ACS – THE ADVANCED CONTROL SYSTEM

Mark Plesko*, Klemen Zagar, Matej Sekoranja, Janez Dovc, Miha Kadunc, Igor Kriznar, Ales Pucelj, Gasper Tkacik, Igor Verstovsek, Dragan Vitas, J. Stefan Institute and Cosylab Ltd., Slovenia
Gianluca Chiozzi, Birger Gustafsson, Bogdan Jeram, European Southern Observatory, Germany

Abstract

The ACS is a CORBA-based control system framework with all features expected from a modern control system. It has been recently installed at the ANKA light source in Karlsruhe, Germany and is being used to develop the ALMA control system. ALMA is a joint project between astronomical organisations in Europe, USA and Japan and will consist of 64 12-meter sub-millimetre radio telescopes. ACS provides a powerful XML-based configuration database, synchronous and asynchronous communication, configurable monitors and alarms that automatically reconnect after a server crash, run-time name/location resolution, archiving, error system and logging system. Furthermore, ACS has built-in management, which allows centralized control over processes with commands such as start/stop/reload, send message, disconnect client, etc. and is fine-grained to the level of single devices. ACS comes with all necessary generic GUI applications and tools for management, display of logs and alarms and a generic object explorer, which discovers all CORBA objects, their attributes and commands at run-time and allows the user to invoke any command. A Visual configuration database editor is under development. An XML/XSLT generator creates an Abeans plug for each controlled object, giving access to all Abeans applications such as snapshot, table, GUI panels, and allowing one to use the CosyBeans GUI components for creating Java applications. For those that write their own control system, ACS allows to define own types of controlled data and own models of communication, yet use powerful support libraries as long as one adheres to some rules in the form of programming patterns. ACS users several standard CORBA services such as notification service, naming service, interface repository and implementation repository. ACS hides all details of the underlying mechanisms, which use many complex features of CORBA, queuing, asynchronous communication, thread pooling, life-cycle management, etc. Written in C++ and using the free ORB TAO, which is based on the operating system abstraction platform ACE, ACS has been ported to Windows, Linux, Solaris and VxWorks. The applications are written in Java and run on any JVM-enabled platform. ACS is based on the experience accumulated with similar projects in the astronomical and particle accelerator communities, reusing and extending concepts and components of implementation. Although designed for ALMA, ACS has the potential for being reused in other new control systems, as proven by the nearly seamless installation at the ANKA this spring.

1 INTRODUCTION

When developing control systems for different kinds of applications, we [1] have found that some of the issues we needed to resolve were present in all of them. Apart from this, we identified many best practices that contribute to the resulting control system's usability and quality.
We have decided to collect the solutions to the ubiquitous issues in a framework, which would allow control system developers to focus on their concrete application, and not have to resolve the same problems over and over again.
The result of our work is a framework called the *Advanced Control System* (ACS). The requirements that the ACS was designed to fulfil are:

- Strong type checking to avoid most common errors at compile time.
- Use of *Common Object Request Broker Architecture* (CORBA) as the middle-ware that handles complexities of network communication.
- Support for object-oriented modelling of the control system's devices to aid developers and operators alike in taming the inherent complexity of the system. Thus, all devices are represented as distributed (CORBA) objects.
- Support for the *pull model* of control, where the user initiates queries for values of monitor points.
  Support for the *push model* of control, where the monitor points notify the user of changes (*monitors, alarms*).
- Support for handling errors in the control system, ranging from hardware failures, via control system malfunction, to networking problems.
- A well-defined interface to a configuration database, and a default implementation of this interface based on XML [2].
- The *component-container model* for managing deployment and interactions of distributed objects.
  Support for basic services such as logging, archiving, security, etc.

2 HIGH-LEVEL ARCHITECTURE

From the high-level point-of-view, ACS fits well in a multi-tier architecture of a control system (see Figure 1):

- The *data-access tier* for working with the data in the configuration database and other databases (e.g., the archive database with historical values of control point values).

* mark.plesko@cosylab.com

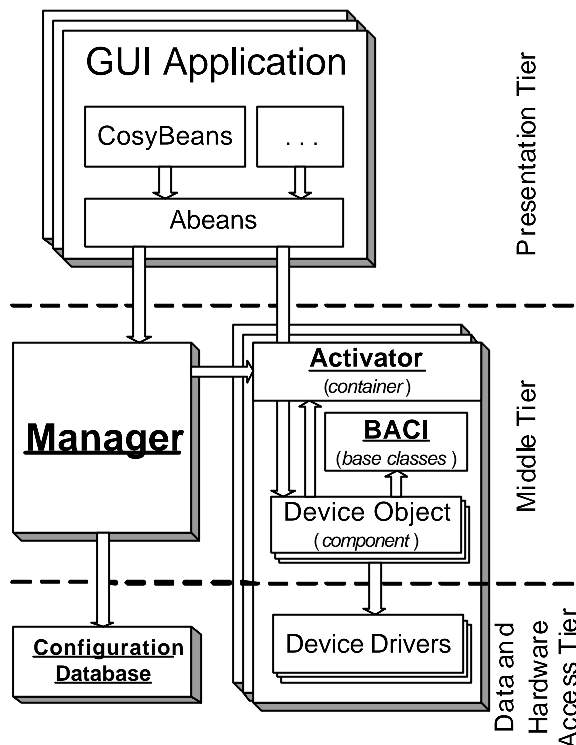

Figure 1: High-level architecture of the control system. Parts written in bold and underlined are part of the ACS framework implementation. Arrows indicate the *uses* relationship (e.g., *a Device Object uses a Device Driver*). Shaded boxes denote standalone processes.

- The *hardware-access tier* (device drivers) for retrieving the data from monitor points implemented in hardware, as well as sending commands to control points.
- The *middle tier* (also called *business logic tier*) where the devices and their interactions with the rest of the system are modelled using *Device Objects* (DO), and through which they can be monitored and controlled.
- The *presentation tier* that exposes the control system to the human (GUI) or machine (API) user.

The architecture is such that only the following interactions of tiers come into play:

- Business-logic tier uses the services of the data-access and the hardware-access tiers.
- Presentation tier uses the services of the middle tier.

The ACS framework focuses on the data-access and middle tiers.

2.1 Enforcing Architecture through Generators

Adding a new type of devices to a control system, or modification of an existing device type, requires modification of all tiers. However, the modification is often trivial (e.g., adding a property to a class, or adding a new field to a configuration database). We decided to automate some of these tasks, because manual modifications tend to produce unnecessary errors.

To that end, we only modify the business-logic tier, then extract information about modified device types using CORBA introspection (the *Interface Repository* service) and encode it in XML format [2]. Then, we use XSL/T transforms [3] to generate Abeans plugs from XML device descriptions.

We are in the process of introducing a more powerful source code generator, which uses a specialized language for describing source code templates (the *Extensible Program Generator Language* [4]). We anticipate to use this generator for producing other artefacts as well, such as BACI-compliant Device Objects that model the devices and schemas for the configuration database.

3 MIDDLEWARE

Middleware is the cornerstone of distributed computer systems, as it enables the interaction of distributed components, residing on different hosts. There are several middleware implementations available, ranging from the OMG's *Common Object Request Broker Architecture* (CORBA), through the business-centric *Simple Object Access Protocol* (SOAP) to Microsoft's proprietary *Distributed Component Object Model* (DCOM), *.NET Remoting*, and Sun's Java *Remote Method Invocation* (RMI).

3.1 CORBA

In ACS, we have decided to use CORBA [5] as the middleware. In particular, we have chosen to use the TAO implementation of CORBA [6]. Our choice is based on the following facts:

- CORBA is platform independent.
- CORBA provides many pre-implemented services, such as the naming, logging and notification services.
- In our benchmarks, TAO CORBA has been found to offer superior performance to other alternatives. It also supplies all the additional CORBA services that we need.
- TAO CORBA is built atop ACE (*Adaptive Communication Environment* [7]), which is a highly portable collection of operating system wrappers and common design pattern implementations. Consequentially, ACS is portable to most platforms that ACE is portable to (e.g., Windows, Linux, Solaris and VxWorks).

4 MANAGEMENT SUBSYSTEM

Management services are essential in all of today's distributed computer systems. The role of the management subsystem is to provide management services to the rest of the control system. Management services facilitate deployment of control system building blocks on host computers, and enable them to find each other. In addition, it enables human operators to view and manage the state of the control system through a generic user interface.

4.1 MACI

Management and Access Control Interface (MACI) implements the *container* part of the component-container model. It is a service that knows about all the Device Objects that together compose the control system and manages their interconnections and lifecycle. MACI has two major elements:

- Activators are the C++ containers. They are deployed locally on all hosts involved in the control system, ranging from real-time local control units to high-performance workstations. Their primary task is preparing the local environment in which DOs (the *components*) are created, giving them all the resources they need to perform their tasks, such as CORBA connectivity, connection establishment with other DOs and Configuration Database access.
- The Manager, which is set up at one central location that is widely known across the entire system. The Manager is acquainted with all the Distributed Objects and Activators in the system, as well as other resources, such as configuration database and CORBA services. In particular, the Manager closely cooperates with the CORBA Naming Service, in which it publishes all of its acquaintances, making them accessible to non-MACI-aware CORBA software.

Within the scope of MACI, a *component* is the *Distributed Object* (DO). It is given a unique, non-volatile identification, called *Component Unique Resource Locator* (CURL). CURL does not directly specify the host on which the DO resides: instead, it serves as a handle through which the designated DO can be accessed.

4.2 Object Explorer

Object Explorer is a GUI application that allows the user to find and view all objects in a system. Every object can be introspected, which is a process through which the object's methods and properties are identified and presented to the user. Furthermore, the user is able to query a property, set a property's value, or to invoke a method on the object, without Object Explorer's compile-time knowledge of the target object.

4.3 Administration Client

Administration Client is a GUI application that allows the user to view the state of the management subsystem. The client is capable of contacting the Manager and the Activators, and querying them for their internal state, which is then presented to the user. The user can also influence the Activator's and Manager's state, for example by instructing the Activator to bring a Device Object off-line, or to restart/switch off the Activator itself.

5 DATA AND HARDWARE ACCESS TIERS

In the three-tier architecture, Data Access Tier is the lower-most layer in a sense that it ultimately handles all requests to read and write data to a physical medium. In control systems, the *reading and writing to a physical medium* should also be understood to cover the accessing and controlling of physical devices (the *hardware-access tier*). The purpose of this tier is to decouple access to the data from the logic that actually handles the data (the middle tier). A well-designed data access tier makes it possible to switch physical sources of data (e.g., SQL databases, devices, device simulations, …) without any changes to the implementation of the tiers built above it.

5.1 Configuration Database

The ACS configuration database has three important parts:

1. The database engine used to store and retrieve data. It may consist of a set of XML files in a hierarchical file system structure or it may be a relational database or another application specific database engine.
2. The Database Access Layer (DAL) that hides the actual database implementation from applications, so that the same interfaces are used to access different database engines. For each database engine a specific DAL CORBA service is implemented. The DAL is defined in terms of CORBA IDL interfaces and applications access data in the form of XML records or CORBA Property Sets.
3. The database clients access data from the database using only the interfaces provided by the DAL. Data clients like Activators, Managers and DOs retrieve their configuration information from the Database using a simple read-only interface. On the other hand, CDB Administration applications are used to configure, maintain and load data in the database using other read-write interfaces provided by the DAL.

We have provided several database engine implementations. The most advanced implementation is based on a set of XML files representing structured records (e.g., configuration information for a complex device). The XML files themselves are hierarchically organized in the file system. Every XML file is assigned an XML schema (XSD), which defines its exact structure, making it possible for an XML file to obtain default values from the schema, as well as model the specialization relationships of the schemas.

5.2 Device Drivers

ACS does not impose any special requirements on the devices and their drivers. So far, the ACS has been made to interact with hardware connected via the CAN bus, as well as LonWorks. We are also considering interacting with EPICS at the level of device drivers.

6 MIDDLE TIER

The middle tier represents the tier that actually *knows* what the application is all about. For example, in a business application this is the tier that would understand the concept of an *order*, and would know that after the client submits an order, the client's charge should be calculated, debited from the client's credit card, and the order delegated to the manufacturing department.

In control systems, the middle tier is the one that knows the business, too – the business being control of hardware devices. For example, if a value of a controlled property

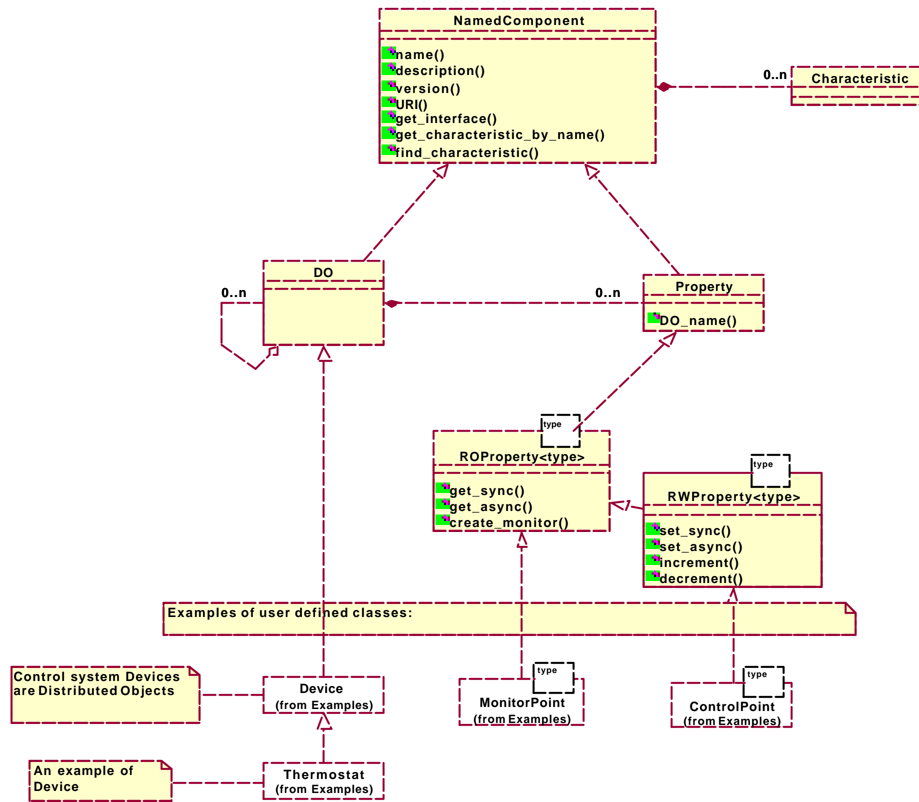

Figure 2: BACI Class Diagram.

changes, the middle tier should be capable of remembering the value in the property's history, and if a certain threshold is reached, raise an appropriate change notification or alarm.

Note that a well-built middle tier never accesses physical sources of data directly; instead, it always uses the data access tier to do the job.

6.1 BACI

BACI defines a set of base classes and interfaces, which are used by Device Object implementations. BACI itself does not define any specific control system. It does, however, restrict the set of all definable objects to a specific set that conforms to BACI design guidelines and uses BACI interfaces in a predefined way.

The heart of BACI is a distributed object model (see the BACI Class Diagram on Figure 2).

In BACI all devices/controlled objects are defined by means of Device Objects (DOs). DOs are implemented as objects that are remotely accessible from any computer through the client-server paradigm.

Each DO is further composed of *Properties*. A *DO* can also contain references to other DOs to build hierarchical structures of components.

DOs and Properties have specific *Characteristics*, e.g. name, unit, and minimum/maximum. The common behavior of DO and Property has been factorized in the *Named Component* common base class.

While there are in principle an infinite number of DO types, for example one for each physical controlled device, there are very few different *Property* types: in principle one for each primitive data type and one for each sequence of primitive data types.

BACI also defines patterns as callbacks (using Asynchronous Completion Token - an object behavioral pattern for efficient asynchronous event handling), on-trigger and on-change monitors and event sets (alarms).

6.2 Error System

The Error System propagates error messages through the ACS, making use of OO technology: exceptions and serialization.

In the ACS Error System, all errors that occur are encapsulated in exception objects. The ACS Error System defines how to and provides means to chain consecutive error conditions that depend upon each other into a linked list of exception objects within a single process.

For inter-process communications such a CORBA method calls or monitors, the ACS Error System serializes the linked list of exceptions and transfers it to the calling process. There, it is deserialized back into a linked list, to which new exceptions can be appended.

6.3 Logging and Archiving