Application development using the ALMA common software

G.Chiozzi[*a], A.Caproni[ae], B.Jeram[a], H.Sommer[a], V.Wang[a], M.Plesko[bc], M.Sekoranja[b], K.Zagar[c], D.W.Fugate[d], S.Harrington[e], P.Di Marcantonio[f], R.Cirami[f]

[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 1N4

[e]National Radio Astronomy Observatory, Socorro, N.M. 87801, USA

[f]INAF-OAT, Osservatorio Astronomico di Trieste, via G.B.Tiepolo 11, I-34131 Trieste Italy

ABSTRACT

The ALMA Common Software (ACS) provides the software infrastructure used by ALMA and by several other telescope projects, thanks also to the choice of adopting the LGPL public license. ACS is a set of application frameworks providing the basic services needed for object oriented distributed computing. Among these are transparent remote object invocation, object deployment and location based on a container/component model, distributed error, alarm handling, logging and events. ACS is based on CORBA and built on top of free CORBA implementations. Free software is extensively used wherever possible. The general architecture of ACS was presented at SPIE 2002. ACS has been under development for 6 years and it is midway through its development life. Many applications have been written using ACS; the ALMA test facility, APEX and other telescopes are running systems based on ACS. This is therefore a good time to look back and see what have been until now the strong and the weak points of ACS in terms of architecture and implementation. In this perspective, it is very important to analyze the applications based on ACS, the feedback received by the users and the impact that this feedback has had on the development of ACS itself, by favoring the development of some features with respect to others. The purpose of this paper is to describe the results of this analysis and discuss what we would like to do in order to extend and improve ACS in the coming years, in particular to make application development easier and more efficient.

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

----

1. INTRODUCTION

The Atacama Large Millimeter Array (ALMA) is a large radio interferometer presently under construction in the Chajnantor area of the Atacama Desert in Chile. The 5000 meter-high site offers exceptionally dry and transparent meteorological conditions (better than the South Pole for a substantial fraction of the time).

The project was started as an equal partnership between Europe and North American institutes, and has more recently been joined by Japan.

ALMA will consist of 50 12m antennas capable of receiving in the frequency range 37 GHz to nearly 1 THz. Most of the photons in the universe are actually emitted in the ALMA observing range. The antennas can be relocated on fixed pads distributed over the site surface, with baselines up to nearly 14km, resulting in spatial resolutions as fine as 10 milli-arcseconds. NAOJ is providing four 12m antennas optimized to provide total-power (i.e., autocorrelation) data as well as a compact array of 7m antennas to provide higher sensitivity to broad, low-surface brightness features (each interferometer is sensitive to angular size scales inversely proportional to the separations between pairs of antennas (baselines) on the ground).

Prototypes of the antennas have been installed for evaluation and testing at the Very Large Array site, near Socorro, New Mexico. The first production antenna will arrive in Chile in the first quarter of 2007. By the end of 2008 there will

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

be 3 antennas ready for commissioning of the interferometer. An early science period will start in 2010 and the construction will be completed at the end of 2011.

The development of the ALMA software[3] is a very distributed effort, encompassing all of the many institutes involved in the project. It was therefore agreed at the beginning to require all developers to adopt a common software infrastructure, in order to establish a common, technical way of working in practice not just in principle, as it would be if based on written standards. Among other advantages, this approach minimizes the support burden required of the maintenance staff that would otherwise have to absorb development differences from the very distributed and heterogeneous ALMA development team.

ACS has been under development for more than 6 years and, by now, the core concepts (that have been presented at previous editions of this conference [1] as well as in other conferences [2]) are very stable.

This is a short summary of the most important aspects of ACS:

- ACS is used as the software infrastructure for the whole ALMA software, end-to-end, from data reduction to control applications.
- ACS provides the basic services needed for object oriented distributed computing using different programming languages. Among these are:
    - 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 wherever possible, to avoid “re-inventing the wheel”.
- ACS itself is publicly available under the Lesser GNU Public License (LGPL) license [8]
- ACS’s primary platform is Red-Hat Enterprise Linux, but it works and is used also on other Linux variants.
- Real time development is supported on Real Time Linux (for ALMA) and VxWorks (for other projects).
- Development is supported in C++, Java and Python. Any other language with a CORBA mapping can be used, if needed. Coherent support of multiple programming languages is one of the key motivations for the implementation of ACS.

The total effort allocated to ACS development by the ALMA project is on the order of 30 man years, but the project can count on additional external contributions as well**.**

In the last few years, ACS has been used extensively in the field:

- The ALMA prototype antennas have been running a first version of the ALMA Control Software and are now running the latest ALMA software baseline release.
- The end-to-end 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 good testing and feedback concerning the ACS global infrastructure, performance and tools from the operational and deployment points of view.
- ACS-based software is also used in various laboratories by the teams developing hardware and devices for ALMA. This provides feedback from users outside the community of ALMA software developers.
- Other projects, such as the Atacama Pathfinder Experiment (APEX) now in operation near the ALMA site[11], are using ACS. The feedback from the community of ACS users outside of ALMA provides bug reports and suggestions for design improvements as well as contributions to the development of ACS itself.

In this paper we look back and analyze the evolution of ACS in the past years. What happened to the initial concepts and features and how did they evolve in response to users’ requests and feedback? Where are we headed in the future?

2. THE FREE-SOFTWARE PHILOSOPHY

One of the first key decisions for the development of the ALMA software has been the one of embracing the free-software philosophy.

The problems we face in the design of our system are very similar to the problems encountered by many other projects, in particular the community of scientific experiments.

The adoption of commercial packages ties one to specific vendors, often with license costs that would be prohibitive for the budget of our project and with the high risk of being affected by changes in the commercial strategy of the vendor (for example when a product is no longer supported on the desired hardware platform).

Therefore we decided to build our software infrastructure by taking advantage of the experience of other projects, using as much as possible freely available and at the same time widely used software. By now there are also many industrial systems that similarly depend on free software.

A wide open software community promises also good and fast support through the usage of newsgroups and discussion forums. Open community forums are very active and replies come very often within a few hours.

The experience of these years with open software has confirmed these positive aspects, but has also shown us the downside:

- The lifecycle of open software is very fast and there is no or little support for older versions.
  When a bug is identified, the fix usually arrives very quickly, but it is almost always tied to the latest, "bleeding-edge" version of the software. Patches to previous versions are rare.
  Accepting the fix thus often means accepting new features, backward incompatibilities and, perhaps, new bugs. The alternative is patching the old code ourselves (this is possible since the source code is freely available).
- When an open software product really becomes mainstream the resources that the authors would have to put in support become quite substantial. We have seen that very often at this point a company is founded to provide support and consultancy as a way to pay the costs. This corresponds normally to a sharp decline in the contribution of the core authors to the newsgroups, in order to convince the users to purchase support from the company.
- Documentation for free software is very different from the documentation you normally expect from commercial products. First of all it is very different from package to package. Detailed and comprehensive user and reference manuals are typically absent. Good introductory books are written by the authors of the packages, again as a way to finance the project. Very often we end up having to look inside the source code.

All in all we can say that the advantages outweigh the disadvantages, but the costs should not be underestimated.

We have seen for example, that the costs during the past year for keeping pace with real-time Linux releases and having a stable system have been much higher than originally foreseen. We have been often forced to take versions of the real-time Linux development software that is "hot off the press," just to get basic features running, although reported problems have been fixed quite promptly.

3. THE CHOICE OF CORBA

The choice of CORBA[9] as the cornerstone for our infrastructure was perfectly in-line with the free software philosophy: CORBA is an open standard, with many free (and commercial) implementations to choose from; and it is a mature technology, also widely used in industry.

CORBA is by design vendor interoperable, platform independent and programming language independent and provides a large number of fundamental services or the building blocks to put together the ones we needed.

We are now using 5 free implementations simultaneously:

- TAO for C++ development and some services
- JacORB for Java development
- OnmiORB for Python development
- Mico for the Interface Repository service
- The OpenORB engine for some internal code generation

In the past several years, we have been able to change CORBA implementation or reallocate services to one or the other a few times without problems. This confirms that CORBA is up to our expectations in terms of openness and interoperability. We also always found an implementation capable of satisfying our requirements for performance.

But the free implementations suffer from the problems mentioned above. In a number of cases we had to implement our own patches. For example, JacORB does not support pre-processor macros in IDL files at all; this is accepted by the community of Java users of the tool, but it is not acceptable for us and we had to implement some extensions ourselves. Documentation is sometimes not satisfactory. When a company became available, "free support" degraded.

CORBA is also a very complex middleware, with a steep learning curve. A major task of the layers ACS builds on top of CORBA is to hide this complexity. The ACS development team should be in principle the only one dealing with CORBA machinery, while ALMA subsystem developers and other users should not have to care about that.
The strategy we have adopted to reach this objective is the following:

- Analyze which of the many features of CORBA we want to use and how
- Implement easy to use libraries that implement the selected patterns, hiding the complexity of the underlying middleware
- Provide advanced users with the possibility of doing more complex and unusual things accessing the underlying layers.

Initially we did not want to impose strong limitations on the usage of CORBA by developers. ACS should have made "normal things" easy, but not be a brake in the implementation of complex things. But we have since realized that in this way it is difficult for us to keep under control how much of "pure CORBA" is being used. With the future prospect of perhaps replacing CORBA with another middleware it is useful to keep under control what is used and how.

Therefore, we now prefer to hide the underlying middleware from the developers as much as possible, but keep the door open to provide access to additional functionality upon request. This has been implemented in the Component-Container model as described below.

4. COMPONENTS AND CONTAINERS

One of the core elements of the ACS architecture is a Component/Container Model[10].

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

By now .Net and Enterprise Java Beans (EJB) models have become mainstream commercial implementations and there are a few CCM free implementations available.

ACS has kept its original design and is not an implementation of CCM, but we have extended and cleaned up substantially our implementation as well as made concepts and naming more consistent with those of CCM.

The major advantage of our implementation is that, albeit currently simple, it is interoperable with all CORBA implementations and all programming languages we use. There is no CCM implementation that would work seamlessly with the C++, Java and Python ORBs we are using. As mentioned already in [1], CORBA 3 Interface Definition Language (IDL) specifications have additional features directly linked to CCM that are extremely interesting. While CORBA 2 IDL only allows specifying the interfaces provided by an object, CORBA 3 IDL allows, for example, defining also: used interfaces, published events and consumed events. As a first step in the direction of harmonizing ACS with CCM, we would like to work on extending our Component/Container model to support CORBA 3 IDL specifications.

The main purpose of the Component/Container model is to simplify the work for the application developer[1]:

- The Container, implemented by the ACS team, takes care of the technical concerns of the system, while 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.
- 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 the location 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 Container manages the lifecycle of the Components, activating and deactivating them when requested and managing their state according to a standardized state machine.

At the beginning, when the applications developed were small and not integrated together, it was difficult to get the application developers to appreciate these advantages. It is only above a certain threshold of complexity that the overhead of setting up the infrastructure pays off.

However, now that the ALMA software is regularly integrated and tested as a complex assembly of subsystems the advantages are clear in many areas. In particular the flexibility of re-deploying Components in different Containers (possibly on different hosts) provides good support for debugging and error tracking as well as for performance optimization. Re-deployment also allows keeping multiple test configurations that adapt to the hardware available at different sites. The whole issue of the configuration of the runtime system is now being actively discussed and revised. We are having problems in keeping aligned the configuration of the various ALMA deployments (test antennas, simulators, laboratories) and we are planning the implementation of higher level configuration tools capable of generating each specific runtime deployment from a central description.

In the recent ACS releases we have significantly improved the separation between Components and Containers by defining a ContainerServices interface. This interface provides an abstraction of the services that a Container provides and CORBA is largely hidden behind it. Before (in particular for C++), there were various situations where the Container implementation and CORBA were directly accessed by the Components.

In this way it is also in principle possible to replace our Container implementation with another, possibly based on different technology, by implementing ContainerServices and Component classes, without touching the component's application code. This was requested by the Offline data reduction subsystem, with the objective of implementing data reduction components to be used outside of ALMA, on systems based on other component/container technologies.

When access to features of the underlying middleware is necessary, we implement advanced ContainerServices sub-classes that have the additional features. In this way we can control who is requiring special capabilities.

Most recently, we have started concentrating our work on the deployment of the Containers. Until now our model assumed that the Containers for the deployment of Components were already running, for example started automatically in the boot procedure of the host computers or manually by startup scripts. We are now adding to the centralized Manager the capability to deploy dynamically not just Components on Containers, but also Containers on host computers. On the one hand, this will help us to better control startup and shutdown of the system and to manage failures by restarting or re-deploying Containers. On the other hand, it will allow us to satisfy requirements from the data processing and pipeline subsystems, allowing the implementation of load balancing recipes and the insertion/extraction of number crunching nodes from data processing clusters.

5. ASYNCHRONOUS COMMUNICATION

Since its initial design, ACS has provided four ways to communicate between components.

- Synchronous method calls.
  A client calls a method defined in the IDL interface of a Component.
  The action is complete when the method returns.
  We handle the special case of calls passing complex data structures using XML serialization techniques[10]
- Asynchronous method calls.
  A client requests explicitly a service from a Component by calling an asynchronous method defined in the IDL interface of the Component and registers callbacks that directly "connect" Component and client.
  The method returns immediately and the Component will invoke the callback periodically to report back to the caller or when completed.
- Notification Channel
  A Component publishes data over a CORBA Notification Channel and any interested client subscribes to the Notification Channel to be notified when data is available. Distinct from the asynchronous method call, there is no direct "connection" between publisher and subscriber, and the publisher is not aware of the subscribers collecting published data.

- The special case of publishing huge data volumes is handled in ACS by the Bulk Data Transfer package[5]. This implements high efficiency transfer of high volumes of streaming data and is based on the TAO implementation of the CORBA Audio/Video streaming service. This is essential for the ALMA Correlator to send data at rates between 6-60 MB/s to the Archive.

The ACS Notification Channel was originally meant to be used when one or more clients are interested in receiving events of a certain type regardless of their source. It is very well suited for n-to-m connections where n can be 1. The Notification Channel is also used internally in ACS, for example in the implementation of the ACS Logging and Alarm systems.

In recent years we have seen the Notification Channel used much more than expected, as a general mechanism to:

- Synchronize the activity of subsystems by means of the publication of synchronization events
- Publish data to be retrieved by one or many subscribers, not known a priori.

Components in many ALMA subsystems have been designed to publish status changes and data on the Notification Channel and clients can pick up this information without any specific knowledge of the publisher but only of the format of the data they want to retrieve and without establishing any direct connection to the client.

This mechanism has been very often preferred to callbacks, although callbacks provide better performance in the 1-to-1 case, because they establish a point to point, direct connection while in the Notification Channel mechanism all published messages are received by a Notification Service that re-dispatches them to the subscribers.

The ACS classes that implement the Notification Channel have evolved together with increasing usage and are by now very easy to use[12], hiding in an effective way the underlying CORBA Notification Service. The event names and data structures to be transported are defined in the IDL interface specifications and can be used with few lines of code. The definition of events at the level of IDL interfaces has been very important, not only to simplify usage, but also to make it possible to infer the usage of events using source code and run time analysis tools that we have developed.

While it is true that decoupling publishers and subscriber makes it easier to implement the code, it nevertheless makes it much more difficult to know which component is dependent on which other and to keep track of the dependencies. The source code tools we have developed analyze the code and build maps of dependencies, while a run time event browser allows inspection of the Notification Channel in terms of publishers, subscribers and events published. A bigger help in this direction would come from the adoption of CORBA IDL 3 event specification notation and this is why we are very much interested in this extension.

More in general we have seen that the ACS Component /Container model and the communication facilities provided, including in particular the Notification Channel, make it very easy to build systems with complex dependencies between Components and, more generally, subsystems. This is useful when such complex dependencies are really needed in the system. But it can be also very dangerous because circular dependencies make a system much more fragile and difficult to maintain and test. It might be very difficult, for example, to test in isolation two subsystems when they are both publishers and subscribers for each other's Notification Channels.

We urge system architects to stick to unidirectional, cascade, dependencies between Components as much as possible. This allows one to build the system in sequence and, very important, to test the subsystems in an ordered sequence.

Callback-based design techniques can help substantially in resolving architectural bidirectional dependencies at design time. If two Components have a bidirectional dependency, often one can be defined master and the other client and one of the two dependencies can be inverted by changing the interface to install a callback.

6. MULTITHREADING AND DISTRIBUTED APPLICATIONS

ACS provides the infrastructure to build highly distributed systems, where each Component behaves like an independent object. Such systems are intrinsically asynchronous and multithreaded: every Component receives messages independently from other Components living in the same or in other Containers. CORBA normally handles each message with a separate thread and different parties can invoke a method (also the same one) of an object at the same time, although it is possible to tune these policies.

Application developers have to be well aware of the concurrency problems they can encounter, but experience shows that developing reliable concurrent applications is much harder than implementing traditional single threaded code. Also debugging is much harder, because often the behavior of tests depends on timing issues and it is not deterministic.

On the part of ACS, we have to help the subsystem developers with support tools, the implementation of robust concurrent design patterns and with guidelines and documentation.

The first important aspect to keep in mind when developing ACS Components is that any method can be called at any time, in parallel to the same or any other method of the same component. Implementation of methods must be therefore always thread-safe. The Java language provides synchronization features as part of the language. In C++ we can rely on the powerful classes that are part of the ACE library, but we have implemented simple support synchronization primitives on top of them, to make writing thread-safe classes easier.

When a method is invoked concurrently to the same or other methods of the same class, there can be different options:

- Concurrent execution is allowed and it is sufficient to protect access to resources (for example reading values is normally allowed concurrently)
- The newly invoked method must be rejected (for example moving a device while the system is shutting down should not be allowed).
- The newly invoked method supersedes the old method, which should be aborted (for example, moving a telescope to a new target is a long operation and a new target should override the previous one, without waiting for the telescope to have reached position).
- The newly invoked method is queued and executed after completion of the previous one (for example telescope offsets could be implemented in this way).

At the present time, ACS leaves the implementation of these policies to the application developers. However, we think it would be very useful to provide support tools for their implementation. In particular all the policies described above could be implemented using state machines and ACS could provide reference state machine implementations based on a generic state machine code generator that we have developed for ACS internal purposes[2].

Another important aspect is the handling of user-created threads. Very often Components need to spawn their own threads, for example to implement control loops or to handle long lasting activities. Actually, when a call to a method in the interface of a Component takes a long time, it is very important to assign its execution to a separate thread and return soon to the caller. This is because CORBA resources remain allocated during the whole time a remote call is being executed, with the risk of creating deadlock situations if too many long lasting actions take place at the same time.

This is an important reason to use asynchronous method calls based on callbacks. But, also when using the Notification Channel, the functions that retrieve the data from the channel must process them as fast as possible or hand over the processing to separate threads. For this reason we have implemented monitoring of method execution time to warn the application developers when actions take more time than normally acceptable.

In order to help developers, we have implemented thread management classes. In particular we have introduced the concept of a Thread Manager, by which each Component has an associated object responsible for the management of all threads spawned by the Component itself. This makes it possible to tie the lifecycle of the threads to the lifetime of the Component. It is actually very important to make sure that when a Component is de-activated all related threads are cleanly terminated. Failure to handle this situation might introduce large instabilities in the system, often difficult to diagnose. Problems in this respect can come from the integration in Components of functionality coming from 3rd party packages: in this case we cannot rely on the Thread Manager to handle threads spawned by the external libraries.

Experience has also shown that the centralized logging system is probably the most essential tool to debug the system when dealing with multiple asynchronous components. Traditional source code debugging is, in such situations, of little help, because break points and function stepping dramatically affect the timing and concurrency of the system and therefore make it often impossible to reproduce run time error conditions. Our best option is to analyze a-posteriori the sequence of actions that took place by looking at the detailed logs. Other very useful debugging tools are run time code checkers like Purify and Valgrind that analyze in particular run time memory errors and generate postmortem reports.

Development and debugging is also improved by fast turnaround in the compile-run cycle. Since the initial design, ACS C++ Components have been implemented using dynamically loaded libraries. This allows one to unload the library when a Component is de-activated, modify and recompile it, then reload the Component with the new library, all without having to stop the Container. We have now implemented the same features in Java, by using custom class loaders, and in Python. This mechanism works very well and transparently for stateless Components. Since ACS does not provide any specific feature to handle and preserve the internal state of Components across reactivation, application developers have to take up this responsibility. This has been and still often is a problem in application code. This is therefore an area where we think ACS could be conveniently extended by providing standard mechanisms to handle stateful Components.

A suite of Tools for Automated Testing (TAT) helps in developing automated regression tests involving distributed Components. These tests for ACS and the whole ALMA software are integrated in a nightly build infrastructure that also evaluates code metrics and statistics. This is an essential tool to assess and improve the quality of our software.

7. DOCUMENTATION

Documentation is really critical for an application framework that is used by many developers distributed over many sites.

The learning curve of the framework for a newcomer is pretty steep, because before being able to develop code it is necessary to have a basic understanding of the Container/Component architecture and of the principles behind distributed systems based on CORBA or similar middleware.

Putting together the *"right documentation"* is very difficult, in particular for a team with limited resources. We cannot afford to produce the same level of documentation as that of a commercial product, and our users are aware of that. In the past few years we have changed direction a number of times in the way we put together our documentation and we have not yet found a really satisfactory compromise among all the different constraints.

Initially we have tried to write "classical" design documents and user manuals. But we soon realized that the maintenance of such documents was too expensive. Also, the design documents were good to introduce the principles to the users, but not sufficient to enable them to work with the APIs.

We have therefore put the emphasis in writing introductory tutorials and examples, limiting design documents to the key elements. The users are happy to use tutorials and examples and they often look at them. For us it is also useful to write step by step instructions, because this gives us a good opportunity to verify the usability of our design. Sometimes, though, the users consider the example code to be "perfect code", not realizing that to be able to easily show some aspects of the code we have to minimize the relevance of others. Typically, error handling needs to be sacrificed to make clear how code should be put together to reach a specific objective, but we often find application code which copies verbatim from incomplete examples. We have not found a real solution for this problem.

Detailed documentation for the APIs and classes is kept together with the code using Doxygen (Doxygen is almost identical to Javadoc but it is capable of supporting all our programming languages, including CORBA IDL). The tool works very well and maintaining the documentation is much easier; but still writing complete documentation is a huge effort and we still have code that is not documented as well as the users would like.

There are a number of reasons for this:

- It is very difficult to get developers spending a sizable amount of time writing documentation, even inline.
- What a developer writes is often not what a user needs. Important concepts are assumed as obvious.
- It is difficult to put good code examples inline. This should always be done with external tutorials.

A good solution would probably be to have a documentation team that includes also users. But our project is not structured in this way and we have to rely on the feedback received on the mailing lists to improve our documentation.

In the last few years, the introduction of the Wiki to disseminate free format documentation and FAQs has been a major step forward. The Wiki is very easy to maintain, since adding or correcting information is always just a few clicks away. The number of FAQs is growing very fast and we put there essentially all interesting questions we receive in terms of system configuration, coding, design patterns and so on.

We have had very good results from internal code and documentation reviews, in which ACS team members are assigned the task of inspecting the code and documentation of packages developed by others and of trying out the tutorials and examples. We have recently intensified this practice and we plan to do more in this direction.

8. THE ROAD AHEAD

By now, most of the ACS packages needed by the ALMA project have been implemented. But when we look in the details of the features provided by these packages, we see that we still have a lot of work to do.

Once the ALMA subsystems build applications using these packages, they think of new features they would like to have or new, unplanned, ways of using the available features. A big part of the work we do now does not come from long term planned developments but from change requests and extensions triggered by the feedback from the subsystems.

So, if we look at the list of features provided by ACS in the last releases and for the coming years we will not see major new packages. But the whole ACS is significantly improving in stability, usability and global consistency.

The core design patterns and technologies used in ACS are widespread and used also by other projects. We can share development and specialized packages with projects that do not use ACS but have similar architectures and the same underlying tools. This would be a very interesting step forward in component reuse between projects. It is often impossible for technical and managerial reasons to use the same software infrastructure in its entirety across projects, but the reuse of subsystems and components would nevertheless bring major advantages. As a first relevant example for this possibility we have started a collaboration with CERN for the adoption of the Alarm System developed for the Large Hadron Collider (LHC) within ACS, by replacing some interface components. This project is described in detail in another paper in this conference[4].

While most ALMA developers are by now "experts" in the usage of ACS, there are always newcomers and new projects interested in using it. It would be therefore very useful to be able to flatten the initial learning curve. We think that code generation could help a lot. In particular we would like to have code for Components directly generated from specifications. The Hexapod Telescope (HPT) and APEX projects have accumulated some interesting know-how by developing two generations of code generators able to produce very complete Component implementation code from the IDL specifications. The next release of ACS will integrate for the first time the latest code generators. This is really a major contribution to ACS coming back from the user community. The next step would be to start directly from a UML model, instead of the IDL files.

9. PROJECTS USING ACS

ACS is the common middleware for the entire ALMA software development project and it is therefore installed at the sites of all ALMA partners in North America, Europe and Japan, with the objective to satisfy the requirements of this project. ACS can, however, be used as a general software infrastructure for other scientific facilities, even outside the astronomical community. Since we believe that a wide community of users can provide excellent feedback and help us to have a solid system for ALMA, we have made ACS available under the Lesser GNU Public License (LGPL)[8].

There are a number of projects that have decided to base their system on ACS, such as the Atacama Pathfinder Experiment (APEX), the Spanish OAN 40 meter radio telescope, the Sardinian Radio Telescope in Italy, the Hexapod Telescope (HPT) in Chile and the ANKA Synchrotron in Germany.

We have also started collaboration with a group at the Universitad Tecnica Federico Santa Maria, at Valparaiso in Chile. The team develops code for ACS. ACS is also going to be taught in control software classes. This aims at preparing Chilean software engineers for ALMA, as they will be needed once the deployment phase of the project begins in Chile.

An active community of ACS users has emerged, with mailing lists, phone meetings and workshops. 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. We hope to see a wide base of reusable application components in particular for the astronomical domain.

10. CONCLUSION

The development of the ALMA software system is widely distributed and encompasses a mix of software cultures, with more than 20 development sites in 4 continents. In contrast to other projects, a single software team (albeit very distributed geographically) is responsible for the whole end-to-end project, from the control of the hardware up to the proposal preparation and data reduction tools running on the astronomer's desk. This provides a unique opportunity for building a coherent and uniform system, as opposed to having two separate systems for data flow and control, with unavoidable interface problems. Using a common software framework is key to reaching this objective. Such a framework increases maintainability and facilitates integration and testing activities.

On the other hand, this means imposing a way of working, technologies and tools that may be alien to development teams with a different culture and engineering tradition. It is often difficult to match both the needs of the control software developers and those of the data analysis groups and often conflicts need to be solved at the management level. But we have no doubt in the advantages for the teams responsible for high level analysis and integration as well as, later on, for the maintenance of the operational system.

In the last few years we have been able in many cases to demonstrate that teams responsible for specialized subsystems benefit as well from the "common software approach" and that their productivity is substantially increased. The first prototypes, isolated and with limited features, were comparatively complex to implement because of the overhead of the infrastructure. Now complex applications become much easier to put together thanks to the features made available by well tested infrastructure.

For small projects with limited resources, adopting ACS provides a big head start and the steep learning curve is compensated by the availability of a robust system and of an experienced community, willing to share solutions.

ACS reaches the objective of integrating off-the-shelf, freely available, infrastructural tools and making their usage more uniform and easier. We are also happy with the choice of CORBA and, in general, of most of the tools we have selected. More details, in particular on the technical aspects of ACS, are available from the ACS home page [7].

ACKNOWLEDGMENTS

The ACS project, based on an initial kernel developed by the Joseph Stefan Institute in Ljubljana, is managed by ESO with contributions from a number of ALMA sites, JSI, Cosylab, INAF-AOT and other projects. 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 feedback and important contributions to the definition and implementation of ACS.

REFERENCES

1. G.Chiozzi et al., "The ALMA common software: a developer friendly CORBA-based framework, Proc. SPIE Vol.5496-23, Astronomical Telescopes and Instrumentation, Glasgow, June 2004.
2. G.Chiozzi et al., "The ALMA Common Software, ACS, Status and Developments", ICALEPCS'2005, Geneva, Switzerland, October 2005
3. B.E. Glendenning, G. Raffi, "The ALMA Computing Project – Update and Management Approach" , ICALEPCS'2005, Geneva, Switzerland, October 2005
4. A.Caproni, K.Sigerud, K.Zagar, "Integrating the CERN Laser Alarm System with the ALMA Common Software", these proceedings.
5. R.Cirami, P.Di Marcantonio, G.Chiozzi, B.Jeram, "Bulk data transfer distributer: a high performance multicast model in ALMA ACS", these proceedings.
6. A.Bridger et al. "A common framework for astronomical observing tools, these proceedings.
7. ACS Home Page: http://www.eso.org/projects/alma/develop/acs/
8. LGPL, Free Software Foundation, http://www.fsf.org/licenses/lgpl.txt
9. CORBA Home Page: www.corba.org/
10. H.Sommer, G.Chiozzi, "Container-component model and XML in ALMA ACS", SPIE 2004 - Astronomical Telescopes and Instrumentation, Glasgow, Scotland, June 2004, paper 5496-24.
11. D.Muders et al., "APECS - The Atacama Pathfinder Experiment Control System", 2006, ASP Conf. Series, Vol. CS 351, ADASS XV, Oct. 2-5, 2005.
12. D. Fugate, "A CORBA event system for ALMA common software"", SPIE 2004 - Astronomical Telescopes and Instrumentation, Glasgow, Scotland, June 2004, paper 5496-24