

The ALMA common software: a developer friendly CORBA-based framework

G.Chiozzi^{*a}, B.Jeram^a, H.Sommer^a, A.Caproni^{ae}, M.Plesko^{bc}, M.Sekoranja^b, K.Zagar^c, D.W.Fugate^d,
P.Di Marcantonio^e, R.Cirami^e

^a European Southern Observatory, Karl-Schwarzschildstr. 2, D-85748, Garching, Germany

^b Joseph Stefan Institute, Jamova 39, SI-1000 Ljubljana, Slovenia

^c CosyLAB, Teslova 30, SI-1000 Ljubljana, Slovenia

^d University of Calgary, 2500 University Dr. NW, Calgary, AB, Canada T2N 3Y7

^e INAF-OAT, Osservatorio Astronomico di Trieste, via G.B.Tiepolo 11, I-34131 Trieste Italy

ABSTRACT

The ALMA Common Software (ACS) is a set of application frameworks built on top of CORBA. It provides a common software infrastructure to all partners in the ALMA collaboration. The usage of ACS extends from high-level applications such as the Observation Preparation Tool [7] that will run on the desk of astronomers, down to the Control Software [6] domain. The purpose of ACS is twofold: from a system perspective, it provides the implementation of a coherent set of design patterns and services that will make the whole ALMA software [1] uniform and maintainable; from the perspective of an ALMA developer, it provides a friendly programming environment in which the complexity of the CORBA middleware and other libraries is hidden and coding is drastically reduced. The evolution of ACS is driven by a long term development plan, however on the 6-months release cycle the plan is adjusted based on incoming requests from ALMA subsystem development teams. ACS was presented at SPIE 2002[2]. In the two years since then, the core services provided by ACS have been extended, while the coverage of the application framework has been increased to satisfy the needs of high-level and data flow applications. ACS is available under the LGPL public license. The patterns implemented and the services provided can be of use also outside the astronomical community; several projects have already shown their interest in ACS. This paper presents the status of ACS and the progress over the last two years. Emphasis is placed on showing how requests from ACS users have driven the selection of new features.

Keywords: ALMA, ACS, CORBA, common software, middleware

1. INTRODUCTION

ACS was presented at SPIE 2002[2]. At that time, the only real user of ACS was the team developing the Control System for the ALMA Test Interferometer (TICS) [6]. All other ALMA subsystems were just starting their architecture and design activity at that time.

Since then, there have been 4 more releases of ACS, TICS has been used “in operation” to evaluate the ALMA prototype antennas [6], many subsystems have done substantial development based on ACS and there have already been two global integrations of the ALMA system to verify the soundness of the architecture [1].

During these two years ACS has therefore experienced a fast and substantial evolution, driven by the requirements coming from these users. The initial focus on Control Software related features, motivated by the need to support TICS development, has been gradually replaced by a focus on providing support for higher level software, pipeline and data reduction applications.

This evolution is part of the ACS development strategy, since our team shall be able to implement ACS while at the same time the other ALMA teams have to implement their subsystems. On one hand, a long term development plan defines the goals for ACS, while on the other we rely on a release cycle of 6 months with alternating major and minor releases to quickly respond to the requests from the other subsystems. The content of each ACS release is decided upon in a meeting with all subsystem team leaders at the beginning of each release cycle, allowing us to set priorities in an

* gchiozzi@eso.org; phone: +49-89-32006543

agile way. The major releases also provide an occasion to clean up and harmonize interfaces. For example, with ACS 3.0 we have done a major re-factoring of all ACS interfaces that has substantially streamlined them based on the experience and feedback accumulated in these years.

Close cooperation with the ALMA High Level Analysis group ensures that we have a consistent architecture for ACS and all ALMA software.

It is important to stress that with ACS we do not want to re-implement code or facilities that are already available for reuse. Whenever we have to introduce a new feature in ACS we first look at what is available in the public community. In most cases our work consists of adopting an existing solution, integrate it in ACS and provide wrappers to make it easier to use in our particular application domain or more consistent with the rest of ACS. Sometimes the balance of pros and contra pushes us to develop our own solution.

In this paper we will analyze high level choices done for ACS and some of its more important packages to summarize what was the status in 2002, what has changed and why by now and what are the future directions for the design and development. There are also several other packages (like the ACS Sampling System, described in detail elsewhere [5] in this same conference or the Configuration Database) that are very important for the development and the deployment of applications but that we do not want to analyze in this paper. Other packages will come with the future releases.

If we put all these features together, it might seem that ACS is a very complex and difficult to use framework.

Actually our objective is the opposite: we want to simplify the development of complex applications and the usage of complex tools like CORBA. We reach this objective in three ways:

- Separating technical and functional concerns.
The Container (described in section 2) takes care of many technical aspects of the framework, leaving to the application developer the task of implementing just the functional aspects of the application
- Identifying what is the best way to perform a certain task.
CORBA and the other tools that are used underneath ACS allow reaching the same objective in many different ways. We select the one that we consider the most suitable for our application domain and we document and support it so that everybody can use it easily. This contributes also to uniform and more maintainable code across the whole project.
- Wrapping with convenience APIs.
We hide as much as reasonable of the underlying infrastructure in simple APIs with the purpose of making easier their usage and of providing a smooth and coherent integration between sometimes very different tools that we take as the foundation for ACS.

To put this into evidence, we will dedicate one section of the paper to analyze how a developer proceeds with the implementation, testing and debugging of an application in some typical use cases.

2. COMPONENT CONTAINER MODEL

One of the core elements of ACS is a Component Container Model. Because of its core role, this model is described in detail in other papers being presented at this conference [1][3] and therefore we will not give here any technical details.

The main purpose of the model is to facilitate the separation of concerns [1] in two dimensions:

- Technical and functional development.
The Container, implemented by the ACS team, takes care of the technical concerns of the system and application developers need only to implement the functional aspects in the Components. Therefore the developers of subsystems need to be domain experts and not (or, at least, much less) experts of the technologies used underneath, like CORBA.
- Implementation and deployment/administration.
The deployment of the Components in the system is completely decoupled from the implementation of the Components themselves. This means that Components are unaware of the fact that they are deployed and of where other Components they might use are deployed. In this way it is easier to change the system deployment to improve performance and to make it scale up while it grows.

The original ACS Component/Container model was designed in 2000, when the first commercial implementations were appearing on the market. The CORBA Component Container Model (CCM) was just being discussed and we therefore had to implement our own custom solution.

Since then, the .Net and Enterprise Java Beans (EJB) models have become well established commercial implementations. On the other hand good, high-quality public source implementations of CCM are just appearing and still have major restrictions; the most important one being lack of vendor interoperability and multi-language support.

Having our own, simple implementation is therefore still a good choice because we can easily tailor it to our requirements.

At the same time, with ACS 3.0 we have taken the opportunity of a major release and of the fact many subsystems were going to start real development for a major re-factoring of the model. Rather than striving for backward compatibility with the previous versions, we have decided to re-map as much as possible the interfaces following the terminology and concepts of other mainstream Component/Container models from the world of business applications. In this way we have been able to significantly simplify some parts of the documentation and courseware and reduce the learning effort for developers who have already been exposed to these commercial implementations.

The original ACS Component Container model (described in the Management and Access Control Interface - MACI - section in [2]) was implemented in C++ and adequate for the static structure of a control system. But the constraint of having Components implemented only in C++ was too limiting (it was only possible to implement *clients* in Java and Python). Higher level applications are instead much better implemented in Java and the pipeline team has presented us solid use cases for implementing Components in Python, which is the scripting language of choice for ALMA (see section 3).

We have therefore implemented the Container and all that is needed to support the development of Components in both Java and Python. The Containers manage the lifecycle of Components implemented in all these languages and provide them with a very simple way to access common centralized services such as logging, alarms, error handling, configuration database, archive, object location, and at the same time hiding most CORBA.

The Manager that administers the whole system has been re-implemented in Java for portability and maintenance reasons. For us, this has been a very good demonstration of the maturity of the Java language: since this porting to Java the number of problem reports on the Manager has dropped drastically and we have been able to easily implement new features.

The Manager available in 2002 was only capable of handling Components fully specified in the configuration database. This followed the (quasi-)static model of a control system, where for example the number of Antennas and Receivers is fixed and they are statically configured. The new Manager is capable of handling fully dynamic components to satisfy the requirement of higher level sub-systems for software components whose number and configuration varies according to the usage of the system. An example is a component that calculates FFT transforms in an image pipeline: if we have many users executing pipelines, we may need more instances of this component.

In the next releases we are going to further extend the Manager capabilities in two directions:

- Manager federation.
There is now a single instance of the Manager for the whole system. This is a single point of failure (although a crash of the Manager does not bring down the system, but introduces only a recoverable failure in name resolution) and does not scale well when increasing the number of Components and their geographical distribution. We will implement “Manager federation” allowing several Managers to cooperate and create a virtual namespace on the model of the DNS used to resolve Internet domain names.
- Load balancing.
We will extend the deployment capabilities of the Manager to allow applications to install “load balancing algorithms”. This will allow plugging into the Manager code able to decide dynamically how to deploy Components, for example by choosing the node with the lowest load or the node with the most powerful CPU. This work will be very important for pipeline applications and a first prototype has been already implemented for an application external to ALMA using ACS.

Future versions of ACS will address security and scalability issues for the implementation of systems that are distributed across multiple sites and continents.

In the last few months we have been evaluating CCM implementations and we have found that CORBA 3 Interface Definition Language (IDL) specifications have additional features that are extremely useful. While CORBA 2 IDL only allows specifying the interfaces provided by an object, CORBA 3 IDL allows defining also:

- Used interfaces
- Published events
- Consumed events

This is something that we are now handling in ALMA with text descriptions and we would therefore greatly benefit from formal specifications. We plan therefore to work on extending our Component/Container model to support CORBA 3 IDL specifications. This will be at the same time a further step in harmonizing ACS with CCM.

3. SUPPORT FOR MULTIPLE PROGRAMMING LANGUAGES

Back in 2002 we were supporting development in C++, Java and Python with the following usage in mind:

- C++ for high performance and real time applications in the Control System domain
- Java for higher level and coordination applications, also in the Control System domain, and for GUIs
- Python for prototyping, scripting and in general as a “glue” language.

As already mentioned, a consequence of this usage was strong support for developing Components in C++ and Component base classes were mainly supporting a device model (see BACI section in [2]) tailored to control applications.

But this perspective has partially changed during the last two years.

First of all, C++ is not used only for control applications. Pipeline applications also rely a lot on C and C++ for the algorithmic and numerical functions. This is because:

- A huge amount of high performance Fortran code is available for reuse. It is very easy to integrate Fortran with C/C++ wrappers while it is difficult to integrate it with Java.
- C and C++ can provide much better computation performance than Java and there is by now a considerable amount of legacy C and C++ numerical code available for reuse.

Therefore we have implemented C++ Component base classes that are not integrated with the BACI device model.

At the same time, we have seen that Java Components are necessary for developing high-level applications, but that Java Components can also be used at lower levels, whenever there is no need for real time performance. The last release of ACS contains therefore a first implementation of the BACI model for Java as well.

Pipeline applications can be divided into two levels:

- At the lower there is the implementation of numerical and data analysis algorithms.
- On top of that, the algorithms are connected together to build a pipeline recipe for a specific purpose.

The algorithms are best implemented as C++ Components, very often using legacy Fortran subroutines.

But the higher level pipeline has a much more dynamic development cycle, often implemented by astronomers. Using a scripting language for this seems therefore like the natural choice. At the same time these pipeline glue applications fit naturally in the Component paradigm from the point of view of the higher level user of the data reduction system. This use case triggers the requirement for implementing Components in Python. ACS 3.0 has therefore introduced a Python Container.

We have also seen that Python is convenient for implementing small test and engineering user interfaces. This holds true especially when dealing with GUIs to be used in conjunction with prototype Components where the interfaces are likely to change frequently to begin with. Development time could be lost if a major effort had to be put into propagating changes from Components into GUIs also. Due to this fact, we now leave to developers the choice between Python and Java for this purpose, while for major GUI efforts we still mandate the usage of Java.

Now ACS provides most of the same features for all three languages.

This triggers problems of redundant development efforts. If we really had to make everybody happy, we would have to implement everything in each language, but we do not have sufficient resources for that. Therefore when we go to higher level facilities than the mere Component/Container implementation we try to select carefully the most appropriate language for the feature. We do not want to have three general purpose languages, but to have three languages that together fill all our requirements. In some cases we address this problem by wrapping a C/C++ implementation with 3rd party tools that produce Java and/or Python classes. For example the ACS Time System is implemented in C++ and the SWIG tool is used to generate the Python wrapper.

When designing the support for the same feature in multiple languages we also have to find the right balance between two different options:

- Implement the same APIs in all languages, to allow writing more standard code with all of them
- Implement for each language an API that that is tailored better to the specific characteristics of each language.

There is no one “best choice” in absolute terms. If there is a well established API for that service already available for one language, using or emulating that API has often the easiest learning curve. An example of this is the Logging System. In Java JDK 1.4 there is a standard Java Logging API and therefore the ACS Java Logging relies on this standard.

In other cases, we have decided it was more advantageous to push for the same API across languages, as we have done for the event system [4]

4. ERROR SYSTEM

It is extremely important to have a coherent and complete way of handling error conditions all over the system. For ACS applications this involves handling errors

- in different programming languages:
an error in a C++ Component has to be propagated and understood by a Java or Python Component
- distributed over the network:
an error in a Component in one host has to be propagated over the network to a client component on another host

CORBA allows defining exceptions in IDL and throwing exceptions over the call to IDL operations. This means that a remote call can throw an exception that goes back over the wire to the caller and looks the same as a local exception.

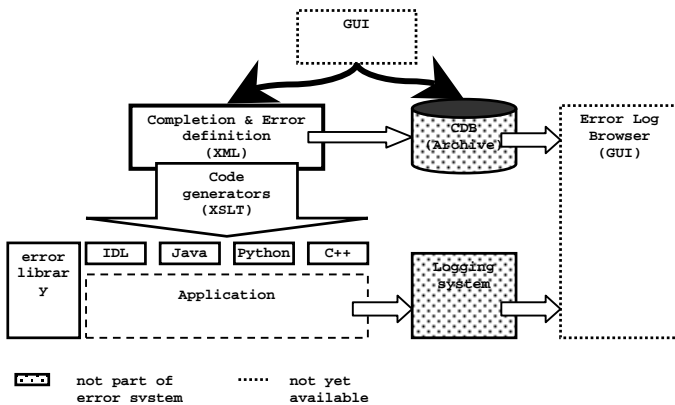


Fig. 1. ACS Error System architecture

The possibility of treating local and remote exceptions in the same way is extremely important in order to build transparency in the distribution of Components, but it is not sufficient. There are many other issues that we need to solve to allow treating efficiently error conditions in Components. The ACS error system (Fig. 1) addresses these problems:

- Exception hierarchies.
CORBA IDL does not allow building hierarchies of exceptions in the object oriented sense, in order to support IDL mapping for non exception-aware programming languages.

ACS uses external error definitions in XML and code generation to provide exception

hierarchies using the native C++, Java and Python exception support, while at the same time generating “flat” IDL exception definitions.

- Error format standardisation.
A part from the exception “name”, we often profit significantly from additional context information in the data coming with the exception. But to be able to interpret this information the data structure shall be standardized

in the format and contents.

ACS error definitions are based on an XML Schema that provides a standardized data structure.

- Error handling design patterns.
The ACS Error System provides the implementation for a number of well proven error handling design patterns [10]. Providing a standard implementation for these patterns helps a lot in writing solid applications.
- Error trace.
In a stand alone application running in a single executable, a low level error is propagated up through the call chain until it reaches somebody that is capable of handling it or until the application is terminated. At each level useful context information can be added. Some languages like Java provide native support for retrieving and manipulating the call chain, but others like C++ do not. This is the Backtrace design pattern and the ACS Error System provides an implementation that works over CORBA network calls.
- Error logging.
With a distributed system the Backtrace pattern allows to trace the chain of errors across distributed components, but the errors traces end up all the times in different places, i.e. where the component that finally handles them resides. It is important to have a centralized place where it is possible to browse and search for errors, with context information allowing identifying where each error occurred. This is done in ACS by sending all error traces to the centralized logging system
- Synchronous and asynchronous error handling.
The exception mechanism works for synchronous calls: the execution of an operation fails, an exception is thrown and it is caught by the caller. But in highly distributed systems many actions have to take place asynchronously: an activity is started by a method call, but the method returns immediately and later on a callback is used to report the result. ACS defines a standardised mechanism to report errors also in such asynchronous situations.

5. XML SERIALISATION

XML is now days a standard way to represent complex data structures. XML has also clear advantages as a means to serialize data and transport it with respect to CORBA structures and object-by-value, as described in more detail [1][3] elsewhere in this conference.

As can be seen in the aforementioned papers, for the Java programming language the Container now integrates transparently the use of type-safe Java binding classes (like a complete Observing Proposal or an Observing Script) to let applications conveniently work with XML transfer objects without having to parse or serialize them.

For the implementation of the Java binding we rely on Java public source libraries; unfortunately there are no comparable libraries for C++ or Python. With these two languages, if data entities are serialized it is necessary to directly parse the XML with a standard XML parser.

While this is not a major problem since most of the code involving complex data entities is going to be developed in Java anyway, transparent binding of XML to classes is very convenient. We are therefore now working on a solution based on code generation of binding classes from a UML representation. This approach is also described in [3] and is already showing promise.

6. INTERFACE TO HARDWARE

As far as interfacing to hardware is concerned, ACS provides a generic abstraction of devices based on the Component/Property/Characteristic design pattern (see BACI in [2]).

A major improvement to this model has been introduced by decoupling the implementation of the Property classes from the actual access to the hardware (Fig. 2).

Property classes no longer implement access to the hardware themselves, but are given at instantiation time an object implementing the simple DevIO abstract interface [9] that will be responsible for that.

The code of the Properties is therefore fully generic and provides to the application synchronous and asynchronous access to the value of monitor and control points, and contains features like telemetry logging and alarms.

To implement access to new hardware, it is now sufficient to implement a new DevIO class, in most cases simply by implementing a read() and/or a write() method that goes down to the hardware level.

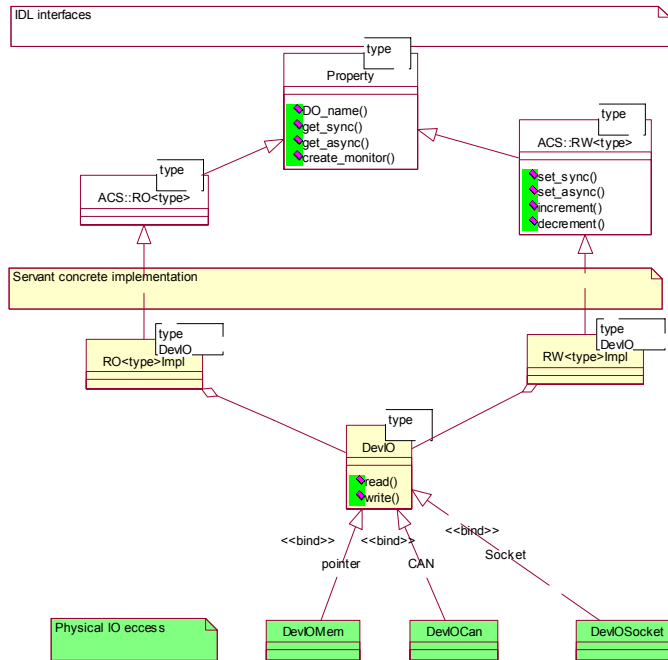


Fig. 2. ACS DevIO architecture

For example, ACS high-level code controlling a Power Supply is only aware of the SetPointCurrent control point and ReadbackCurrent monitor point. If the actual Power Supply is controlled via an analog I/O board, the properties are configured with a DevIO capable of reading and setting the channels of the board. If the Power Supply is replaced by a new one controlled via a serial interface, only the DevIO implementation needs to be replaced.

ACS itself provides a default DevIO implementation, which simply writes to and reads from a memory location. The aim of this memory DevIO is to represent values that are calculated by the software and not directly associated with hardware, like for example the status of a device.

There are other major DevIO implementations developed by groups using ACS: for example, a CAN bus communication is used by ALMA and a socket based implementation is used by the Atacama Pathfinder EXperiment (APEX) project:

- ALMA Antenna Test Facility (ATF):

Controller Area Network (CAN) bus. All devices at the ATF (i.e. the ALMA prototype antennas) are attached to a CAN bus, through which they are controlled and monitored. TICS implements several DevIOs for the different devices used.

- APEX: socket (TCP and UDP). The APEX project uses on one side the TICS software. On the other side it interfaces the system to existing hardware. Those existing embedded systems mostly communicate via socket connections. Thus a DevIO based on a pure socket protocol has been implemented. The socket commands have been standardized using the SCPI syntax.

Other examples of DevIOs that have been implemented include a generic serial device interface, an Heidenhain Encoder board, a motor control unit, a Shack-Hartmann sensing unit and a Web camera.

7. MASTER COMPONENT AND STATE MACHINES

The ALMA Software is composed by many subsystems that have to interact and that have to be administered (started, stopped, checked for health) by a central coordination application, the *Executive*. This is a very common architecture and appears in many other scientific (and industrial) facilities.

The *Executive* does not want to know about the peculiarities of each subsystem and want to be able to treat them all as much as possible in the same way. It is therefore reasonable to define a standard interface that each subsystem has to implement and expose to the administrator.

The most natural type of interface for such purpose is a state machine and therefore we have specified subsystem level state machine and implemented it in a Master Component.

This is a big help in getting a system that is easy to integrate even if the subsystems are developed by completely independent teams, as it is the case for large international collaborations.

Components implementing state machines are very useful also in many other places, but strangely enough we have not been able to find any general state machine implementation in the free source world that we could use for our implementation. Therefore we are implementing in ACS a general solution based on code generation directly from UML models. Using this approach it will be possible to draw a state machine with any commercial UML drawing tool and get the component infrastructure generated automatically.

This implementation relies on the Open ArchitectureWare code generation engine [11]. This tool already include the capability of parsing UML models saved in XMI format and is therefore compatible with models edited with most commercial UML drawing tools. Once the model is parsed it is very easy to generate code, as we do in ALMA also for the data modeling (see section 5.4 in [1] for some details).

8. NOTIFICATION CHANNEL

The ACS notification channel framework provides developers with an easy to use, high-performance, event-driven system supported across multiple programming languages and operating systems. It sits on top of the CORBA notification service and hides nearly all CORBA from developers.

The CORBA notification service is extremely flexible, but this makes it also rather complex to use. The ACS development has concentrated in identifying the most important uses case for ALMA purposes and developing APIs to make the implementation of these use cases as trivial as possible.

This has involved a lot of discussions with the users and a few iterations of re-factoring and has lead to three cornerstone principles:

- Data is published on a channel by *suppliers*. Therefore ACS provides a *SimpleSupplier* abstract base class that takes care of everything from the publisher point of view and requires only implementing the access to the data to be published.
- Data is retrieved asynchronously by clients using a callback interface. Therefore ACS provides a *Consumer* abstract base class and requires only implementing the callback that will “do something useful” when data is received.
- The data that is sent through the Notification Channels is defined in terms of CORBA IDL structures. This ensures that we have a clear interface specification between suppliers and consumers

A detailed description of the ACS Notification Channel is given in [4] in this same conference.

9. A DEVELOPER'S LIFE

In this section we want to analyze with some examples how a developer uses ACS to implement an application, i.e. typically a Component that integrates itself in the system together with other Components, being able to use them and to be used by them.

There is not much use in discussing a “HelloWorld” application and compare it with a C-style equivalent because the added value of a simple ACS Component is not in how simple it is to write a bare message on the screen (and this is without doubt much more complicated than with a simple C program), but in the seamless integration in a distributed and heterogeneous environment together with other Components.

9.1. Designing and implementing the control for an Antenna Mount

9.1.1. Problem definition

We have a simple motorized mount that can be controlled via a serial interface. Something like the mounts for amateur telescopes commercially available. We want to be able to control this mount from an ACS application and integrate it in an already existing system based on ACS. The mount can be positioned in azimuth and elevation coordinates, and we can read the actual position from an encoder by sending a command string on the serial port.

Therefore the developer has to implement a Component that provides:

- Access to the serial port with the specific protocol provided by the vendor of the mount
- An interface to the outside (ACS) world to command and monitor the mount

9.1.2. Interface Definition

The first step is to define the external interface of the mount Component. This interface is the contract with the external users.

Once this interface is defined we can implement clients that use it even before an actual implementation is available. Tests on the clients can be performed using mock-ups or simulators well before the real implementation.

As discussed in [3], interfaces are defined using the CORBA IDL language. Fig. 3 shows a somehow simplified IDL interface for the mount, providing one `move()` method to position the mount and four read only Properties representing the commanded and actual position of the mount. The Mount interface inherits from a base ACS interface that defines basic capabilities every Component is bound to implement.

```
interface Mount : ACS::CharacteristicComponent
{
    void move(in double az, in double elev);
    attribute ACS::ROdouble cmdAz;
    attribute ACS::ROdouble cmdEl;
    attribute ACS::ROdouble actAz;
    attribute ACS::ROdouble actEl;
};
```

Fig. 3. Mount IDL interface

9.1.3. Select the implementation language

In order to implement the component we first of all need to select the best implementation language.

A mock up can be implemented very quickly and easily in Python. But our final implementation will benefit a lot from using all features that are made available by the Component/Property/Characteristic model provided by ACS in C++ (and Java) and by the DevIO serial interfaces already available because implemented by other groups working with ACS. We select therefore C++ as our language.

9.1.4. Write the implementation for the Component

Given the IDL interface, an IDL compiler generates an abstract base class (called *skeleton*) that we have to use as a starting point for the actual implementation.

We have to write a C++ class that inherits from this skeleton and from a base ACS class providing a default implementation for the life cycle interface of the component.

What remains to be implemented is the `move()` method, access methods for the Properties, as well as a few standard methods. Actually most of this code follows a very systematic structure and could be provided by a code generator, as will be discussed later on. The `move()` method would typically have to send the commanded position to the mount via the serial port, but instead of coding this inside the method itself we delegate the problem to the two commanded azimuth and elevation properties. The `move()` method simply writes the value in the properties.

In this way we have written a Mount control implementation that is up to this point independent from the hardware. As we have seen in section 6, access to the hardware is delegated to the implementation of a DevIO class. Initially we can use the provided memory based DevIO and test our implementation without the hardware.

9.1.5. Write the DevIOs for accessing the hardware

Now we can implement our DevIO classes. We will have to implement the specific protocol for the strings used to send the read and write commands to the mount hardware. The DevIO can be tested independently from the Component until we get an implementation that works reliably.

9.1.6. Writing the Configuration Database entries

The Properties of our Mount have units, limits, precision and other parameters that depend on the specific hardware. All this information is stored in the ACS Configuration Database (CDB) [2] as XML files. Therefore we need to prepare the CDB file that contains such information (see also later for more examples of the information contained in the CDB).

We need also to prepare in the CDB the deployment information that the Manager will use at run time to deploy the Component on the proper Container.

9.1.7. What have we got?

These were quite some steps, but what can we do now with this Component?

If we startup ACS, for example using the ACS Command Center GUI, with the appropriate Container we can already do many things. For example:

- Using the Object Explorer we have a generic GUI to access any component in the system. Without writing any client we can interact with our Mount and, for example, call the `move()` method, or monitor and plot the values of the Properties
- All messages produced by the Component are stored in the central Logging System, where can be analyzed.
- If the Properties have been configured in the CDB, their value will be periodically published and stored in the monitor archive.
- If the position limits are reached, alarms are generated and logged.
- We can easily write Python or Java scripts or GUIs to interact with the component, even if it was implemented in C++.
- We can control the Mount remotely from the Web by running the Object Explorer via Web Start

9.2. Designing and implementing a Pipeline

9.2.1. Problem definition

The primary function of the ALMA science pipeline is to automatically calibrate and image ALMA interferometry data and store the reduced data in the ALMA science archive.

We can identify two levels of processing:

- At the higher level, depending on the type of observation and on the parameters, it is necessary to determine what processing steps need to be performed and in which order. This is a very dynamic process and changes a lot with time, with the astronomers better understanding the capabilities of the machine and developing new ways of using it. Astronomers are responsible for developing these processing “recipes”.
- On the lower level, high performance algorithms are used as building blocks for the higher level “recipes”. These algorithms are implemented necessarily with high performance languages and need to be deployed efficiently on machines with the required computational resources.

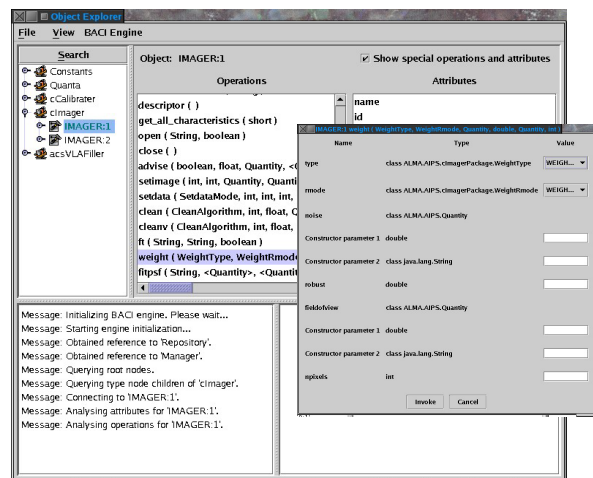


Fig. 4. Using a Pipeline from Object Explorer

9.2.2. Implementation architecture

A good architecture based on ACS would implement the lower level building blocks as C++ components to get the better performance and to allow easy integration of fast and reliable Fortran legacy subroutines.

On the other hand, the high level layer would be implemented as Python components. Python is a very good scripting language, well known to many astronomers and therefore ideal to implement sequences of steps each requesting the services provided by the C++ components or by other Python components with a finer level of granularity. Actually a Python Component will never be aware of the implementation language of another Component it is calling.

9.2.3. Pipeline “recipe” Component implementation

The astronomer/developer of a high-level recipe would have to follow a simpler path compared with the developer of the previous example:

- Define the IDL interface, mostly consisting of the pipeline activation method
- Implement the Python class for the Component, inheriting from the ACS base class. Most of the times the Component’s activation method will execute the sequence of steps in the pipeline by calling other components and passing them the proper data.
- Component activation would be handled dynamically, so CDB entries would probably not be necessary.

9.2.4. What have we got?

This component-based architecture provides us with a number of advantages, on top of what has already been described in the previous example:

- Developers of Components in the two categories are not required to have the same qualification profile.
- The higher level recipe and the algorithms are fully decoupled and they are not aware of the respective implementation. We can initially develop an algorithm in Python or Java and later on replace it with a more performing implementation in Fortran.
- Deployment and scalability issues can be delegated to the Manager at run time. There is no single executable where all code is linked together, but the components are deployed by the Manager, dynamically if necessary. This means that we can deploy together, inside the same Container, Components that need tight interaction; but we can also deploy on a special host a Component with “extreme” resource requirements; or we can re-deploy Components on multiple hosts if there are too many requests for the default host.

10. ACS INSTALLATION ISSUES AND “PURE JAVA” ACS

In 2002 ACS was a monolithic package: the complete package was installed as a single entity including both development and run time environments. Also all development tools such as the C++ compiler or the Java Development Environment were installed at the same time.

This approach has some well-defined advantages:

- It ensures that all machines have the same configuration and tools
- The whole procedure can be as easy as unpacking an archive file to the correct location
- Maintenance and support are easier because we have limited dependencies on components that are external to our distribution.

On the other hand there are also clear disadvantages:

- The distribution is big and uses a lot of disk space. This might be an issue for old systems, embedded machines or for network installations.
- The distribution is not flexible and is more complex to integrate with other applications.

We therefore have the goal of providing a palette of more modular installation options that would allow choosing among:

- Installation of specific versions of tools or reliance on the tools that are part of the operating system distribution or that have been installed by the users.
- Development and run time installation
- Installation of support for selected languages and features

A number of changes have been introduced in all these directions, although we still have more work to do for future releases.

A very important step that is worth mentioning here is enabling "pure Java" development and deployment on any platform where a Java Virtual Machine is available. A Java-ACS package provides the possibility of installing only the Java components anywhere a JVM is available. Otherwise, the complete ACS package provides Java, C++ and Python support on Linux and limited support on other select platforms.

This capability is important to provide a light run-time and development environment for high-level applications, like the Observing Tool, whose first prototype has already been distributed to test users [6]. With ALMA, ACS will reach the desk of astronomers working with the observing preparation tools. Also, all Java GUIs for the control and administration of the system will be able to run on any machine without any specific requirements. At the same time, Java-only ACS is integrated with the entire system (an example is the capability of accessing the central logging system) so no needed functionality is lost.

To make deployment easy, we use Java Web Start [8] technology which allows installing and updating the whole Java ACS and ALMA applications like the Observing Tool simply by accessing a web site and without interfering with the normal environment of the machine where the installation is done, since Web Start applications run essentially in a sand box.

11. FUTURE DIRECTIONS OF DEVELOPMENT

In these two years ACS has been extended in many directions, while it has at the same time reached maturity and reliability in many areas.

But there are still many requirements that need to be addressed.

We are currently working on the planning for the next major release (ACS 4.0, due for the end of 2004).

A part from extensions and change requests concerning the existing packages, many already described in the previous sections, we will work on the following main areas:

- Bulk Data Transfer.
An important category of applications addresses the transfer of large amounts of data as images or as continuous streams. ACS will provide a solution based on the CORBA Audio Video Service. This specification defines standard CORBA services to establish and administer multimedia data streams and techniques to transfer the actual data not only using CORBA as the transport protocol but also using out of band techniques. For example, it is possible to use socket interfaces or other high performance streams to bypass the CORBA transport and optimize transfer rates.
We have now defined the use cases we want to support and we have implemented the first prototypes. We plan to release soon a first implementation to be used by some ALMA subsystems that urgently need this support. We will also make extensive performance tests.
- Alarm System.
The Alarm System delivered with the first releases of ACS was just a prototype or the basic alarm generation mechanism. Our requirements call for a more powerful system and in particular for a flexible "alarm reduction" system capable of generating synthetic views or cascade alarms. We are currently working on a prototype of the new system based on the Laser Project [14] at CERN. This project is very interesting because it has very similar requirements and a similar architecture and therefore we should be able to reuse most of the design and probably also sizable parts of the code for ACS.
- IDL Interface Simulator.
For testing purposes, it is very convenient to have automatically generated simulators of Components, based on the IDL interface. We have demonstrated with a feasibility prototype that it is possible to implement

automatically Python Components for any given Component IDL interface using the introspection mechanisms that CORBA makes available through the Interface Repository service and the Dynamic Invocation APIs.

In a first approximation, such components will implement a default behavior for each method, but augmenting them with Python scripts it is easy to implement special test behavior like reporting error conditions, returning specific values in the out parameters or waiting a specified amount of time before returning.

- Code Generation.

As already described in the previous sections, code generation from UML models based on the Open ArchitectureWare project [11] is providing very promising results in the area of ALMA Data Modeling and state machines. We think that the same approach can be very well applied to Component code generation from a UML specification of the interfaces. An approach to ACS Component generation from IDL has been successfully demonstrated in the APEX project using a custom code generation written in Perl.

We also need to systematically assess ACS/CORBA performance. Until now we have mostly qualitative measures of the performance we can expect in various areas and we have assumed that data published by other projects apply to our case as well. This is a good first approximation, but we will now also make specific quantitative measures based on the ALMA requirements. This work has already been started.

12. ACS USER BASE

ACS is used as the common middleware for the whole ALMA software development. These account for many ACS installations at the sites of all ALMA partners in America, Europe and Japan.

But ACS, although developed with support for ALMA as the main objective, can be used as a general software infrastructure for other scientific facilities and not only in the astronomical community.

For this reason, other projects are collaborating with ACS, already using or evaluating it, thanks also to the fact that ACS is publicly available under the Lesser GNU Public License (LGPL)[12], which allows its free dissemination and use that is unrestricted for all practical purposes.

For example, the ANKA Synchrotron in Karlsruhe is in scientific production, the APEX radio-telescope in Chile is being commissioned and the 1.5m Hexapod Telescope in Bochum is in an advanced implementation stage and will be soon moved to Chile. ESO and NRAO are evaluating ACS for the upgrade of already existing systems and for new development, as well as other external institutions. A consortium of research institutes from the astronomical and accelerator communities along with industrial partners are evaluating the possibility of developing an ACS for embedded systems (eACS) to extend ACS with specific support for low-priced, mass-produced embedded controllers.

In March 2004 we had the first ACS Workshop in Garching, with the participation from many projects.

The fact that ACS is publicly available and other projects have shown interest or are already using it is a big advantage. This is because we get independent, unbiased feedback, and the framework becomes naturally more general. The spin off of collaborations allows us to implement features for which we would not have the resources. ACS is also commercially supported by Cosylab[13], which offers training, custom development and deployment of ACS-based systems, as well as integration of ACS with other systems (e.g., EPICS and legacy systems).

At the same time, as ACS development team, we pay attention to focus as much as possible ACS development on our “core business” (i.e., satisfying the requirements of the ALMA Project). This avoids the mistake of developing a framework that is too general and while trying to satisfy everybody does not make anyone really happy.

13. CONCLUSION

The development of ACS since SPIE 2002 has been very fast. ALMA requirements have driven the development in the direction of Java and Python to support high-level software beyond the domain of Control Software. This promotes seamless development of the software for a scientific facility.

While this high-level software is being developed, we have functioning Control Systems running based on ACS so that its performance and reliability can be verified and improved upon where necessary.

The fact that ACS is publicly available and other projects have shown interest or are already using it is a big advantage. This is because we get independent, unbiased feedback, and the framework becomes naturally more general.

At the same time, we focus as much as possible on our “core business” (i.e., satisfying the requirements of the ALMA Project). This avoids the mistake of developing a framework that is too general; trying to satisfy everybody does not make anyone really happy.

For more details on ACS and for the online documentation and tutorials you can look at the ACS home page [15].

ACKNOWLEDGEMENTS

The ACS project is managed by ESO in collaboration with JSI and INAF-AOT and with contributions from many other ALMA sites. The ACS development team is actually distributed across many institutes involved in ALMA development. This work is the result of many hours of discussion, test and development inside our groups and in the various ALMA centers. We thank all our colleagues in ALMA and in other projects using ACS for their important contributions to the definition and implementation of ACS.

REFERENCES

1. J.Schwarz, A.Farris, H.Sommer, “The ALMA Software Architecture”, these proceedings.
2. G.Chiozzi et al., “CORBA-based Common Software for the ALMA project, Proc. SPIE Vol.4848, Advanced Telescope and Instrumentation Control Software II, pp.43-54, August 2002.
3. H.Sommer, G.Chiozzi, “Container-component model and XML in ALMA ACS”, these proceedings.
4. D.W.Fugate, “A CORBA event system for ALMA common software”, these proceedings.
5. P.Di Marcantonio, R.Cirami, G.Chiozzi, “ACS sampling system: design, implementation and performance evaluation”, these proceedings.
6. R.G.Marson et al., “ALMA test interferometer control system: past experiences and future developments”, these proceeding
7. A.Bridger et al. “Proposal and Observing preparation for ALMA, Proc. SPIE Vol.5493, Optimizing Scientific Return for Astronomy through Information Technologies, June 2004.
8. Java Web Start home page: <http://java.sun.com/products/javawebstart/>
9. B.Jeram et al., “Generic Abstraction of Hardware Control Based on the ALMA Common Software”, ADASS 2003, Strasbourg, France, October 2003
10. K.Renzel, *Error Handling for Business Information Systems*, ARCUS, sd&m Muenchen, Germany, 2003
11. Open ArchitectureWare project home page: <http://sourceforge.net/projects/architekturware/>
12. LGPL, Free Software Foundation, <http://www.fsf.org/licenses/lgpl.txt>
13. Cosylab home page, <http://www.cosylab.com>
14. CERN Laser project home page: <http://proj-laser.web.cern.ch/proj-laser/>
15. ACS Home Page: <http://www.eso.org/projects/alma/develop/acs/>