
1740 Krugersdorp, South Africa

Abstract

An Object-Oriented Application Programming Interface (OO API) can provide applications with an abstract model of the components of an accelerator. The main question is how to encapsulate different control systems into one single abstract model. The abstract model of an OO API can be described in a formal way via object models in order to clarify the semantic issues, to describe the important concepts (device, attributes...), and to decompose the objects up to the granularity where the model of some objects can be shared between labs. A C++ API (as well as C API) can be derived from the object-model. This paper presents a common object model which is derived from both the current CERN-PS model and the current ESRF model. We describe the technical difficulties we encountered in migrating existing control systems into a shared but usable model. We also aim to increase the universality of the model by taking into account the CDEV library, as well as CORBA. A high-level description of the model will be presented with examples of the derived API.

1. INTRODUCTION

The control systems of CERN-PS and the ESRF have, respectively, been rejuvenated and designed during the early 90s. They are both based on objects in the front-end computers. They also have additional similarities due to the fact that they are based on similar technologies and that there has been a regular collaboration between a number of European institutes (SoftCol).

In this framework and with the objective of enhancing software sharing, we asked ourselves the following questions: (a) can we agree on a common API (Application Programming Interface) ? and (b) can we agree on a common model for objects? These questions are particularly pertinent in the context of the current and future OO (object-oriented) technologies, like designing a C++ API or using CORBA technology. CDEV, another candidate for an OO API, can potentially close the gap between object based control systems and the successful EPICS collaboration; it has been included in our analysis here.

In this paper a “draft” model is described, focusing on the technical difficulties identified in this process. It especially focuses on the feasibility of a common model and API. Comments on the sharing process are given by both institutes. Common conclusions are presented at the end.

For more information on the different technologies discussed in this paper, the reader is referred to the following documents: CERN PS equipment access is described in [1] and [2], ESRF device access and device server model are described in [3] and [4], CDEV is described in [5] and two different implementations of CORBA are described in [6] and [7].

2. THE BASIC OBJECT MODEL

The basic object model describes the device object and its associated entities: device class, device attributes and device commands. Shared generic software should be based on the features specified in this basic model. By generic software, we mean software which implements services for any piece of equipment, whatever its specific type is.

One object can have many representations. The same device object can, for instance, have three representations: (a) a description in a file or data-base, (b) one "concrete" instance in a front-end and (c) many "images" objects in the consoles. All the representations can use the same object model, although the implementations can differ.

2.1 Device

A device object implements the services provided by the control system to an accelerator equipment, it has a clearly defined functionality, it is unique and persistent, in the sense that it exists only once in the control system and that its state is not maintained by application programs. A power-supply and a beam position monitor are two examples.

A device has a name which is an identifier unique inside the installation. The devices are classified and have an interface composed of attributes and commands.

2.2 Device Class

Every device belongs to a class which describes equipment with the same functionality. Device objects belonging to the same class have the same interface. Every device class has a unique name inside the institute scope or inside a wider scope (e.g. HEP institutes scope), if some device classes are standardized.

Device classes are organized with parent/child relationships (or inheritance tree); defining a class as a child of another one makes it "inherit" its interface (attributes and commands). Multiple inheritance is necessary, especially for adding generic services to a device class, like inheriting the alarm interface from a "deviceAlarm" class in addition to deriving from another device class.

The device object must implement access functions to the class name and to the description of all attributes and commands. It must also implement the "isA" operation which specifies if a device is an instance of or is derived from a given device class. (cf. IBM's CORBA's "Object" implements the "GetClassName", "GetClass", "IsA" and "IsInstanceOf" functions). In the CERN-PS implementations, the device classes ("equipment modules") are concrete entities which can provide services (i.e. objects).

The device class is not necessarily the C++ class of a device object. For example, the C++ class of a power supply device will be "Pow" on a front-end, while the same object can be an instance of a generic "Device" class on the consoles.

2.3 Device Attributes

The device attributes are the "instance variables" of a device object. There are a variety of synonyms in control systems, like "property" or "parameter". The description of a device attribute is composed of: (a) its name, which is a unique identifier inside the device class scope and (b) the type description of its data. The attribute description should also support a "readonly" qualifier which specifies that an attribute cannot be "set" (e.g. instrument acquisitions).

Device attributes can be simple or composite. As an example, the settings of an instrument can be interfaced with a set of simple attributes, like "gain" or "timing" or with a single, composite attribute "settings". The acquisitions, as well, can be interfaced with a set of simple attributes, like "vertical_position" or, at the other end, with a single, composite attribute: "acquisitions" returning all settings as well as all acquisitions.

Simple attributes must be supported as they provide direct access to equipment values and adapt well to simple, built-in types. Simple attributes are very useful for the integration of external software packages like a spread-sheet or a data presentation package. Unlike composite attribute, they do not imply class-specific data description (e.g. mean value first, then...) or class-specific type definition (e.g. struct {...} PickupAcq).

Composite attributes are currently used in both ESRF (often) and CERN-PS (some classes). This is extensively used by class-specific software (applications which know what kind of device they are talking

to). However, as both systems are based on persistent objects in the front-ends, a simple attributes interface can be added when not present. Otherwise, the composite attributes can be re-defined as compositions of simple attributes.

2.4 Attributes Data Types

It is necessary to define, in the basic model, a "basic" set of types for the device attributes and this set must be supported by all generic software. CERN-PS, ESRF and CDEV have basic sets which are very similar, so a small "basic" set of types can be conventionally defined comprising: (a) scalar types: "float", "double", "int", "char" and "long", (b) strings, (c) one dimension arrays of scalars, (d) one dimension arrays of strings and (e) multi-dimension arrays of scalars (CDEV only).

A run-time representation of the data types (e.g. a value meaning "data is an array of short integers, maximum size is 5") is required in the attributes's description and there are a variety of such representations. There are some equivalencies between the different representations concerning the "basic" set of types, this means that conversions are possible with some additional conventions (e.g. CDEV's "offset" is ignored, CORBA's multi-dimension arrays are recursive sequences of numeric types, etc.). The additions of "class-defined" types (like a class-defined structure for a composite attribute) also depends on the representation of the data types.

2.5 Device Commands

The device commands describe the actions that can be applied to a device. The transactions (described below) execute the devices commands and the exchange of data between the device and the application is described by the device command.

In the reviewed control systems and APIs, the device commands belong to three categories: "get"/ "set" commands, "simple" commands and "send" commands. The "get"/ "set" commands are based on attributes, values are exchanged either from the device or to the device and the data are described by the attribute description. The "simple" commands imply no exchange of data, they implement commands like "on" or "reset". The "send" commands (or "putget") have a more flexible description, they are composed of a command identifier and a pair of data description: data sent and data received to/from the device. The CERN-PS control system implements the "get"/ "set" commands as well as "simple" commands. ESRF uses "send" commands, as well as "simple" and "get"/ "set", as sub-cases of "send".

The "get"/ "set" commands and the "simple" commands must be supported. The "simple" commands are described with a command name, unique inside the device class scope. The "get"/ "set" commands are described with an attribute identifier and a direction qualifier. The "send" commands are described with a command name, unique inside the class scope, and a pair of data type descriptions.

3. TRANSACTIONS

3.1 Transaction

The transactions (or requests) execute commands on (distributed) devices. A transaction is an association between (a) device(s), (b) device's command(s), (c) data object(s) and (d) device error(s). Synchronous and asynchronous forms of transactions are possible.

Synchronous transactions, as they exist in all control systems, send one command for execution to a device and wait for the response.

Asynchronous transactions can be split into three forms: (a) non-blocking (parallel or deferred) transactions, with some synchronizing action on the application side (cf. CDEV's "flush", "poll" and "pend"), (b) transactions triggered by "events" (institute scope identifiers), like cyclic acquisitions synchronized with the "end-of-last-ejection" event (a CERN-PS example) and (c) cache/monitor transactions, where system

software updates local copies of remote objects, with some subscription actions, these are also triggered by events. All forms of asynchronous transactions require a callback function (a reference to an application function), executed on completion of a command execution.

On top of these five basic transaction forms, a possibility for sending commands to a group of devices should be available for synchronous and asynchronous operations. The multiplicity of the transactions is defined by the multiplicity of the devices (one or many) and by the multiplicity of the commands (one or many). The multiplicities of the data objects and of the errors are derived from this multiplicity (cf. below). A necessary convention is that a transaction with many devices requires that all the devices "belong to" a same device class and that this class contains the description of the transaction's command(s) ("belongs to" means "instance of" or "derived from", cf. "isA"). In this case, one command can be applied to a set of devices, while all devices have a common description of the command. These "many device, one class" transactions are extensively used at CERN-PS.

The "event" based transactions, as well as the "cache/monitor" transactions, might not be part of the basic model, because the functionality and the definition of standard events needs further discussion. Synchronous and (non-blocking) asynchronous transactions should be implemented, as they are already part of CDEV. The ESRF and CERN-PS control systems today implement only a subset of these transaction forms.

3.2 Device error

In the reviewed control systems (and in many others, as well), transactions produce an integer error code per device and per command. It is a fact that the definition of these error codes is local to each institute, but these codes should be encapsulated into a device error object to provide uniform error services, such as: name, message, severity and "generic" errors. Error severity (e.g. none, warning, error) is a widely used concept although there exist some variations in its definition. A "generic" error can be defined as an error which has a meaning for generic software; CERN-PS examples are: "communication_error" or "not_implemented" (for this instance). It would be (at least) useful that a single device error object manages a set of devices, to be used by the "many device" transactions. If some device classes were standardized, class-specific errors could be added to the description of the classes.

3.3 Data object

The current CERN-PS and ESRF transactions use data parameters composed of a type description and a pointer to an application's buffer (C API). In the context of an OO API and also for the implementation of generic "services", data objects are necessary in order to provide services like memory management and type conversions.

Data objects are built on a type description which cannot be completely hidden (some services need to ask what is inside a data object created by another service). In addition, data objects will be exported/imported outside the device's services (e.g. data presentation, archiving). These two points imply that the types' representation (cf. above) is a very important issue which requires further analysis.

To avoid problems in data representation and data transfers a standard data type representation format should be adopted. Today's candidates are the XDR (xternal data representation) format or the "type code" description used in CORBA implementations. Implementing a dedicated data format might cause problems in adapting to commercial products later on.

4. ADDITIONAL CONCEPTS

The basic model defines the "common agreement" concepts. Many additional concepts must be added either as an option which is not relevant in every context (e.g. beam) or as an optional refinement of a basic concept (e.g. discrete attributes). Some examples are described here and there are also many additional candidates ("host", "accelerator", "archive", "state attribute"...).

4.1 PPM and Beam

The CERN-PS control system implements PPM (Pulse to Pulse Modulation) which implies that any execution of a device command must specify on which "PLS line" it applies. The PS complex is divided into three different "PLS" (synchronization) systems. Every beam that the complex can produce has, in its definition, a "PLS line" selection for each "PLS" system[8].

This means that additional classes must be defined in the CERN-PS implementation (e.g. PlsLine and Beam). The "standard" transactions described in the basic model can then be implemented by means of a global instances of these classes. In addition to that, transactions with an explicit beam reference are required, as well as additional data in the description of devices ("PLS" selector, "ppm" flag) and in the description of device attributes ("ppm" flag). These are mandatory extensions for the CERN-PS.

4.2 Operational state

The "operational state" is a qualifier whose values are: "on", "off", "stand-by", "warning", "error" and "unknown". The operational states are used at both CERN-PS (for attributes and devices) and ESRF (for devices only). Presentation services can use this (e.g. colours) as well as alarm services (e.g. if "off", don't care about tolerance warnings). This is an important concept (it could be in the basic model) and a mandatory extension for ESRF.

4.3 Discrete Attributes

A sub-category of a device attribute is the category composed of attributes for which the set of values is restricted to a known small set. Such attributes are qualified "discrete" in some control systems, while non-discrete attributes are qualified "continuous". By convention, discrete attributes are restricted to simple, integer, data types. A typical CERN-PS example is a "control" attribute whose values implements "on", "off" and "stand-by" commands. The discrete attributes have their description extended with the description (name + numeric value) of all "legal" values. They support "get" and "set" commands with values' name as a parameter (e.g. "set" the "control" attribute to the "on" value). This is a CERN-PS extension for the user-interface services. An operational state is also included in the description of each values for discrete attributes.

5. APPLICATION PROGRAMMING INTERFACE (API)

A complete model must include a complete API. In fact, an OO API and a formal definition of the object model are two very close things.

In this paper we only give a class diagram for a C++ API (Fig. 1). It uses OMT notation [9]; the classes are represented with their variables and functions; the associations are represented with roles and multiplicity (the big dot means "many"). It is only a "high-level" description of some classes (e.g. no creation/ destruction, no access function, no "dispatching" functions, etc.); the type of the function's parameters are not displayed (the actual type may not even be unique). Variable length signatures have also been replaced with ellipses ("(...)"). In addition, "simple" command has been renamed "exec".

The API includes some design options. In the illustration, the device class' services and data are implemented by means of a dedicated C++ class: "DeviceClass". An alternative is to implement these services as class variables and class methods (C++ "static") in the Device class. Another design option, in the illustration, is to implement multi-device transactions as a class service.

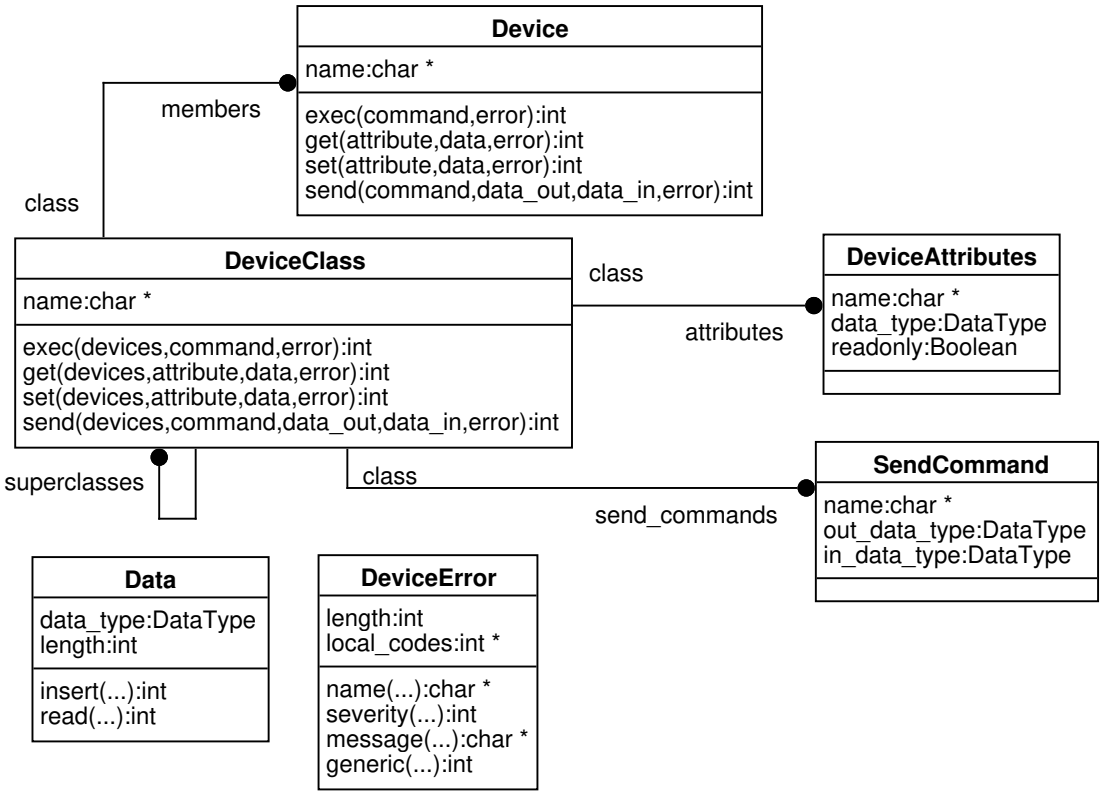

FIGURE 1. Class Diagram

6. CONCLUSIONS

6.1 CERN-PS comments

The model needs to be completed and this should occur in parallel with the identification of "target" applications or services (ones that we want to import or export). On-line modelling may not require, for instance, multi-device transactions, although data-collection services would require these. A particular issue, for us, is to define what "target" services or application use "true" send transactions (i.e. both data out and data in) and for what.

The major constraint we have is that we cannot easily modify the structure of the current front-end objects. These objects are based on attributes and "basic" types and this has a strong influence on our current model. Outside of this scheme, we need to introduce extensions.

We are also extensively using "generic" applications, while "generic", here, means: for any CERN-PS device. Moving these application to a common model would imply some dedicated efforts in the model (e.g. some user-interface extensions) and in the applications. A concrete issue, however, is that the generic applications, as well as the generic services (e.g. data collection), require that the model is clear on the representation of data types, on the data objects and on errors.

As a result of these first concrete activities toward a common model and API, we are convinced that they are highly valuable for the evolution of the console software, that they should continue and that they will, at least, influence any new development at that level.

6.2 ESRF comments

To adapt the ESRF API to the proposed model, is today only possible in a reduced form. With minor modifications, a restricted common API (CERN-PS and ESRF) can be implemented, which is only based on a “get”/ “set” command set, with the basic data types, synchronous transactions and a common set of operational states for devices.

To implement most features of the common model we have to restrict the possibilities of the ESRF control system for some features.

- Today’s commands use the “send” format which allows input and output data transfer for the same command. Restricting to a “get”/ “set” command set would require implementing new commands and to use another philosophy.
- Data types to transfer can be defined by the user as part of a device class. To restrict to simple data types would cut this freedom, but it is necessary for sharable devices.

Various extensions are needed to cope with the model.

- Today we do not have asynchronous transactions in the ESRF control system, but they should be implemented in the near future.The outcome of a standard model could guide the development of asynchronous transactions. The implementation might immediately contain all the agreed functionality and standards. We would not need to port it afterwards.
- The ESRF control system also needs to implement a C++ interface on the application side. Today we are using only a C API. A C++ interface would be one of the results of implementing a standard model.
- A new kind of data and error treatment is necessary on the application side, with the use of data and error objects. A mapping to the current system still has to be found.

The discussions on a common object model and API should continue and will influence the software design of new parts of the control system. The application layer is the most promising candidate for sharable software. But the API and the object model must be clearly defined before an application can represent devices of different underlying control systems.

6.3 Conclusions

The implementation of a standard model is possible. But further work has to be carried out. There are many unresolved issues in this version, the major ones being the data objects and the data type representation.

We should study now what kind of software we want to share and adapt the model and the standardisation to the needs in a second iteration.

Evaluating CORBA showed up that it nicely implements distributed objects, but a standard model for accelerator objects must be defined in order to agree on common interfaces.

CDEV is a good example for an API which offers a large range of functionality and is surely a step forward in collaboration. Nonetheless CDEV doubles all object definitions and is restrictive if it is used on top of an object oriented control system. It can be used as a starting point for a common API, but needs further definition of data types, error treatment and additional support for the object model for querying commands of a device or its class hierarchy.

Finally we would like to conclude by saying that the work described in this paper will be continued. The results achieved so far have convinced us that collaborative approaches to object technologies are the way to follow for future accelerator controls.

REFERENCES

[1] J. Cuperus, F. Di Maio*, C.H. Sicard, “The Operator Interface to the Equipment of the CERN PS Accelerators” in ICALEPCS 1993 proceedings, Berlin, Germany, Nucl. Inst. and Meth., A352(1994) pp 346-349.

[2] Franck Di Maio, Alessandro Risso, “The CERN-PS Equipment Access Library” PS/CO/Note 93-87, CERN, Switzerland, 1993

[3] A. Goetz, W.D. Klotz, J. Meyer, “Object Oriented Programming Techniques Applied to Device Access and Control” in ICALEPCS 1991 proceedings, Tsukuba, Japan, Nucl. Inst. and Meth., A352(1994) pp. 514 - 519.

[4] A.Götz, “Device Server Programmer‘s Manual”, ESRF, France, 1993.

[5] Chip Watson, Jie Chen, Danjin Wu, Walt Akers, “cdev User’s Guide”, CEBAF, USA, July 1995

[6] “AIX Version 4.1 SOMobjects Base Toolkit User’s Guide”, SC23-2680-01, IBM, October 1994.

[7] “Orbix Programmer’s Guide”, IONA Technologies Ltd., Release 1.3 July 1995

[8] Julian Lewis*, Vitali Sikolenko, “The New CERN PS Timing System” in ICALEPCS 1993 proceedings, Berlin, Germany, Nucl. Inst. and Meth., A352(1994) pp 91-93

[9] James Rumbaugh et al., “Object-Oriented Modelling and Design”, Prentice-Hall, ISBN 0-13-630054-5, 1991