THE ALMA COMMON SOFTWARE, ACS STATUS AND DEVELOPMENTS

G.Chiozzi[1], A.Caproni[1], R.Cirami[4], P.Di Marcantonio[4], D.Fugate[2], S.Harrington[5], B.Jeram[1], M.Plesko[3], M.Sekoranja[3], H.Sommer[1], K.Zagar[3]

[1]European Southern Observatory, Garching, Germany

[2]University of Calgary, Calgary, Alberta, Canada

[3]Cosylab, Ljubljiana , Slovenia

[4]INAF Osservatorio Astronomico di Trieste, Trieste, Italy

[5]National Radio Astronomy Observatory, Socorro, New Mexico, USA

ABSTRACT

The ALMA Common Software (ACS) is a software infrastructure for the development of distributed systems based on the Component/Container paradigm. ACS is being developed primarily for the ALMA collaboration to provide a common and unifying infrastructure used by all partners and across all layers of the system [5]. The usage of ACS extends from high-level applications such as the Observation Preparation Tool that will run on the desk of astronomers, down to the Control Software domain. From a system perspective, ACS provides the implementation of a set of design patterns and services that make the whole ALMA software 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. ACS was presented at ICALEPCS for the first time in 2001 and then in 2003 [1][2]. Since then, the services provided by ACS have been extended and made reliable and scalable. The control system for the ALMA prototype antennas is based on ACS and is routinely used for the technical and scientific evaluation of the antennas [6]. Thanks to the fact that ACS is available under public LGPL licence, the community of users outside ALMA is growing. In particular the control systems of several telescopes are being developed on top of ACS, with the different teams sharing and reusing design concepts and actual software Components. The ACS community has met at a workshop hosted by this conference. This paper presents the status of ACS and the progress over the last two years. Emphasis is placed on describing ACS from the point of view of control system development. More details on specific ACS services and on projects using ACS will be provided by other papers.

INTRODUCTION

ACS has been under development for 6 years. By now, the core concepts have become very stable and have been presented at previous editions of this conference [1] and at other conferences [3].

These are the main characteristics of ACS:

- ACS provides the basic services needed for object oriented distributed computing. Among these:
    - Transparent remote object invocation
    - Object deployment and location based on a container/component model
    - Distributed error and alarm handling
    - Distributed logging
    - Distributed events
- The ACS framework is based on CORBA and built on top of free CORBA implementations.
- Free software is extensively used whenever available, to avoid “re-inventing the wheel”.
- ACS primary platform is Red-Hat Enterprise, but it works and is used also on other Linux variants (Solaris support was recently abandoned because not used).
- Real time development is supported on Real Time Linux and VxWorks.

- Development is supported in C++, Java and Python. Any other language with a CORBA mapping can be used, if needed.

The total effort allocated to ACS development by the ALMA project is of the order of 25 man years, but the project can count also on additional external contributions**.**

In this paper we will discuss the evolution of ACS with respect to these papers, taken as reference material.

In the last couple of years, ACS has been extensively used “in the field”.

The ALMA prototype antennas are running a control system based on ACS [6] and they are used routinely for the evaluation of the antenna performance and for testing the ALMA architecture, hardware and software.

The complete ALMA software is integrated and tested as a whole on a periodic basis, with two official releases per year, by the Integration and Testing team. This provides very good testing and feedback of the ACS global infrastructure, performance and tools from the operational and deployment point of view.

ACS-based software is also used in engineering laboratories by the teams developing hardware and devices for ALMA, as the basic software infrastructure. This provides feedback from users outside the community of ALMA software developers.

The development of ACS is driven by two forces:

- The long term development plan, which defines the major ACS features and the timescale for their availability.
- The release planning activity (twice a year) where the feedback and requests from all ACS users are collected and discussed to prepare the detailed planning for the next release.

Based on this input, the work of the last two years has followed essentially three main paths:

- Implementation of new features foreseen in the development plan or requested as new requirements by ALMA subsystems. This has covered mainly areas in the high-level and data flow software.
- Cleanup or improvements in the design of core ACS packages. These changes have been driven by feedback from the users either in terms of specific requests or as the result of our analysis of support requests. A major role here has been played by the Integration and Testing activity.
- Improvement of reliability, stability, scalability and performance. These aspects are becoming more and more important as we approach a more “operational” environment, even though we are still far from having the ALMA interferometer running at the Chilean site.

Important feedback comes also from the growing community of ACS users outside of ALMA, in particular in terms of bug reports and suggestions for design improvements.

COMPONENT CONTAINER MODEL

The ACS architecture is based on the implementation of a Container-Component model [4]. The Containers provide the infrastructure for the applications and the developers only need to concentrate their efforts on developing the domain specific code in their Components.

This enforces a uniform structure in all subsystems.

Master component

The ALMA Software, like many experimental facilities, is composed of subsystems that have to interact and that have to be administered (started, stopped, checked for health) by a central coordination application, the *Executive*. It is crucial for the Executive to be able to treat all subsystems as much as possible in the same way, to avoid having to code a lot of domain specific intelligence in the Executive itself. This can be achieved by defining a standard state machine that each subsystem has to implement and expose to the administrator. We have therefore specified a 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 is the case for large international collaborations.

The introduction of the Master Component has been very effective and has made much simpler the system integration, forcing many subsystems to restructure, in a standardized way, their startup and shutdown procedures and also think about how it's possible to monitor their own health.

The state machine, initially very simple, has been refined in a couple of design iterations to take into account the need to handle interdependencies between subsystems. It is shown in Figure **1**.

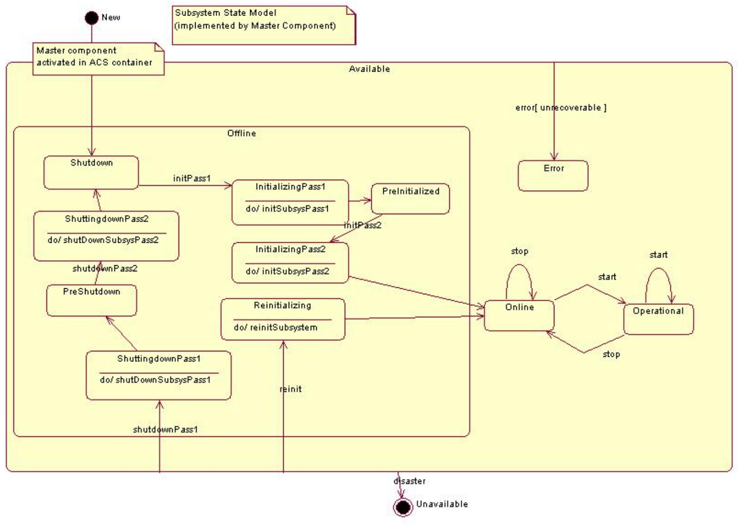

Figure 1: Master Component state machine

This state machine Component has not been implemented by hard-coding state logic. We have instead developed a prototype state machine Component generator from UML state machines. This implementation relies on the Open ArchitectureWare code generation engine that is used in other parts of the ALMA software [12] and in particular to build the ALMA data model. This tool already includes the capability of parsing UML models saved in XMI format and is therefore compatible with models edited with most commercial UML drawing tools. The cost of implementing this prototype of generator has not been higher than the cost of developing the Master Component in a traditional way and we believe that this approach to state machine development is very efficient. The prototype satisfies our current development needs inside the ACS team and we are ready to make it a reliable tool for general usage when we will have requests from our user base.

Container Services

The Containers provide to Components an infrastructure and an environment to live in[4]. Whenever a Component needs to access the external world or needs services like logging, threading or naming resolution, the Container provides them through an interface called ContainerServices.

In the last couple of ACS releases we have significantly improved the encapsulation of the services by better defining the ContainerServices interface and making it almost identical in the three programming languages officially supported by ACS (C++, Java and Python).

Before, particularly for C++, there were various situations where the Container and CORBA were not properly hidden to Components, while now the ContainerServices provide a good abstraction of the services that a Container should provide and CORBA is hidden as much as is reasonable.

In this way it is in principle possible to replace our Container implementation with another, also based on completely different technology, by implementing an adequate ContainerServices and base Component classes, without touching the component's application code. This was requested by the Offline data reduction subsystem, with the perspective of implementing data reduction components to be used outside of ALMA, on systems not based on ACS but on other component/container technologies.

The most important services provided now by the ContainerServices interface are:

- Access to other Components
- Access to the logging system
- Access to the Configuration Database
- Component state management and life cycle (described hereafter)
- Component threading service (described hereafter)

Component Lifecycle

Initially the Container was simply instantiating Components upon request of the Manager [4] when needed (for example when requested by other components) and destroying them when not needed any more.

Experience with the Components developed by the ALMA subsystems has shown this was not sufficient to properly handle runtime dependencies between multiple Components, access to hardware and software resources and error conditions.

We have therefore defined a simple lifecycle interface that each Component has to implement. The standard lifecycle methods (initialize, execute, cleanup, aboutToAbort) are called by the Container according to a lifecycle state machine that is used by the Container to keep track of the health and activation conditions of each Component.

Components have access to their own life cycle state machine through the ContainerServices interface.

EVENT HANDLING AND NOTIFICATION CHANNEL

ACS provides an event system based on the publisher/subscriber paradigm and implemented on top of the CORBA Notification Service [13].

The ALMA software makes very extensive usage of this system, actually well above the original expectations at ACS inception time. The designers of ALMA subsystems find it very convenient to decouple the subsystems by publishing and receiving events instead of using direct interfaces.

The event system is used to:

- Synchronize the activity of subsystems by means of the publication of synchronisation events
- Publish data to be retrieved by one or many subscribers, not known a priori. This mechanism has been preferred in many cases to the callback registration design that is also available with ACS.

In the last years the design of the ACS Notification Channel classes has been drastically improved to make the usage of the system trivial and to shield completely the users from CORBA.

Essentially now it is sufficient to define in the IDL modules the names of the channels to use and the data structures to be transported. Publishers and subscribers can then handle the data with a few lines of code. The configuration of the Quality of Service and Administrative properties for the channel can be done through the configuration database.

This extensive usage of the Notification Channel has caused some problems at integration time at the level of the interfaces between subsystems.

If it is true that the publisher and subscriber are very well decoupled, it is also true that it is much more difficult to spot misalignments due to changes in names or data structures, since there is no direct invocation of IDL interfaces. Therefore it often happened that changes introduced by one subsystem in the channel names used to publish data and in the structure itself of the data published were spotted only at integration time. In some cases, expected events were never published or published events had no subscriber waiting for them.

To overcome these problems we have worked in two directions:

- Defined strong coding and naming conventions for the definition of notification channels in IDL files
- Developed checking and debugging tools

The Notification Channel classes have been modified to make it easier to handle events following the IDL conventions, and to make it more difficult to publish and receive arbitrary data on arbitrary channels. In this way application developers are pushed “to follow the rules” and to use special features only with full awareness.

Checking tools have been developed by the integration team to analyse the code and check consistency with the help of the naming conventions. In the future we will make use of CORBA IDL 3 event specification notation, which has been introduced with the purpose of addressing these issues.

At the same time we have implemented an Event Browser GUI (seeFigure **2**: ACS Event Channel Browser) which allows run-time inspection of all the activity in the Notification Channels to allow operators and integrators to spot anomalies in events published and in the network of publishers and subscribers.

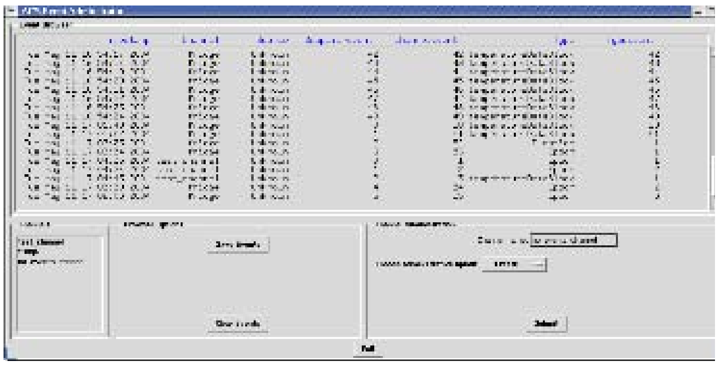

Figure 2: ACS Event Channel Browser

THREADING SUPPORT

Many Components, in particular in the area of the Control Software, have a multithreading structure. This means that there are threads of execution, like control or monitoring loops, that are intrinsically associated with the Component, i.e. are started when the Component is initialised and stopped when the Component is taken down.

We have seen that the management of such threading Components was a source of problems in the application code, with threads left hanging after Component destructions and other misbehaviour.

We have therefore decided to provide support for well behaving thread design patterns, in particular for C++ and Java.

Each Component now has an associated pool of threads. The ContainerServices provides Components with a Thread Manager object that can be used to get hold of Component-specific threads. When the Components are destroyed, the Thread Manager makes sure that all threads are properly terminated.

In C++ we have built threading classes based on top of the very good APIs provided by the ACE framework [14]. Sub-classing and overriding one method is sufficient to have a thread function executed once (in order to have one-shot asynchronous action) or in a repeated loop (as in the implementation of a control loop). Complete management of the thread (start, stop, resume, etc) is possible.

In Java, where a good support for threading is available natively, the Thread Manager relies on the Java classes provided by the *concurrent* library. Also in the case for Python we rely on the native libraries.

REAL TIME SUPPORT

In the last two years the ALMA project has carried on a major paradigm shift in the implementation of real time software.

The real time part of the control system for the Test Interferometer was implemented using VxWorks. This implied the usage of Local Control Units running the VxWorks operating system. A complete implementation of ACS Containers in VxWorks was necessary to support the deployment of Components on the LCUs.

Now we use Real Time Linux (in the RTAI flavour[17]) and the new implementation of the control system based on RTAI is being deployed on the test antennas in these months. The decision to change the real time operating system has been the result of an evaluation of the two alternatives in terms of technical advantages and disadvantages, cost and future perspectives.

RTAI puts a real time kernel inside a normal Linux PC. Most of the code runs in a standard non-real time environment. Only the time critical parts run in the real time kernel.

With the previous architecture, the code of entire components was developed and deployed in the real time operating system.

With the new architecture most of the code is “standard Linux code” and only tiny parts are developed as real time kernel modules. A very critical part is then the communication between real time and non real time code.

This requires a major change in the architecture of applications, because real time and non-real time code need to be clearly separated and their communication needs to be very well thought out, otherwise it will introduce performance problems and possibly disrupt the real time behaviour. Using Real Time Linux, the code running in kernel space must be as small as possible, because it is much more difficult to develop, test and debug than the “normal” code (and also with respect to VxWorks code); code must be written in plain C and only very limited libraries are available.

The transition has been longer and more complex than originally foreseen because we were not facing a simple porting but really a re-design, with a completely different allocation of responsibilities between the various parts of the code. The two approaches have very different advantages and disadvantages.

On the ACS side this means that we do not need to provide a full Container implementation for a new real time operating system, since the container will in any case run on the non-real time side of Linux. We have instead provided small support libraries for the communication and tools to manage kernel modules. We have also provided convenient makefile and build support, integrated with the rest of the development environment. All this work has been done in strict collaboration with the Control and Correlator teams, mainly in the form of contributions to ACS consisting of code initially developed directly by them.

Once the phasing out of VxWorks from ALMA is complete, we will probably maintain VxWorks just for and with contributions of other projects outside ALMA using ACS in that environment.

BULK DATA TRANSFER

ALMA has very strong requirements for the amount of data that needs to be transported by software communication channels, in particular from the correlator to the archive (raw data from the antennas is luckily enough not under software responsibility).

A major development of the last year has been the bulk data system, devoted to the transport of huge amounts of data and based on the CORBA Audio/Video streaming service specification.

The bulk data system is described in details in [9] in this conference.

We have implemented very easy to use classes on top of the A/V streaming that implement the use cases we have identified for ALMA shielding completely CORBA and the details of the A/V itself.

Using this system we avoid the performance penalty introduced by the CORBA communication protocol, transmitting data outbound directly in TCP or UDP format. On the other hand, we still use a well defined and standardized protocol for the handshaking and administration saving the effort of designing and implementing our own proprietary solution.

Unfortunately the only implementation we have available is the TAO C++ implementation. For the time being we do not have strong requirements to have the bulk data transfer available in Java or Python. We think anyway that it would be a reasonable effort to port to Java the basic components that would be needed to have our use cases working.

IDL SIMULATOR

ALMA is composed of many subsystems, each providing several Components. The development is distributed across many sites and proceeds in parallel on all subsystems.

As a consequence, when testing or integrating multiple subsystems or Components not all needed features are available at all times. It often happens that entire Components are not available at all.

We therefore have the need of simulating components or the features that are missing.

Another important development of the last year has been a powerful interface simulator.

This simulator, described in detail in [7] in this conference, allows simulating entire Components starting from their IDL formal interface specification. The simulator is capable of providing default behaviour for all methods and attributes, but the user can implement “intelligent simulation” using configuration files and a runtime API.

The simulator leverages standard CORBA features like the Interface Repository and the dynamic characteristics of the Python interpreted language to make it possible to discover and change at run time how interfaces behave in simulation.

ALARM SYSTEM

The Alarm System is a very important part of any Control System. For ALMA we have standard requirements, common to many other scientific experiments, including the need for reduction of alarms to reduce the flooding of events and make it easier for operators to understand the real root of the problem.

During ICALEPCS 2003 we were starting to look into the design of the ACS Alarm System, when at the conference we saw the LASER system, under development at CERN for the LHC accelerator[15].

This system has requirements very similar (actually, more stringent) to ours and uses concepts compatible with the architecture of ACS.

We have therefore started an evaluation project with the aim of using the LASER system within ACS. This requires bridging the technologies used in the LASER system (for example the Enterprise Java Beans and Java Messaging System) with what is used in ACS (for example Components and CORBA Notification Channels) with minimal effort so that it is possible to maintain a common code base.

This project has been carried on until few months ago with low priority, but in the last months the priority has gone up and we are now in the conclusive phases of the evaluation, being able to handle the complete chain of alarm propagation in ACS with a customised LASER system.

We have encountered difficulties because the technologies used in LASER and ACS are apparently very similar but, in practice, the details make it quite difficult to keep the code working with both.

But we now expect to put in place an active collaboration so that the customised LASER system can be integrated in ACS and used for ALMA and other projects, while providing valuable feedback and contributions to the LASER team.

TASKS AND PARAMETERS

A *task* is a program which starts up, performs some processing, and then shuts down. A task may or may not require various ACS services, depending on the context.

Tasks have been introduced in ACS in the last year, as a requirement for certain types of offline data processing[18]. In these use cases a simple stand-alone executable is required, while asking the user to perform a complete ACS startup (services, containers, manager, configuration database) is not desirable. In these situations, running a *task* should involve simply typing one command, possibly with some input data, at the command prompt.

The *task* mechanism (currently implemented by ACS in C++) is able to run either with or without ACS services:

- A special "*static container*" allows the functionality of the ACS container to be linked directly with the task executable at compile/link time (normal ACS Components are dynamically loaded upon request).
- The static container allows a task to function as a valid Component in the ACS Component-Container model while not requiring a complete ACS startup.
- In the event that ACS services are available, the task Component, using the linked-in static Container, can operate as a full-fledged ACS Component.

Input *parameters* can be complex data sets, with mandatory and/or optional elements.

ACS provides:

- a mechanism for the task *developer* to define a task’s expected parameters and other meta-data via XML, including strong parameter validation rules
- a mechanism for the task *user* to define the values of those parameters (for a particular task execution) either via XML or as arguments on the command line
- a mechanism to parse and validate the parameters provided at run time by the task user
- an API allowing the task developer to query an in-memory representation of the parameters provided by the task user.

This enables new tasks to be added rapidly and with a minimal programming effort.

While developed explicitly for situations related to offline data processing, we see now new applications for these concepts, in particular for the implementation of small clients used from the command line.

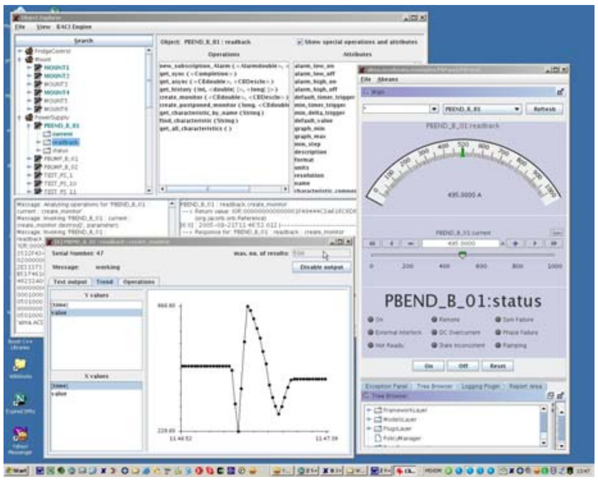

Figure 3: ACS tools and user interfaces

GRAPHICAL USER INTERFACES

ACS supports the development of GUIs in Java providing the ABeans libraries[16]. In the last ACS release we have completed the switch to Eclipse as a Java development environment by integrating ACS GUI development with ABeans in the Eclipse Visual Editor and providing a small plug-in for this purpose. Before, interactive GUI development was supported under NetBeans.

A number of GUIs have been developed in Java text editing or interactive mode within Eclipse, like for example the antenna mount GUI or the ACS command center, showing that Eclipse is a very powerful and convenient development platform.

Other engineering user interfaces have been implemented in Python.

Some non-ALMA projects are also interfacing ACS to GUIs developed in C++ using the Qt libraries. Even prototypes with LabView interfacing ACS Components have been implemented.

This means that, while we strongly support GUI development in Java, in particular with the ABeans framework, other solutions are possible and are being exploited by different teams because ACS can be interfaced with any code written in Java, C++ and Python. The main reason for selecting something different from Java is previous experience with other languages or libraries, but this can speed up significantly GUI implementation for expert developers.

BENCHMARKING AND BENCHMARKING TOOLS

ACS has to satisfy strong performance requirements to allow ALMA and the other projects to run properly. More over, changes to the code and the upgrade of external libraries can have heavy and often unexpected impact on performance.

Now that ACS is in an advanced development stage and we have already operational environments it is essential to have means to verify the performance and to keep track of how it changes over time. We have realized that it is important to have a set of standard performance tests that are repeated in similar conditions at each release and compared with historical records.

The ACS Performance Measurement framework was created as a means to determine the performance limitations of various ACS APIs. This includes very low-level tests such as determining how many method invocations can be invoked on a component per second to more abstract tests such as determining how long it takes to start the core of ACS.

The framework has been created in such a way that not only is it useful to ACS, but it can be used by other ALMA software subsystems to build their own benchmarks.

It consists of:

- “Profiler” objects in C++, Java, and Python. These profilers are essentially stopwatches that obtain interesting data about a particular block of code.
- A wrapper script which profiles entire executables.
- A report generator which turns the raw output from the profilers into human-readable HTML reports (http://www.eso.org/~almamgr/AlmaAcs/Performance/BenchmarkDoc/).

We have created and maintained benchmarking tests for the Component to Component communication, Bulk Data Transfer API, Notification Channel Publishers, the Logging System and Makefile performance just to name a few.

FUTURE DIRECTIONS OF DEVELOPMENT

The core of ACS is by now very stable, both in terms of design and implementation.

Most packages are available, but not all features foreseen in the architecture have been implemented. Our main objective is to provide at each release what is needed by the subsystem development teams and our release planning is the result of a trade-off between urgent requirements and the need to “fill the holes”.

Clearly, for many of the features not jet implemented the time of need is coming. Therefore with the next releases we plan to work on them.

Many are in areas related to performance and scalability, since ALMA will have to go from the 2 antennas of the test interferometer to the 64 of the final system and from the one operation site of the tests to a very distributed network with the mountain site, the operation center and regional data centers. For example, we have to work on some aspects of federation of ACS domains (already available as prototype) that have to allow scaling up ACS applications to systems running reliably across several sites and continents.

A lot of input is also coming now from the ALMA Integration and Testing activity. From them we are getting requests and feedback in the areas of deployment, administration and debugging tools and facilities.

Looking in another direction, we want to make nicer and easier the work of developers, providing more abstract ways to implement the architecture and the code of the Components. Together with the ALMA High Level Analysis team we think that code generation from UML will be able to relieve the programmers from a lot of code editing, since a big part of the Component’s code can be easily generated. Actually other projects using ACS have successfully used Component’s code generation from CORBA IDL in the context of their specific system. We follow these developments with great interest.

THE ACS USERS COMMUNITY

ACS is publicly available under LGPL licence, because we think that a wide community of users can provide excellent feedback and help us to have a solid system for ALMA.

There are a number of projects that have decided to base their system on ACS, like the Atacama Pathfinder Experiment, the Spanish OAN 40 meters radio telescope[8], the Sardinian Radio Telescope in Italy, the Hexapod Telescope in Chile or the ANKA Synchrotron in Germany[10]. Other projects are evaluating this possibility.

The community is very active. The Component/Container model allows different projects to share Components and we have created code sharing areas. The possibility of sharing solutions is a major driver in choosing ACS, in particular for projects in the same domain.

Discussions take place on a mailing list and we have periodic phone meetings and workshops.

CONCLUSION

ACS is at half of its development life and it has been used extensively for the development of ALMA and of other projects. We can therefore start evaluating the benefits and the drawbacks.

ALMA is a highly distributed development and a pot of software cultures, with more than 20 development sites in 4 continents. Using a common software framework is essential to create a coherent system. This increases maintainability and facilitates integration and testing activities.

On the other hand, this means imposing a way of working, technologies and tools that might be alien to development teams with a different culture and engineering tradition.

The teams involved in “global activities” (like high level analysis or integration) see immediate advantages. Smaller subsystem teams feel sometimes more the constrictions in their freedom of design than the advantages, if they were not used to this development process from previous projects. We have to put high priority in demonstrating also to these groups the added value they get, trying to make the overhead as small as possible and listening to the feedback. This is why the planning of each ACS release is discussed and agreed with all ALMA subsystems and based on their requests.

For project management this is an investment for the future, when the whole software will be handed over to the support team. Already now it is a big advantage that during the system tests the integration team can look into and understand the software of all subsystems.

For other projects, in particular if small and with limited resources, adopting ACS provides a bit start jump and the learning curve is compensated by using a solid system and having access to a wide and experienced community, willing to share solutions.

In this paper we wanted to give a flavour of the recent development of ACS, trying to describe the reasons and the different forces driving the technical choices. The interested reader will find architecture, design and reference documentation in the ACS Web pages[11].

ACKNOWLEDGEMENTS

The ACS project is managed by ESO in collaboration with NRAO, University of Calgary, Cosylab and INAF-AOT. 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] G.Chiozzi et. al, “The ALMA Common Software (ACS): status and developments”, ICALEPCS’2003, Gyeongiu, Korea, October 2003.
[2] K.Zagar et. al, “ACS – Overview of technical features”, ICALEPCS’2003, Gyeongiu, Korea, October 2003.
[3] G.Chiozzi et al., “The ALMA Common Software: a developer friendly CORBA based framework“ SPIE 2004 - Astronomical Telescopes and Instrumentation Glasgow, Scotland, UK, June 2004, paper 5496-23
[4] H.Sommer et al., “Container-component model and XML in ALMA ACS”, SPIE 2004 - Astronomical Telescopes and Instrumentation, Glasgow, Scotland, June 2004, paper 5496-24
[5] B.E. Glendenning, G. Raffi , “The ALMA Computing Project – Update and Management Approach”, ICALEPCS’2005, Geneva, Switzerland, October 2005
[6] A. Farris et. al., “The ALMA Telescope Control System” , ICALEPCS’2005, Geneva, Switzerland, October 2005
[7] D. Fugate et al., “A generic software interface simulator for ALMA common software” , ICALEPCS’2005, Geneva, Switzerland, October 2005
[8] P. de Vicente et al., “Development of the control system for the 40m radio telescope of the OAN using the Alma Common Software” , ICALEPCS’2005, Geneva, Switzerland, October 2005
[9] P. Di Marcantonio et al., “Transmitting huge amounts of data design implementation and performance of the bulk data transfer mechanism in ALMA ACS” , ICALEPCS’2005, Geneva, Switzerland, October 2005
[10] I. Križnar et al., “Migration from ACS 1.1 to ACS 4 at ANKA” , ICALEPCS’2005, Geneva, Switzerland, October 2005
[11] ACS Web page, http://www.eso.org/projects/alma/develop/acs/
[12] Open ArchitectureWare project home page: http://sourceforge.net/projects/architecturware/
[13] D. Fugate, “A CORBA event system for ALMA common software“”, SPIE 2004 - Astronomical Telescopes and Instrumentation, Glasgow, Scotland, June 2004, paper 5496-24
[14] S.D.Huston, J.CE Johnson, U.Syyid, **The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming**, Addison Wesley.
[15] K.Sigerud, N.Stapley, M.Misiowiec, T.Zygula, “First operational experience with LASER”, ICALEPCS’2005, Geneva, Switzerland, October 2005
[16] I.Verstovsek et al., “Java Abeans: Application Development Framework for Java”, ICALEPCS 2003, Gyeongju, Korea, October 13-17, 2003
[17] RTAI Web page. http://www.rtai.org/
[18] S.Harrington et al., ACS as the framework for integrating offline data reduction in ALMA, ADASS XV, San Lorenzo de El Escorial, Spain, October 2005