

CICERO: Control Information systems Concepts based on Encapsulated Real-time Objects

R Barillere¹, E. Bernard², C Escrihuela¹, W Harris³, J-M Le Goff¹, R McClatchey³

¹ECP Division, CERN

²SPACEBEL Informatique S.A

³Dept Of Computing, Univ West of England

Abstract: Modern High Energy Physics experiments and accelerators require increasingly sophisticated control systems which turn out to be difficult and costly to maintain with the present technology. The situation will seriously worsen in the LHC era. R&D departments of industrial companies are directly concerned with similar difficulties in power plants and complex automated systems. CICERO combines efforts from institutions and industry to address this problem. The main building blocks of a generic control information system have been outlined and constructed. A middleware called Cortex [2] providing the integrating capabilities and the extended communication support for the integration of heterogeneous control products has been developed. To support the long life-cycle of these large projects, the middleware relies on CORBA (Common Object Request Broker Architecture) [1], the powerful emerging standard proposed by all the major computer manufacturers for transmission of objects over networks. In addition, CICERO provides a series of common components, to be used as templates for the construction of specific CORBA-compliant control services such as Archiver/Retriever, Finite State Machines and GUI. A Prototype has been built as a first step towards our main goal of providing technical solutions which could later be the major components of a basic turnkey system for medium-to-large scale HEP experiments and accelerators. The key design constraints and concepts of CICERO are identified in this paper.

1 Introduction

Modern experiments and accelerators at CERN utilise complex equipment for the control of their hardware. The operation of these equipment is managed by increasingly sophisticated control systems software. With the approval of the next generation of accelerators and experiments, the Large Hadron Collider (LHC), this increase in control system complexity continues apace. The technology employed to generate and maintain the LHC beams will require considerable advances in both hardware and software design. The LHC experiments themselves will involve collaborations of many tens of institutes and over 1,000 physicists, engineers and computer scientists from around the world. The knowledge required to construct and monitor parts of the LHC experimental (sub-) detectors will be distributed between these institutes making it difficult to impose standards. There will consequently be significant problems of information transfer to ensure that each (sub-) detector retains autonomy of control but can work with other (sub-) detectors for data-acquisition. In addition, the LHC detectors (both in the accelerator and in the experiments) will be required to have a long lifecycle since the experiments will take data for several years. As a consequence, maintainability will be an important consideration. Furthermore, the accelerator and experimental groups will also be working to very tight time and cost schedules. As the available hardware grows, so the control systems are required to grow from an initial lab-based test system to test-beam operation and ultimately to the fully-fledged accelerator and experimental systems. Control systems designers are becoming increasingly aware that what is needed to facilitate the development of LHC control systems is an integrating framework (or 'software bus') which enables the building of distributed control systems where responsibilities are distributed amongst control elements that have to collaborate together.

Experience of the development of control systems for the LEP experiments [3] and anticipation of the increased demands which will be generated by the new and larger experiments for LHC, has illuminated the main constraints for the design of such an integrating scheme. Firstly, since the LHC control systems will be equipped with typically more than 100,000 sensors and activators, the **management of control system complexity** is a major consideration. Object-oriented design techniques embodying abstraction, inheritance and the use of classes and objects will aid the design here. Secondly the problem of **concurrent and collaborative software engineering** requires addressing. This is particularly true when the development of the control software is carried out by engineers who are separated geo-

2 The Problem Statement

HEP experiments and accelerators are normally developed by various teams with specific task assignments. Usually, the different parts of a HEP experiment are developed by different institutes in their laboratories and gathered together later after being tested. The experiments are tuned up at CERN before new tests, or calibration and operation. For instance, the gas system and the high voltage system of a muon detector may be produced and tested by two different teams for ultimate integration in an experiment in the accelerator tunnel.

In HEP, control systems are, by definition, required to maintain equipment in a constant state over the lifetime of the accelerator or experimental data-taking period. Invariably this has led to problems when existing systems need to be upgraded or when new control systems are being specified. Consider these two cases: in the former case, when an existing system is being amended, constraints of time, manpower and cost usually leads to a partial re-engineering of the existing system rather than a complete rewrite of the control system. During the partial re-engineering of the heterogeneous control system, two options are normally considered: either the incorporation of industrial solutions (which can be difficult to adapt) or the investment of considerable in-house manpower to provide equivalent functionality. Other issues that arise with partial re-engineering include hardware and software incompatibilities, lack of commitment to reuse software between control systems and the fact that re-engineering often introduces short- and long-term instabilities with the consequent need for increased human interventions. In the second case when a new control system is being specified 'from scratch', the costs of implementation can be high and with new technology maintenance can be an issue. In this case control systems designers often re-use parts of existing control systems either developed at the laboratory or from industry. To re-use subsets of existing control systems can be a risky strategy since the hardware available for control systems continues to evolve rapidly while integrating industrial control solutions with 'home grown' solutions is often technically infeasible or difficult.

Most of the difficulties identified in amending existing control systems or redeveloping complete control systems from scratch are problems of software engineering. In both cases the software engineering problems are those of taking a bottom-up approach to solving distributed computing issues in heterogeneous environments. Since HEP laboratories are not leaders in computing, designers must make use of industrial solutions and deal with the attendant problems of their integration. Essentially what is required is reusable, distributed and collaborative control systems solutions which enable both partial re-engineering of existing systems and facilitate the construction of new control systems from scratch. These solutions must also minimise maintenance costs and provide highly reliable and deterministic control systems.

Existing control systems software technology has been based on the client-server model of computing using proprietary protocols such as TCP/IP and/or Remote Procedure Calls (RPCs) for communication across the distributed network of control devices. This infrastructure requires peer-to-peer agreement between the collaborating controls partners and cannot provide for dynamic invocation of services. Such systems also suffer from issues of scalability: when the number of clients rises, the load on the server also rises so that system throughput of messages and service calls falls. In addition, using TCP/IP and/or RPCs it is not possible to easily move the server process from one platform to another since the clients must be reconfigured to be made aware of the servers location. Consequently, the user-supplied controls software must be technology independent and free from the confines of the client-server model of distributed computing. The user software must be data-driven and designed to be reused in a control system infrastructure which has been put together using a top-down approach. The user must be able to plug a new controls service into a controls infrastructure with no disruption to the operation of the existing controls systems and with no or minimal recoding.

The challenge is how to build such heterogeneous, distributed and collaborative control systems re-using as much as possible of the existing controls software. One solution which has been proposed in the physics controls community [4] is that of a so-called 'software bus'. In an analogy with hardware, a software bus structure is where different modules plugged into the same bus may cooperate if their bus interfaces adhere to a given standard. A software bus must hide the underlying technology from control systems designers - they have no interest in distributed systems technology nor should they have. It must also scale with the technology used so as to preserve designers' investment and must provide more facilities than simple address and data exchange. In software terms this would translate into agreeing on a common software layer with a unique Application Programming Interface which permits the implementation of shareable software as separate and independent modules. One basis on which to build a software bus is the recent Common Object Request Broker Architecture (CORBA) specification [5] for the implementation of object communication in distributed systems. Such a standard facility supplemented by standard control software, if jointly adopted by laboratories using control systems for physics, could make future controls software more 'shareable' and reusable. The joint adoption of a software bus would be complementary to, not competing with, collaborations like EPICS [6] and products like Vsystem. Examples of applications which could benefit from the use of a software bus in the controls environment include: User Interfaces (development & management system), console manager, on-line help facilities, database access control, realtime data management, data logging, archiver/retriever, alarm handling system, equipment access, system configuration mechanisms.

The following section describes the CORBA specification and illustrates how it can be used as the basis of a software bus. In addition, this section notes where further functionality is required on top of CORBA to satisfy the needs of control systems designers.

3 The OMG Standards

The Object Management Group (OMG) is an industry consortium dedicated to the goal of developing an object-oriented architecture for the integration of distributed applications. The OMG emphasises as its objectives the interoperability, reusability and portability of components and its operating procedures attempt to insure that any specifications adopted as standards have their basis in commercially available software. (OMG, however, only deals with interface standards, not the software itself). The OMG has published several documents including:

- Object Management Architecture (OMA) Guide [7]
- Common Object Request Broker Architecture (CORBA) [5]
- Common Object Services Specification (COSS), Volume 1 [8].

3.1 OMG Object Management Architecture

The OMG's Object Management Architecture (OMA) defines a Reference Model which identifies and characterises the components, interfaces and protocols that compose a distributed object architecture. The four main elements of the OMA are:

- The Object Request Broker (ORB) which enables objects to make and receive requests and responses in a distributed environment. The ORB and related facilities are defined by the CORBA specification [5].
- Object Services - a collection of services (interfaces and objects) that provide basic functions for using and implementing objects.
- Common Facilities [9] - a collection of services that provide general purpose capabilities useful in many applications. Control examples of Common Facilities include Archiver/Retrievers and Loggers (see figure 1).
- Application Objects - objects specific to particular end-user applications.

The Object Request Broker in the OMA is viewed as a 'messaging backplane' spanning multiple systems. From the software point of view, the ORB may consist of a single software component, or multiple cooperating (and possibly heterogeneous) software components. The Object Services, Common Facilities and Application Objects in the OMA correspond to different categories of object implementations. In the OMA Reference Model, the purpose of categorising objects into Object Services, Common Facilities and Application Objects is to guide OMG's strategy for developing interface specifications. Object Services include lower-level (and thus the most crucial) object interfaces; and Application Objects reflect the need for independently-developed application interfaces for which the OMG will never develop specifications.

In the OMA, an object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client (which can be another object, or a non object-oriented application). An operation is an identifiable entity that denotes a service that can be requested. The Object Request Broker (ORB) component of the OMA supports cli-

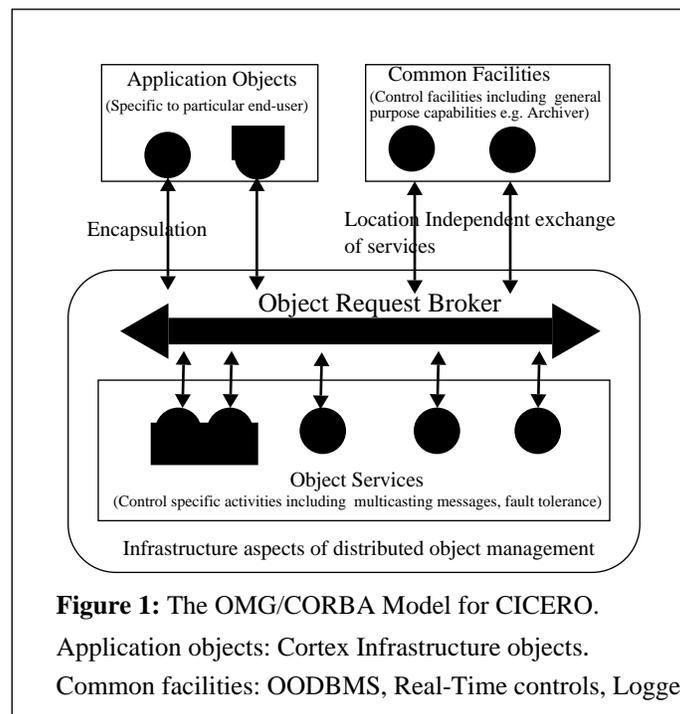


Figure 1: The OMG/CORBA Model for CICERO.

Application objects: Cortex Infrastructure objects.

Common facilities: OODBMS, Real-Time controls, Logger

ents making requests to objects. A request causes a service to be performed on behalf of a client and any results of executing the request to be returned to the client. The ORB is responsible for finding the object implementation corresponding to the target object specified in the request, invoking that implementation, passing it the request for handling and returning the results.

3.2 OMG CORBA

The CORBA specification defines a number of different interfaces (see figure 1). The CORBA specification defines an Interface Definition Language (IDL) for defining the interfaces of objects in [5].

The CORBA IDL plays a role similar to that of the IDL in a Remote Procedure Call (RPC) facility. IDL specifications can be compiled to produce client stubs and implementation skeletons (essentially server stubs). The use of a compiled client stub to access an object interface is referred to as the static interface. A Dynamic Invocation Interface (DII) is also provided that allows clients to construct requests at run-time to objects whose types might not have been known at compile-time.

Objects are made known to the ORB by being registered with an object adaptor. The definition of a Basic Object Adaptor (BOA) is contained in CORBA. The BOA is generally designed to handle objects that are independently constructed, and must be handled individually by the adaptor. The CORBA specification also identifies other types of adaptors that might be more appropriate for objects implemented in different ways. For example, CORBA identifies a DBMS object adaptor that would be more appropriate for objects defined within a database environment.

3.3 OMG Object Services

An Object Service defines interfaces and sequencing semantics that are commonly used to build well-formed applications in a distributed object environment. Each Object Service provides its service to a set of users. These users are typically Application Objects or Common Facility objects that, in turn, provide support for specific application domains like network management or complex systems controls. In non-object software systems, a system's Application Programming Interface (API) often is defined by a single interface. The OMG Object Services API is modular; particular objects may use a few or many Object Services. By being object-oriented, the OMG Object Services API is extensible, customisable and subsettable; applications only need to use services they require.

The operations provided by Object Services are made through the CORBA IDL or through proposed extensions to IDL compatible with the OMG Object Model. However, while OMG requires an IDL interface for each object service, implementations of object services need not themselves be object-oriented. Such implementations may continue to support non object-oriented interfaces for compatibility with an existing product's API or with non-OMG standards, in addition to an IDL interface. Also, objects need not use the implementation of basic operations provided by Object Services, nor must objects provide all basic operations.

Various Object Services have been produced at various times by OMG. Examples include Event Notification, Lifecycle and Naming Services which are specified in [7]. A Persistence Service specification has been approved and will appear in a forthcoming update of [7].

3.4 Implications of Using CORBA in Controls Environments

Any software bus to be used in the controls environment that is based on CORBA can thus provide standard information exchange between controls devices as well as hardware independent services. Controls systems users can use this integrating infrastructure to provide services across the control system and potentially between control systems. Since the infrastructure is based on CORBA, it will be both scalable and reconfigurable and therefore a control system could be optimised to handle on-line loads.

CORBA provides 'plug-and-play' at the object level and the possibility of moving controls objects around the control system without stopping the client applications. Further it facilitates network independence and can run TCP/IP or RPCs on networks such as Ethernet, ATM or SCI. Adherence to OMG standards such as the Common Object Services, (COSS), provides facilities such as naming and authentication as well as object adaptors such as that available for OODBMS access.

CORBA then provides much of the functionality required as a basis for a software bus in physics controls systems. What it fails to make provision for is message multicasting, service invocation/handling support, fault tolerance and object management aspects specific to the domain of control systems. Additional software is therefore required alongside CORBA to complete the software bus and thereby to provide the functionality demanded by control systems users. The CICERO project has created Cortex [2] to provide this controls-specific software and its design concepts are explored in the following sections.

4 Cortex Design Concepts

The Cortex element of the CICERO project provides the software bus [2] which runs on top of CORBA. It enables the building of distributed control systems where responsibilities are distributed amongst control elements that have to collaborate together. Cortex supplies extra features on top of CORBA: multicasting of messages and service handling in addition to a series of control services (archiver/retriever, logger, state machine, distributed user interface) which could be standardised in the framework of OMG.

In addition to the functionality already identified, Cortex promotes the modularity of distributed control systems in order to allow some integrated control systems to be operational while some others are not. Cortex will also support users during the reengineering phases of the integrated control systems by allowing them to reuse and integrate existing software.

Cortex supports the entire collaborative distributed control system life cycle by:

- providing a multi user development environment,
- supporting the testing and simulation of users' control elements with respect to the integration within the Cortex control system,
- offering tools to operate and protect the control system from faulty processes implementing some control elements,
- allowing the integration of already existing control systems,
- allowing collaboration with other autonomous control systems.

Cortex is intended to provide a control system designer with the ability to integrate his particular control system efficiently and with minimal cost and effort.

The architecture of any distributed control system will change during the lifetime of the accelerator or the experiment it is operating. The control system integrating platform, Cortex, must therefore support a mechanism to allow a new version of the distributed control system to be in preparation while an older one is operated on-line. It shall also support the backup and the restore of a given version of the collaborative distributed control system. Furthermore, tests and validation (and possibly simulations) of a new configuration will be needed before it is applied to the operating on-line system.

These constraints have led to a so-called dual-face approach (see figure 2.) being taken in Cortex:

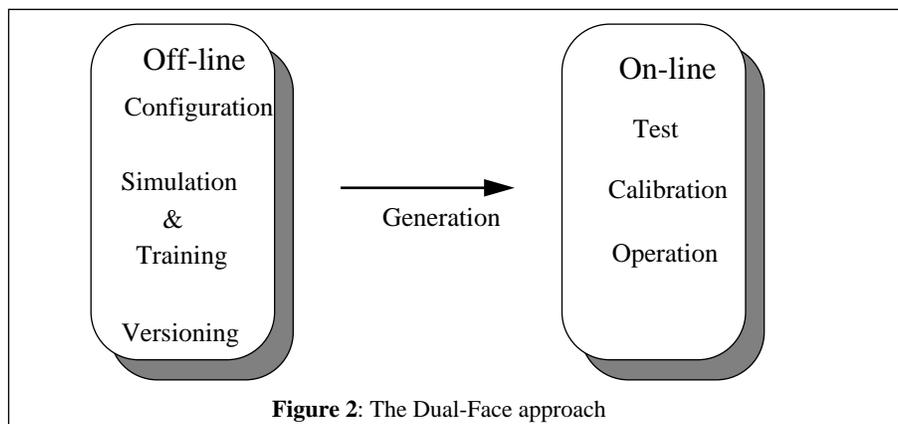


Figure 2: The Dual-Face approach

- an off-line Cortex representation is required to handle the logical descriptions of the architecture of the distributed control system and to describe the various information and commands to be exchanged between the different control elements. This so-called *Repository* also holds the description of the hardware model from which the on-line distributed control system is constructed, and

- an on-line Cortex representation is also required through which the control elements can exchange information and commands in a pseudo- 'plug-and-play' fashion. Control elements operating within the Cortex Infrastructure can access the Cortex Repository through this so-called *Infrastructure*. A generation mechanism is provided to facilitate updates of the on-line Infrastructure according to the Repository contents.

The responsibility of Cortex is twofold: on the one hand it has to support the description of the architecture of the distributed control system and the definition of the information and commands to be exchanged between the control elements. The off-line representation of the control system therefore addresses the issues of the **management of control system complexity** and that of **concurrent and collaborative software engineering** identified earlier. On the other hand, Cortex must transport and distribute these data and commands to the appropriate control elements when part or all of the distributed control system is in operation. This distribution must be independent of the number of hardware elements on which the various control elements are operating. The integrating framework must be flexible enough to support the addition or removal of control elements, without deteriorating the operation of the rest of the distributed control system. The on-line Cortex Infrastructure thereby addresses the demands of **stability, flexibility and availability** of control system elements and that of providing **balanced and distributed processing** for the control system. Cortex is not a product in the sense of FactoryLink but rather a resolution of a set of issues specific to the controls world which may point the way ultimately towards the establishment of a real controls standard.

5 Cortex off-line

Off-line a Cortex control system is visualised as a collection of collaborating components. These components map onto those on-line processes in a control system which produce or consume information and they can be of various kinds. They may or may not be “control components”. For example gas systems or high voltage systems are managed by software that are control components. Each has a read-out system and has responsibility for the hardware. Indeed some control components may need real time capabilities, for instance, elements interfacing front-ends which perform interlocks. On the other hand, an on-line documentation element or a user interface element are “non-control elements” - they do not have responsibility for the hardware. They may, however, support high level control functionalities such as alarm filtering, user assistance, preventative maintenance etc. Cortex can therefore be used to integrate software which implements facilities common to any kind of control element such as loggers, archivers, retrievers, GUIs, final state machines, etc.

Compositeness is the mechanism proposed in Cortex to support the logical encapsulation of a distributed control system. Components may be composed of smaller components. Such components are referred to as composite components and are often used to separate functions or to provide the granularity required by the underlying hardware (figure 3). Users must be able to operate the complete control system from a global standpoint or operate each component independently via the composite components. Additionally, users must be able to specify communication requirements at any level of component, regardless of the inherent hierarchical organisation.

Grouping is a complementary mechanism to Compositeness proposed in Cortex to allow specification of information and command exchange at any level of granularity. A collaboration group is composed of a set of components that make available certain information and services to the other components in the group. Two components will be able to exchange information and commands if and only if they belong to the same collaboration group. Components can be part of more than one group. Collaboration groups can be established across encapsulations of sub-systems. The combination of compositeness and collaboration groups allows the user to refine and optimise the communication at an appropriate level of control system component.

Within a collaboration group, a component can provide information to other components by publishing items (data or services). If granted permission by the publisher, any component of a collaboration group (other than the publisher) can access this information by subscribing to the published items. The set of published and subscribed items handled by a component within a collaboration group is called a component interface. A component usually has a different interface for each collaboration group in which it is a member.

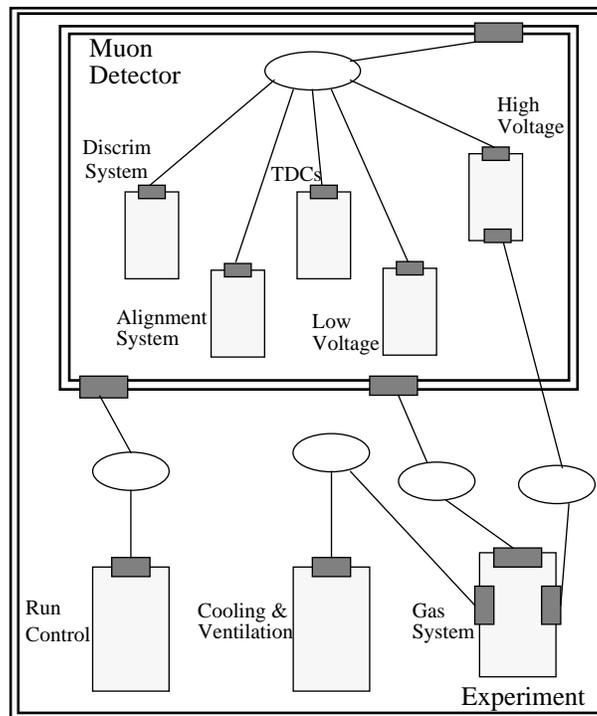


Figure 3: (Composite) Components & Grouping

Key: Groups are represented by ellipses, Components by rectangles and Component interfaces by shaded boxes. Composite Components are shown as double rectangles.

Compositeness and collaboration grouping are the mechanisms for supporting information abstraction in Cortex: some of the collaboration groups can be organised to enforce encapsulation of sub-systems (so-called *strict encapsulation*). This will allow composite components to subscribe within their encapsulation and publish refined information to other composite components in other collaboration groups.

6 Cortex on-line

The Cortex on-line Infrastructure is a set of entities responsible for the distribution of information and for the transmission of service invocations to the appropriate components, according to the model in the Cortex Repository.

The Cortex Infrastructure provides two selectable ways of data exchange between components. The *push mechanism* which allows components to push new information into the Infrastructure or to receive information from the Infrastructure and the *pull consumer mechanism* which allows components to retrieve information from the Infrastructure at their convenience. The push mechanism is recommended for security information such as alarms. The pull mechanism is more suitable for monitoring components offering refreshed information upon user request. In both cases, only information specified in the Cortex Repository will be transported and delivered to the appropriate components. Version inconsistency between the information sender and the information receiver are handled by the Infrastructure at message level. For example, a component may pull from the Infrastructure items which are no longer published (because of modification, replacement or deletion). The dynamic information contained in these items is no longer refreshed and the corresponding component will be informed. Time stamps will contain the last time these items have been refreshed.

The on-line architecture supports a separated set of messages to handle service invocation called command messages. Commands are persistent in the Infrastructure from the moment they have been issued until they are completed (successfully or not) or refused by performers. Two basic types of services are available. Firstly, services can be cancellable or non-cancellable: a requesting component can cancel its request while the component offering the service is processing the command. Secondly, services can be multi- or single-requestable: more than one requesting components can issue a command to the same performer component.

More than one requesting component can invoke the same single-requestable command hence addressing the same performing component. The on-line architecture handles possible access conflicts using an internal protocol based on locking. Commands are not direct implementations of operations in the OMG Object Model, which do not support some specific control functionalities such as authentication, availability and progress report features.

7 Mapping OMG Standards onto the Cortex Design Concepts

In Cortex, an ODBMS is being used as the vehicle for the off-line Repository to support a standardised access for the CORBA objects in the on-line Infrastructure. The Repository has been designed to support the notions of Compositeness and Collaboration Groups, Publishing and Subscription and Components as described in section 5. ODBMSs provide persistence of object information, and all the advantages of DBMS systems such as version management, concurrency control, security and recovery. This enables control system designers to save a full description of the experiment or accelerator setup in an object base and to modify that description over time as the equipment grows.

The Cortex on-line Infrastructure is instantiated as a set of CORBA objects responsible for the distribution of information and for the transmission of commands to the appropriate components, according to the description resident in the Cortex Repository. On-line objects in Cortex are written in C++ and use Iona Technologies implementation of CORBA, called Orbix, for object location, access and communication services. The physical location of the components and the Infrastructure will depend on the hardware setup available. This setup may evolve with time for performance reasons or for maintenance purposes. In these cases, the system functionalities must be maintained when part of the hardware is changing. This operation should take place without disturbing the operation of the parts of the distributed control system which are not involved in this upgrade. The location transparency is fully supported by CORBA. CORBA makes no provisions for message sender identification. Any program can potentially send a message to a CORBA object. To avoid unpredictable overloads, Cortex provides an authentication mechanism to ensure that any new starting process is effectively representing a component known to the Cortex Repository.

As an example of the use of Cortex + CORBA, consider the case cited by Clausen [4] at the last ICALEPCS conference where a data logging system is required which allows data to be stored in a circular buffer, later in a database and finally on tape. From the operators console, data needs to be retrieved from all three sources (buffer, database & tape). Individual solutions exist for each form of data storage (EPICS, ORACLE, TapeManagers). The problem is providing a facility that irons out the differences between accessing these three different technologies and allows the operator to access the data regardless of the underlying technology. The software bus of Cortex + CORBA can provide the solution. Adherence to the CORBA standard enables independently defined (client and server) applications to be specified. All interaction between these services is then handled by the Common Object Request Broker. Furthermore the Cortex layer of software then deals with all issues directly related to inter-connecting controls applications and their inter-operation.

As an example of the use of the OMA standard in Cortex the next section shows how Cortex provides reusability both of control system components and complete control sub-systems.

7.1 Reusability and Cortex

The basis for re-use in the CICERO project is through the use of CORBA IDLs and ORBs. Reusability can be exploited in CICERO both at a level internal to components and at a level external to components. Firstly at the internal level, consider the re-use of component code as the hardware of the control system is evolving. By using IDL stubs for client invocation and an IDL skeleton to package the existing (component) code for CORBA compliance, the control system can be allowed to grow whilst reusing existing components. As an example, consider the incorporation into a UNIX-based control system of an existing data logger which runs on VMS/ORACLE. To reuse this component it is necessary only to build an IDL interface to this logger, using an ORB which supports VMS. Then a UNIX component will be able to send messages such as store and retrieve through this interface to log information without having to know the complexities of UNIX-VMS translation. In addition, such a component will be able to log any additional Cortex messages issued by any other existing components according to the description stored in the Cortex Repository.

Using CORBA IDL for interface specification permits the sub-division of a large software module into smaller, easier-to-manage units with simpler functionality. This facilitates reusability in that it supports:

- the evolution and partial upgrade of complex control systems and
- the re-use of existing (legacy) systems through CORBA objects.

In this example of reusability, partial re-engineering of component code is again required since the IDL specification takes place inside the component code.

CICERO is also able to exploit reusability at a level external to components. This can be achieved through the use of so-called Reusable Components. These can be regarded as templates for components with logical input/output which can be instantiated as many times as there are physical devices. At the time of instantiation, there may be no hardware assignment - only at the time of assignment will the desired functionality become apparent and the component code reused. For example, consider a 16-channel Analogue to digital Converter (ADC) read-out component coupled with an ADC Converter component. The first component is hardware independent, except for the ADC gain, and can be reused as many times as ADCs are needed in the system. The second component is context dependent and can evolve with the hardware of the system. If the second component is data driven and obtains its configuration data from the Repository, then the ADC/ADC Converter pair can be reused with no code modification in any subsystem of the experimental setup. Cortex offers the possibility of decomposing a complex control system into data acquisition components, command components and automation loop components offering the possibility of reusing or sharing components in different control systems.

It is also possible in CICERO to sense whole sets of components at the control system level. Consider two Cortex control systems developed independently and merged at a point in time. For example, a development or test setup

9 Acknowledgements

In a collaborative project such as CICERO RD38, each collaborator deserves the thanks of the authors. Rather than cite each contributor, thanks are extended to members of RD38 from BARC (Bombay, India), CERN (Geneva, Switzerland), CIEMAT (Madrid, Spain), IVO International (Helsinki, Finland), KFKI (Budapest, Hungary), OBLOG (Lisbon, Portugal), SEFT (Helsinki, Finland), SpaceBel (Brussels, Belgium), UID (Linkoping, Sweden), USDATA (Dallas, USA), UWE (Bristol, UK), Valmet Automation (Tampere, Finland) and VTT (Oulu, Finland). Special thanks go to those involved with the development of Cortex.

10 References

- [1] J. R. Rymer, 'Common Object Request Broker - OMG's New Standard for Distributed Object Management', *Network Monitor* Vol. 6, No 9, pp 3-27 (1991)
- [2] R. Barillere et al., 'The Cortex Project: A Quasi- Real-Time Information System to Build Control Systems for High Energy Physics Experiments', *NIM A* **352**, pp 492-496 (1994)
- [3] R. Barillere et al., 'Ideas on a Generic Control System Based on the Experience of the Four LEP Experiments Control Systems', Proceedings of the ICALEPCS '91 Conference, Tsukuba, Japan pp 246-253 (1991).
- [4] B. Kuiper et al., 'Panel Session on Software Sharing' at the ICALEPCS conference 1993, *NIM A* **352**, pp 501-515 (1994).
- [5] CORBA: 'Common Object Request Broker Architecture and Specification (CORBA)', Revision 1.2, Object Management Group Inc., OMG TC Document 93.12.43 (1993).
- [6] L.R. Dalesio et al., 'The Experimental Physics and Industrial Control System Architecture: Past, Present and Future', *NIM A* **352**, pp 179-184 (1994)
- [7] OMG: 'The Object Management Architecture (OMA Guide)', Revision 2.0, Object Management Group Inc., OMG TC Document 91.11.1 (1992).
- [8] OMG: 'Common Object Services Specification (COSS) Vol. 1', Revision 1.0, Object Management Group Inc., First Edition (1994).
- [9] OMG: 'Common Facilities Architecture (CFA)', Revision 4.0, Object Management Group Inc., OMG TC Document 95-1-2 (1995).