

The ALMA Software Architecture

Joseph Schwarz,^a Allen Farris,^b and Heiko Sommer^a

^aEuropean Southern Observatory, Karl-Schwarzschild-Str. 2, Garching, Germany;

^bNational Radio Astronomy Observatory, Array Operations Center, P.O. Box O, 1003
Lopezville Road, Socorro, NM 87801-0387 USA

ABSTRACT

The software for the Atacama Large Millimeter Array (ALMA) is being developed by many institutes on two continents. The software itself will function in a distributed environment, from the 0.5-14 km baselines that separate antennas to the larger distances that separate the array site at the Llano de Chajnantor in Chile from the operations and user support facilities in Chile, North America and Europe. Distributed development demands 1) interfaces that allow separated groups to work with minimal dependence on their counterparts at other locations; and 2) a common architecture to minimize duplication and ensure that developers can always perform similar tasks in a similar way. The Container-Component model implements the separation of functional from technical concerns: application developers concentrate on implementing functionality in Components, which depend on Containers to provide them with services such as access to remote resources, transparent serialization of entity objects to XML, logging, error-handling and security. Early system integrations have verified that this architecture is sound and that developers can successfully exploit its features. The Containers and their services are provided by a system-oriented development team as part of the ALMA Common Software (ACS), middleware that is based on CORBA.

Keywords: software architecture, ALMA, radio astronomy

1. INTRODUCTION

The architecture of a large software system is important to the development of a consistent whole that meets its requirements and whose parts carry out their functions with a minimum of duplication and a maximum of efficiency. In a development environment such as ALMA's, in which developers are scattered over a dozen institutes on two continents, the unifying force of a properly designed and agreed upon architecture becomes essential.

In this paper we present the key features of the architecture we have developed for the ALMA project. In order to motivate our choices, we first briefly describe the project, the requirements that it places on the software, and the constraints imposed by the environment in which ALMA will operate. We then introduce the main functional building blocks of the software and how they operate together. Finally, we discuss the technical solutions to the challenge of developing the ALMA software system.

1.1. The Atacama Large Millimeter Array (ALMA)

ALMA is a joint project of the North American and European astronomical communities, with Japan likely to join in 2004. It will consist of 64 antennas each 12 meters in diameter which will work as an aperture synthesis telescope to make detailed images of astronomical objects in the millimeter and submillimeter wavelength bands. They will be positioned as needed with baselines from 0.5 to 14 kilometers so as to give the array a zoom-lens capability, with angular resolution reaching 10 milliarcseconds. ALMA will represent a leap of over two orders of magnitude in both spatial resolution and sensitivity, making it ideal for medium scale deep investigations of the structure of the submillimeter sky.

ALMA will operate at an altitude of ~ 5000 meters on the Llano de Chajnantor in Chile's Atacama desert. This site was chosen for its dry, nearly rainless weather, which offers some of the best conditions on Earth

Further author information: (Send correspondence to J.S.)

J.S. E-mail: jschwarz@eso.org, Telephone: +49 (0)89 3200 6363

for observations in the mm- and submm-wavelength bands. Even with such a carefully chosen site, however, observations at the shortest wavelengths are possible only part of the time, when atmospheric water vapor content is low enough. To seize these opportunities, ALMA will operate in a dynamic scheduling mode: observing projects in the shortest wavelength bands will be selected when available conditions permit, taking into account their scientific priority as well. This kind of operation is, as we shall see, a major driving force in the design of the software system.

2. REQUIREMENTS

2.1. Scientific Requirements

The ALMA software must support all phases in the life of an observing project, *i.e.*, 1) preparation of an observing proposal by a research scientist; 2) peer review of submitted proposals, which results in either acceptance (possibly with modifications) and prioritization or rejection; 3) preparation of an observing program from an approved, possibly modified, observing proposal; 4) dynamic scheduling of programs, based on assigned scientific priority and needed observing conditions; 5) execution of the programs themselves, with monitoring of progress and observatory performance; 6) online calibration and feedback of results to optimize the observing procedure; 7) final calibration and imaging; and 8) archiving of data products and their delivery to the observer. The Principal Investigator is guaranteed exclusive access to the scientific data from his/her project for a period of approximately one year. Once this proprietary period has expired, the data is made available to the general astronomical community via a Science Research Archive which conforms to the standards set for the Virtual Observatory (VO).

A general requirement is that the software should make millimeter interferometry accessible even to the astronomers who are not expert in aperture synthesis while preserving the expert's ability to exercise full control. The software must be able to translate from the high-level concepts of the user's domain to the technical specifications and instructions for the observatory. For example, the observer may need to specify only the field of view and angular resolution that he wishes for his final image, but the software will need to decide whether a single-field observation or an On-the-Fly mosaic will be required; it will need to recognize when the array is in a configuration that is compatible with the observer's goals. The burden of dealing with this underlying complexity thus falls to the software developer. One task of the system architecture should be to relieve the developer of *unnecessary* complexity. Our primary vehicle for achieving this is the separation of functional from technical concerns, which is discussed below.

2.2. Performance Requirements

ALMA's baseline correlator will produce ~ 1 Gbyte/s, which the software must Fourier-transform from the time to the frequency domain and reduce to average/peak data rates of 4/40 Mbyte/s. The data that results from the later imaging of this raw *uv*-data will increase these figures by about 50% to 6/60 Mbytes/s, implying ~ 180 Tbyte/y to archive. The experience of the Hubble Space Telescope shows that Archive *access* rates, driven by the astronomical community and the general public, may be a factor of ~ 5 higher.

As this data is acquired, the online calibration software must be able to calculate pointing and focus corrections, phase corrections and average phase noise and feed these results back to the observing process in ~ 0.5 s, so that antenna pointing and focus can be adjusted immediately and the time spent observing target and phase calibrator can be adjusted to match atmospheric conditions should they be changing (Lucas 2004).

Once observations belonging to a given Observing Program are complete, the science data processing, *i.e.*, the production of images via calibration, Fourier transformation and deconvolution, must keep pace (on average) with the rate of data acquisition.

2.3. Extensibility and Scalability Requirements

ALMA is breaking new ground in the development of high-precision antennas, sensitive, low-noise receivers in the 100-1000 GHz frequency range and other hardware. Many of the observing modes, especially at the higher frequencies, can only be developed through experience with a working instrument. New observing modes will

imply new procedures for observing and new algorithms for processing the resulting data. Thus the software itself must be flexible and open to change.

Although ALMA will not be operational in even a limited sense until late 2007, key hardware components are being enhanced with capabilities well beyond those called for by the original specifications. For example, the wish to take advantage of such an enhancement to the baseline correlator has produced a proposal to raise the limits on the total data rates cited above to 25/95 Mbyte/s. This would increase the required archive capacity to ~ 750 Tbyte/y, and would exceed by a large margin the communications and storage capacity currently planned for the observatory. We must expect that changing and evolving requirements will be a fact of life during the course of the ALMA project.

From these considerations it follows that ALMA software should rely on standard rather than proprietary solutions. The software responsible for interfacing to the hardware—antennas, receivers, correlator and computing facilities—should be isolated to well-identified and restricted areas of the code. It is worth singling out database technology as a part of the system that needs to be encapsulated in this way; major increases in required archive capacity are likely to depend on the use of new developments in this area. With such an encapsulation strategy, the software can be retargeted to new hardware platforms as this becomes necessary or desirable.

2.4. Maintainability Requirements

As for the ALMA project in general, the development of software for ALMA is a collaborative effort, divided among about a dozen institutes on two continents, whose software staffs differ in development backgrounds, styles and cultures. When all the pieces of the ALMA software are brought into service at the ALMA operations site, it will be important that the observatory's commissioning and (later) operations teams be able to understand and modify them. Experience with the commissioning of the Very Large Telescope in Chile has shown that no developer should “own” the software that he or she develops; it must be accessible to others. To make this possible in practice, it is important to have a common framework and a common programming model that guides every developer in the project.

Using such a widely scattered and diverse group of developers places another demand on the software architecture: it must divide the system into modules that can be developed by individual groups but that can be integrated into a coherent system. This implies that interfaces between groups be as narrow as possible and that the mechanism for defining and changing them be standard across the project.

3. A DISTRIBUTED OBJECT ARCHITECTURE

Large distances will separate ALMA's 64 antennas from each other and from the correlator. Computing capability will be embedded with the instrumental hardware, facilitating independent development, testing and maintenance of these subsystems.

The 5000m altitude of the ALMA array site, chosen to give optimal conditions for observing at mm- and submm-wavelengths, is inhospitable to human activity because of the air's reduced oxygen level. Consequently, all but the minimum necessary operational activity will be carried out at lower altitudes. In particular, operation of the array will be conducted at the Operations Support Facility (OSF), at an altitude below 3000m. All computing activity that does not require proximity to the instrument's hardware will be performed here, or at even more distant sites: the Santiago Central Office, the ALMA Regional Centers in North America and Europe, and the home institutes of ALMA observers. The high bandwidth of fiber-optic network links between the array site and the OSF will be necessary to make this kind of operation possible.

A distributed physical architecture of this kind naturally leads to an architecture of distributed software objects. The large inter-hardware distances result in time delays, bandwidth limitations on data transfer rates, and occasional failures. It is therefore important to design the software in a way that reduces reliance on the network infrastructure to that strictly necessary for the software units to carry out their tasks. We end up with a loosely-coupled federation of software units, each operating with a considerable degree of autonomy.

As will be discussed in more detail in section 4, subsystems may require various implementation strategies. According to their tasks, these subsystems may use different hardware, system software and programming languages. The distributed architecture must, therefore, support this kind of heterogeneity.

3.1. Complexity of a distributed object architecture

Considerations that apply particularly to the distributed object model are:

- Finding local and remote resources (a naming service)
- Accessing such resources (interfaces and communication protocols)
- Granting resource access only to authorized users (security)
- Asynchronous messaging (event handling, publish/subscribe, notification)
- Configuring and controlling the ensemble of distributed software objects from a single location
- Error and failure handling, particularly the issue of *partial* failure
- Supporting object implementation in a variety of programming languages.

Design and implementation of the infrastructure that provides solutions to these requirements is a very complex systems development task. ALMA software uses the Object Management Group’s Common Object Request Broker Architecture (CORBA), which is supported by many high-quality open-source implementations. The choice of CORBA over competing standards was driven largely by the need to support the heterogeneous, multi-language and multi-platform ALMA computing environment. Because the application programming interface (API) that CORBA presents to the application developer is itself very complex, one of the main objectives of the ALMA software architecture is to hide this complexity while still providing the necessary access to its facilities. The framework for doing so is the ALMA Common Software (ACS), which furnishes the fundamental building blocks of the ALMA software system.

3.2. Functional vs. Technical Architecture

Expressing the complexity in software of operating a mm-wavelength interferometer is difficult enough for the developer without the additional burden of having to know in detail the all the subfields of computer science associated with distributed object architecture, such as remote access, network protocols, and database technology. The *separation of functional from technical concerns* is a strategy for enabling the application developer to concentrate on the physics, algorithms, and hardware details of aperture synthesis interferometry, while a specialized, system-oriented team provides an easy-to-use *technical* infrastructure.

The functional architecture further apportions these interferometry-related tasks among subsystems that can be developed in relative independence from each other. The technical architecture furnishes developers of these subsystems with simple and standard ways to 1) access remote resources; 2) store and retrieve data; 3) manage security needs; and 4) communicate asynchronously with other subsystems and components.

4. FUNCTIONAL ARCHITECTURE

We now turn our attention to the principal data structures of the functional architecture, followed by the overall system data and control flows, with emphasis on the role of the ALMA Operational Archive.

4.1. Observing Projects and Scheduling Blocks

The Observing Project is the structure that accompanies and directs a scientific project from its inception as a Principal Investigator’s (PI’s) proposal, through review, scheduling, execution at the telescope, processing and final archiving and delivery. Figure 1 shows an Observing Project and its constituent parts. It includes the original observing proposal with the scientific parameters (*e.g.*, target position, field of view, observing frequencies, spatial and spectral resolution, desired sensitivity and desired dynamic range) that the PI has specified. This “proposer view” has its counterpart in the “observatory view,” which contains the technical parameters and instructions necessary for ALMA to carry out the observer’s program in an automatic way.

Fundamental to the observatory view is the Scheduling Block (SB), which is the smallest sequence of observing instructions that can be scheduled. Changes in atmospheric conditions at the ALMA array site can change the

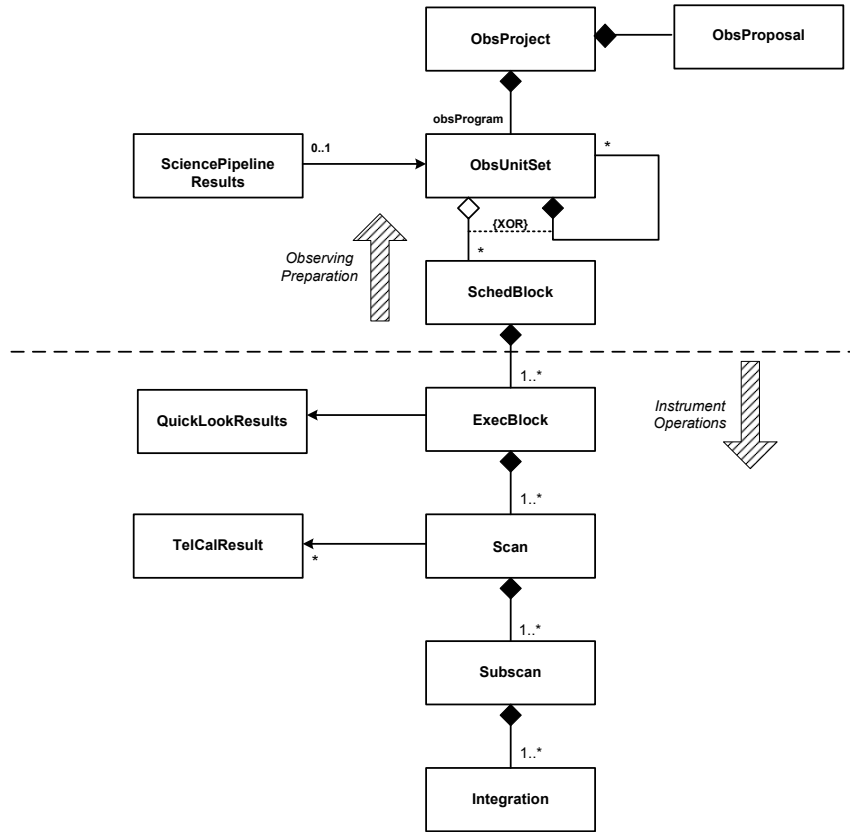


Figure 1. The Observing Project hierarchy.

kinds of observations that can be carried out. When water vapor levels are low, atmospheric windows at the highest frequencies open and observing at these frequencies becomes possible. In order to be able to exploit such opportunities, observing programs are divided into Scheduling Blocks. Each SB has an execution time limit, typically 30 minutes. Within this time limit, the SB must be able to carry out the setups, calibrations, and target observations necessary to ensure that the acquired data can be properly calibrated and used in the construction of the final data product. The end of a scheduling block may be specified in terms of a maximum amount of time or when certain well-defined science goals have been reached. An SB is atomic, in the sense that it cannot be restarted in the middle; an SB either runs to completion or fails or is terminated by the operator.

A scheduling block contains fixed metadata and an observing script. The observing script is interpreted and executed by the control subsystem; it represents the commands that carry out the observing intent of the scheduling block. This observing script may be a standard observing script to which the PI furnishes parameters, or instead, it may be a so-called “expert” script, one that is hand crafted by the PI or ALMA staff in order to test a candidate standard observing mode or to accomplish one which is unusual. The fixed metadata specifies basic observing parameters such as target direction and instrumental setup, as well as required weather conditions and maximum execution time. The scheduling subsystem depends on this information to determine whether an SB can be executed under the current conditions and to rank it according to, among other things, the scientific priority that the ALMA review committees have assigned to it.

In general, SBs that require much more than the nominal execution time are likely to be given lower priority by the scheduling subsystem, since they restrict the subsystem’s ability to react to changes in weather. Therefore execution of several SBs may be needed to perform all observations needed for the final image; it is convenient to group these in Observing Unit Sets. An Observing Unit Set contains either other Observing Unit Sets or SBs. The observatory view begins with the observing program, and descends in a recursive hierarchical structure of

Observing Unit Sets that terminate with SBs as their leaves. This structure gives the observation preparation tool flexibility in structuring complex projects, such as surveys and mosaics. When all SBs in an Observing Unit Set have been observed, the data can be queued for pipeline data reduction (usually imaging). Many programs will require the use of different antenna configurations and, since the full range of configurations repeats over a many month timescale, it is natural to group all SBs for a single target requiring the same (or nearby) configurations into their own Observing Unit Set. In this case, a complete image can be produced only when two levels of Observing Unit Sets (all the SBs per configuration for all needed configurations) have been observed.

4.2. Execution blocks and pipeline processing

For each execution of a scheduling block, the control subsystem creates an execution block that is attached to it. A scheduling block can be executed more than once, so in general there will be many execution blocks attached to a given scheduling block. The execution block contains a record of the parameters and conditions under which the SB was executed along with references to the acquired data.

The basic observational units that result in the production of scientific data form the hierarchy of scan, subscan and integration shown in fig. 1. The actual execution of a scheduling block by the control subsystem consists of the construction and execution of a sequential series of scans. Each scan consists of the construction and execution of a sequential series of sub-scans. Each sub-scan is itself broken into a series of integrations. Although commands are issued at the scan or sub-scan level correlator output corresponds to or is attached to a particular integration. Calibration results from the telescope calibration subsystem and quick-look pipeline results are usually attached to a sub-scan. However, some of these results may be accumulated over many sub-scans and the resulting objects attached to the scan to which the sub-scans belong. The control subsystem is responsible for creating the meta-data represented by the execution block, scan, sub-scan, and integration objects. This meta-data is necessary for the downstream processing subsystems to interpret the raw data as it arrives. Was this a calibration scan or a target scan? If the data correspond to a calibration scan, what kind of calibration was called for, pointing, focus, phase, ...? The telescope calibration and quicklook subsystems use the answers to these questions to decide which of the correlator data to intercept and process.

4.3. Data Flow

Having seen the key structures in the observing process, we can now examine the process itself. Figure 2 shows the flow of ALMA data from Observing Proposal through Scheduling, Data Acquisition, Calibration, Imaging, Archiving and final delivery to the PI. By following the numbered arrows in figure 2, we trace the data and control flow over the lifetime of an observing project. First, 1) an observer creates an observing project using the Observing Tool, which breaks the project into scheduling blocks, and 2) stores it in the archive. Observatory operations are initiated at the OSF by the executive subsystem. At this point, 3) the scheduling subsystem gets project definitions and scheduling blocks from the archive, selects one via an algorithm designed to prefer projects with high scientific priority and exploit changing observing conditions, and 4) dispatches it to the control system to be executed. The control system 5) executes the scheduling block by 5.2) commanding the correlator, which results in 5.3) raw data and 5.4) meta-data being made available to the 5.5a) calibration and 5.5b) quick-look subsystems. The completion status of scheduling blocks is monitored by the scheduling subsystem, which 6) starts the science data reduction pipeline at appropriate times (*i.e.*, when an Observing Unit Set has been completed). The science pipeline generates data products that are (7.2) stored in the archive. Again the scheduling subsystem monitors the completion of the science data reduction and (8) informs the PI that data are now available.

Figure 2 also shows additional functions, some of which lie outside the more-or-less automatic operations flow, such as f) access by researchers to data in the Science Research Archive, c) and e) manual interventions by the telescope operator, g) responses to project breakpoints by the PI, and h) the storage of administrative data. Additional functions that are continuous and automatic, regardless of whether science observations are executing, include a) and b) the storage of engineering data from all of the array's instrumental hardware.

The different subsystems involved in this end-to-end observing process require different implementation strategies. The Observing Tool, for example, is a highly interactive, GUI-based application that is used for preparation of both observing proposals and approved observing programs. It is intended to be used by researchers at their home institutions and, since it must therefore be largely independent of particular computing hardware, it is

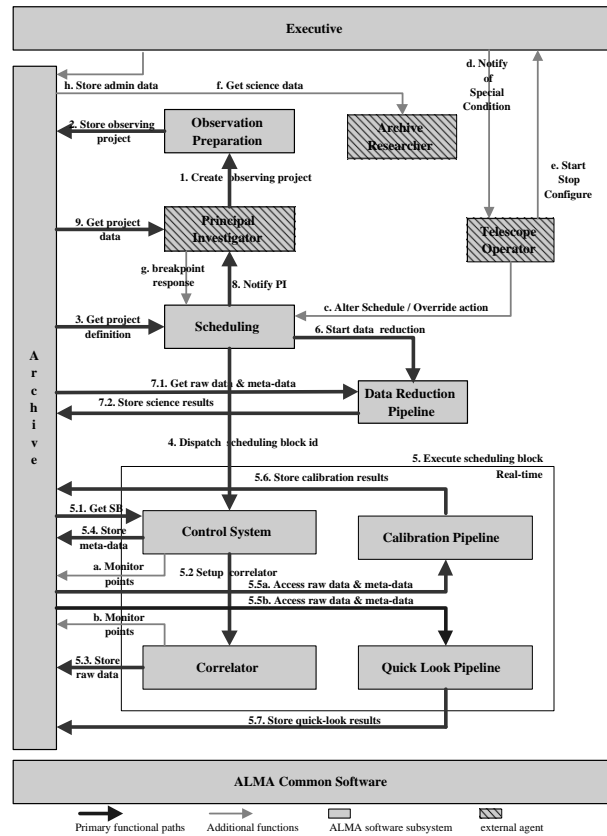


Figure 2. A schematic view of the ALMA system data flow. Numbered arrows indicate the order in which data flows occur during the lifecycle of an Observing Project.

coded in Java. The control subsystem has many interfaces to the instrumental hardware. It is the one truly real-time subsystem within ALMA: it must synchronize actions at each antenna and the correlator to within 48 milliseconds. The portion of the control subsystem that is nearest to the hardware is written in C++, while the higher-level code is written in Python and Java. The data reduction software, although it has no real-time constraints, is very compute-intensive; depending on the complexity of the particular observing project considered, it may even need to be executed on a parallel-processing platform. It encapsulates a large body of aperture-synthesis expertise coded in C++ and Fortran.

The need to make such disparate software units work together is one of the principal drivers for the technical architecture, discussed in section 5.

4.3.1. Central Control and the Executive Subsystem

The operations of the observatory, from the scheduling through data acquisition, processing and storage of generated data in the archive are managed from a single point of control, the Executive subsystem (shown as the horizontal bar at the top of figure 2). This subsystem starts the ALMA Common Software and its CORBA-based services and then initializes and starts the various software subsystems in a two-pass procedure that enables dependencies between subsystems to be resolved and deadlocks prevented. During observatory operations, the Executive listens to asynchronous events published by the subsystems, 1) displaying results from the Control, Telescope Calibration and Quicklook subsystems so that progress and quality of the data acquisition and calibration processes can be monitored; and 2) bringing error conditions to the attention of the array operator.

Planned downtime for all or part of the array is communicated to the Scheduling system by the Executive, which takes the output of a COTS planning tool and converts it to SBs that must be scheduled at fixed times.

4.3.2. Archive

The ALMA Archive plays a central role in the operation of the observatory. Rather than just a terminal point for an already completed data processing run, it serves both as buffer and storage for all data generated for and by ALMA, from the observing proposals and programs, the observing logs, hardware telemetry and raw correlator data, to the final calibrated images.

There are three main types of data handled by the Archive:

- bulk data: raw data from the correlator and image data from the science reduction pipeline; the raw data will consist of individual integrations, amounting to about 3×10^6 records per year of the high-volume, spectral channel data, and 6×10^7 records per year of channel-averaged data;
- complex data structures (see section 5.3): data that describes observations and bulk data (mainly the Observing Project hierarchy discussed above); the size of individual data objects and the number of records will be in the 1000's per year;
- monitor, environmental, and logging data; this data while low in volume, may produce many records on subsecond time scales;

The Archive is optimized for input/output streaming of bulk data, *i.e.*, the 4-40 Mbyte/s data rate of the correlator; random access times are longer.

4.3.3. Data models and data capture

The Archive itself is data model-neutral, in the sense that it does not specify or restrict the organization of the data that it handles. This simple fact is fundamental to its successful operation; there are, in fact, two principal data model domains in ALMA: telescope and science processing. The data models in the telescope domain reflect the data as it is acquired by the array; this is primarily time-sequenced data which frequently crosses target and project boundaries. The science data reduction software, however, needs to access data for a specific target within a specific project. Having to do complex Archive queries to assemble this data would result in unacceptable performance penalties; the data must be organized in a way that is optimal for the processing subsystem. It is also desirable that the data models in these two domains, given the factors that condition them, be able to change independently from each other. To put it another way, the data models in the telescope domain should know nothing about the data models in the science domain; likewise, the science domain should know nothing about what is occurring upstream from it, *i.e.*, in the telescope domain. Figure 3 shows the relationship and data flow between these domains.

There is a single interface between these two domains. A *data capture* component runs within the control subsystem and recasts the incoming meta- and auxiliary data into the model needed for science processing. The bulk data which issues from the correlator is produced in a manner consistent with the science processing data model; the data capture component does not reorganize the bulk data, but rather creates the links in the reorganized meta-data to point to them. Because the meta- and auxiliary data constitute a very small fraction of the total data rate, some of this data can be duplicated for different uses. Engineering analysis, to follow trends in the hardware or diagnose problems for example, is instrument-centric; it is convenient to maintain the monitor data in the form in which it issues from the hardware. That part of this data that is needed for scientific data processing can be extracted from the monitor stream and stored according to a data model appropriate to the science domain.

5. TECHNICAL ARCHITECTURE

The technical architecture for the ALMA software system provides the broad infrastructure necessary to support the functional architecture described in the preceding section. As the headings of this and the previous section suggest, we want to separate two types of concerns, the domain-specific functional concerns, from the technical concerns that arise from the computing environment in which the problems of Alma operations, data acquisition and data analysis are to be solved.

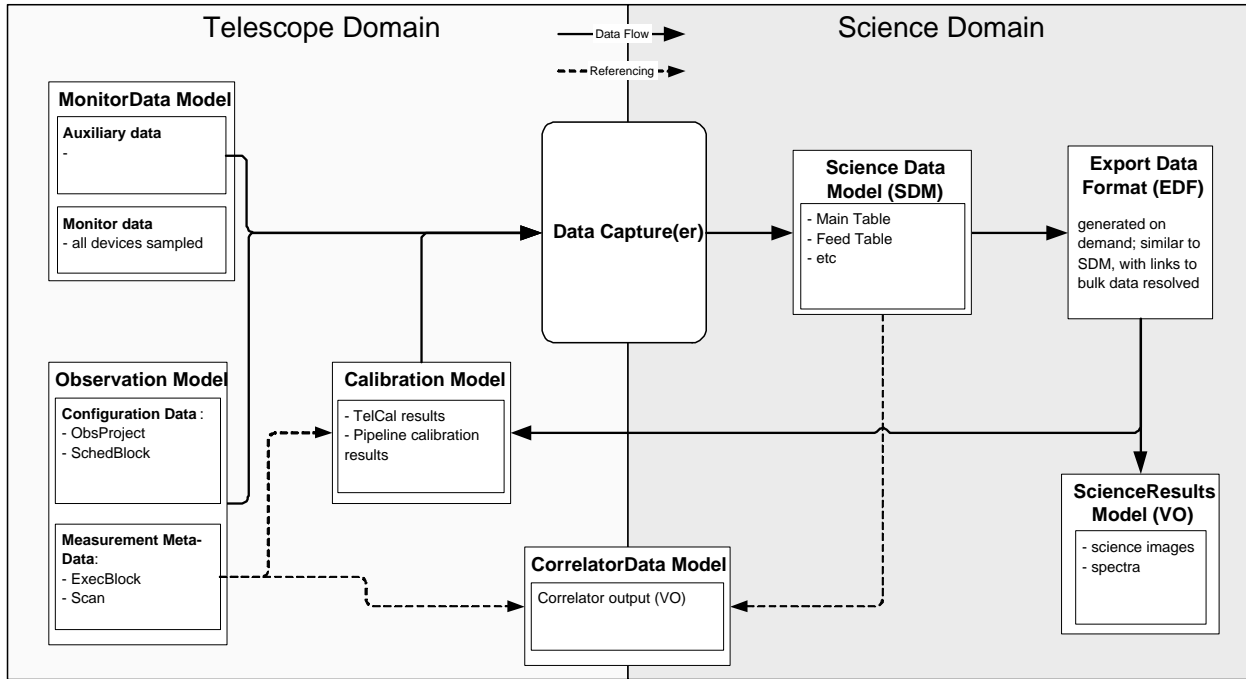


Figure 3. Data models and data flow in the telescope and science domains.

5.1. ACS

The ALMA Common Software (ACS) implements the separation of functional from technical concerns, especially the container/component model, which we discuss in more detail below. More generally expressed, ACS is a framework for a distributed object architecture that is used from the highest level software all the way down to the device level in the subsystems that control antennas, receivers and correlator (Chiozzi¹). Built on CORBA, ACS hides CORBA's considerable complexity, wrapping those CORBA services used by ALMA.

ACS has been carefully designed to be independent of commercial software, for example through the use of high-quality open-source ORBs such as TAO and JacORB. It is constantly evolving to meet developers' needs, having reached, as of this writing, Release 3.1. The most recent developments have included the implementation of the Python Container, needed by the subsystem responsible for pipeline data processing, and an all-Java version (albeit with reduced functionality), which is necessary to meet the portability requirements of the Observing Tool discussed in section 4.3.

5.2. Container/Component model

Following Völter,² we define a component as a coarse-grained software element that exposes its services, explicitly declares its dependencies on other components and resources, can be deployed independently and requires a runtime environment. Components depend on this runtime environment, the *container*, for necessary services.

ACS provides containers for C++, Java and Python. A container may host one or more components; deployment decisions for components are made at run-time. For C++, a container and its components execute in a single process and a single address space. In Java, a container and its components share the same virtual machine. CORBA ensures that communication between components in the same container occurs via native language invocations, while accesses to components in other containers (either on the same node or at a remote node) are mediated by CORBA's remote invocation facilities. ACS adds a layer that hides the need for explicit marshalling/demarshalling of complex data objects to and from XML (see section 5.3).

Components publish a required *lifecycle* interface used by the container to administer them, and one or more *service* interfaces, which specifies the methods offered to clients, including (but not restricted to), other

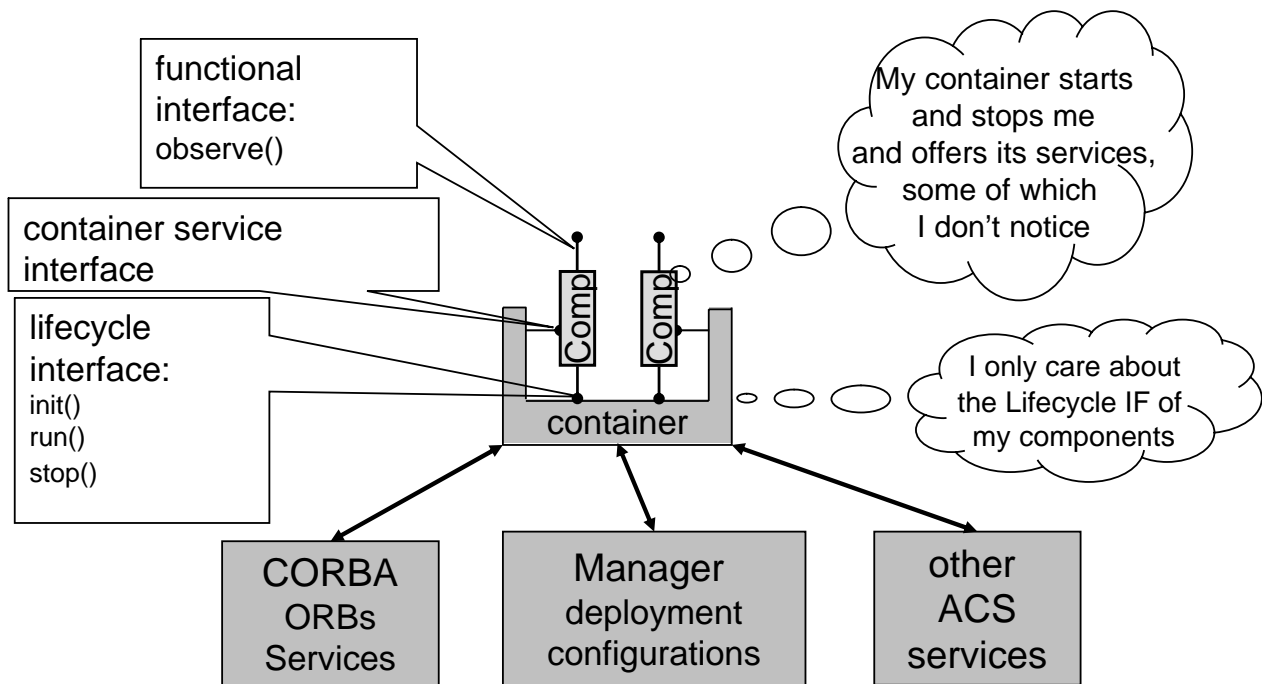


Figure 4. The Container/Component environment.

components. All interfaces are written in OMG IDL, the language-independent Interface Definition Language, which supplies the basis for method invocation by clients. The container/component environment is shown in figure 4.

Commercially, the container/component model has been implemented as Enterprise Java Beans and, by Microsoft, as .NET. A specification by the OMG, creators of CORBA, is CCM, the CORBA Component model. Some open-source implementations of CCM have started to appear; so far, each supports only a single programming language. The commercial implementations require a wholesale commitment to either the platform (.NET) or programming language (EJB). Moreover, the focus in these implementations is on business systems with large transaction rates, which are not characteristic of ALMA processes. At the time of the first ACS release, the CCM specification was not yet complete. For these reasons, the ACS team has developed an implementation of the container/component model tailored to the requirements of ALMA. Nevertheless, ACS will maintain as much compatibility with CCM principles as is consonant with the ALMA's more restricted needs.

5.2.1. Container provided services

The container provides the interface through which components access services that may be implemented by the CORBA ORB, other parts of the ALMA Common Software, or the container itself. Containers are designed to relieve the application developer of knowing the details of CORBA. By requiring clients to implement a simple lifecycle interface, the container (in cooperation with other ACS facilities), eliminates the need for the component to load and configure the ORB. In order to access another component, a component needs to know only the other's name or type; the container handles the interaction with the ACS Manager, which is responsible for component deployment and name resolution using a configuration database and the CORBA Naming Service to find the requested component. The container may pass the CORBA object reference to the client component, allowing the client to communicate directly with the server component (a *porous* container, used for performance-critical components in C++), or may itself mediate all further interaction between client and server component (a *tight* container, currently the norm for Java components).

Similarly, the container wraps CORBA's Notification Service, which provides components with asynchronous event-based communications. Components may *publish* information to which other components may *subscribe*.

Because the full Notification Service has a great deal of functionality not needed by the ALMA software, a much simpler interface can be presented by the container to its components.

The container provides a centralized logging service to components. Log entries are distributed to interested clients via the publish/subscribe mechanism. Each supported language has an API for generating, formatting, filtering and caching log entries that is appropriate for that language. For C++, this is the logging mechanism implemented by the Adaptive Computing Environment (ACE). For Java, the Java Logging API is used. For Python, an ACS Log Server provides generic logging capability.

Finally, exceptions thrown but not caught at the component level are handled by ACS via the container.

In general, additional functionality can be added to the system with minimal impact on components by implementing this functionality in the container. For example, although there is not yet a well-defined need for it, in the future the container could also manage security policies. Each component needing this facility would group its public methods into separate interfaces according to their different security needs. These security policies would then be set at deployment time and the task of enforcing them delegated to the container.

5.2.2. Lifecycle interface and the master component

implements state machine that clients can use to determine subsystem status

5.3. Entity objects and XML

The ALMA software needs to deal with many complex data structures, the most important being the Observing Project hierarchy discussed in section 4.1. These data objects need to be:

- Modeled or defined in a programming-language independent way;
- Created, modified and deleted by application code;
- Passed (often by value) between subsystems;
- Stored in and retrieved from the Archive.

The eXtensible Markup Language enables us to address these requirements in a unified way. We define the contents and structure of, for example, a Scheduling Block with an XML schema. (We can do this readily with one of the many graphical schema editing tools available.) In the case of Java, open-source frameworks are available that generate Java binding classes directly from the schema; these classes include methods to access and modify data elements in an XML document and to validate the document from its schema. It is worth pointing out that the application (component) developer needs only to lay out the structure with a graphical tool and use the binding classes from his/her application code; the rest of the process (generating the schema and from it, the binding classes) is automatic. (For details, see Sommer.³)

Although CORBA allows distributed objects to invoke methods of other distributed objects across the network, data objects such as Observing Projects or Scheduling Blocks are sufficiently complex that it is more efficient (*i.e.*, faster and with less load on the network) to pass the entire object by value across the network and operate on it locally. In order to do this, the data object needs to be serialized for transmission across the network. The XML document itself is the obvious choice as a serialization format, and thus the container provides a service to transparently serialize/deserialize the programming language object to/from an XML document.

Choosing schema-based XML documents as the persistent format for these data objects facilitates their storage in the Archive. Naturally, a real XML database would provide easy storage and query mechanisms for our XML data. Relational databases with an XML facade are also available, however, and usually offer superior performance. On ALMA, subsystem developers use a small-footprint XML database (the open-source Xindice) for development, while the initial production database will probably be an industrial-strength RDBMS such as dB2 or Oracle. In either case, the interface presented by the Archive to the application will be the same. This interface should also be the same if and when ALMA adopts some future XML or other non-relational database that proves significantly more performant than the relational solution.

Access to data in the archive is handled by Data Access Objects (DAO) that encapsulate query details and present the application with whatever interface is most natural to the application's language and implementation environment. Within the DAO, the query may be written as XPATH or XQUERY instructions; most of this will again be of no concern to the component developer.

5.4. Data modeling and code generation

5.4.1. Data modeling with UML

At a higher level of abstraction than XML is the Unified Modeling Language (UML). It brings behavioral specification into the modeling domain, where XML deals with data only. Object-oriented concepts such as inheritance are more easily expressed with UML. It is also true that the frameworks for generating binding classes from XML schema are not as widely available for C++ and Python as they are for Java. For these reasons we are beginning to migrate our data modeling activities from XML to UML.

5.4.2. Schema and code generation from the UML model

During development, the data models are naturally changing; we expect that they will change during ALMA's lifetime as well. New observing modes will require new parameters, new processing algorithms will require additional data, or data organized in a different way, and so on. If changes to the data model require major changes to the software and ripple through several subsystems, such changes will be risky and therefore made only reluctantly and with great effort. However, by automatically generating from the model the software needed to access it, we can simplify the process and minimize the disruption that a change to the model will cause to existing code.

Open-source tools exist that will accept a data model in a standard XMI format and generate user-specified code. By identifying, for example, entity classes in our model, we can instruct the generator to produce XML schemas that correspond to those classes as well as the entity classes themselves (in any language or languages that we choose), including factory, access and modifier methods. Additional behavior, characteristic of so-called wrapper classes, can easily be added. The procedure to generate the implementation artifacts from UML is shown schematically in Fig. 5.

ACKNOWLEDGMENTS

This unnumbered section is used to identify those who have aided the authors in understanding or accomplishing the work presented and to acknowledge sources of funding.

REFERENCES

1. G. Chiozzi, "The ALMA Common Software: a developer friendly CORBA-based framework," in *Advanced Software, Control, and Communication Systems for Astronomy*, M. H. Loew, ed., *Proceedings of SPIE* **5496**, pp. 5496–23, 2004.
2. M. Voelter, *Server Component Patterns*, John Wiley and Sons, New York, 2003.
3. H. Sommer, "Container-component model and XML in ACS," in *Advanced Software, Control, and Communication Systems for Astronomy*, M. H. Loew, ed., *Proceedings of SPIE* **5496**, pp. 5496–24, 2004.

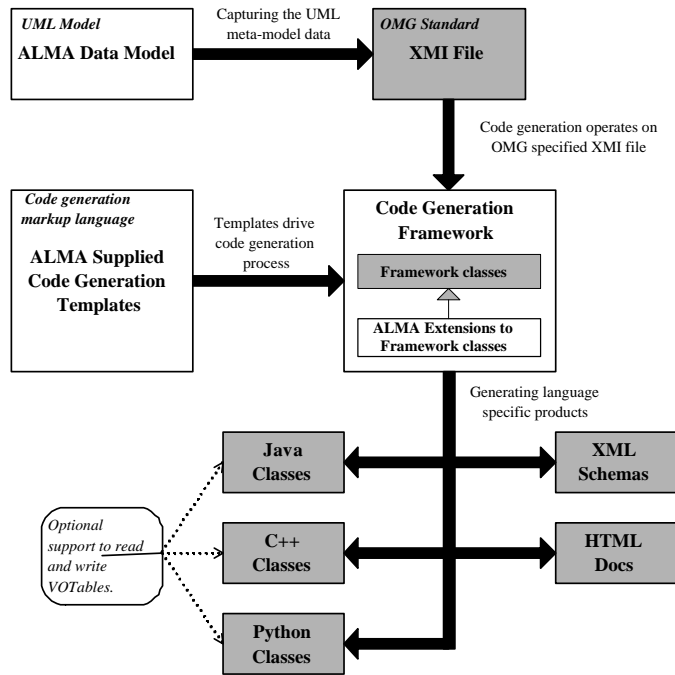


Figure 5. The process of generating code, XML schemas and html documentation from a data model defined in UML.

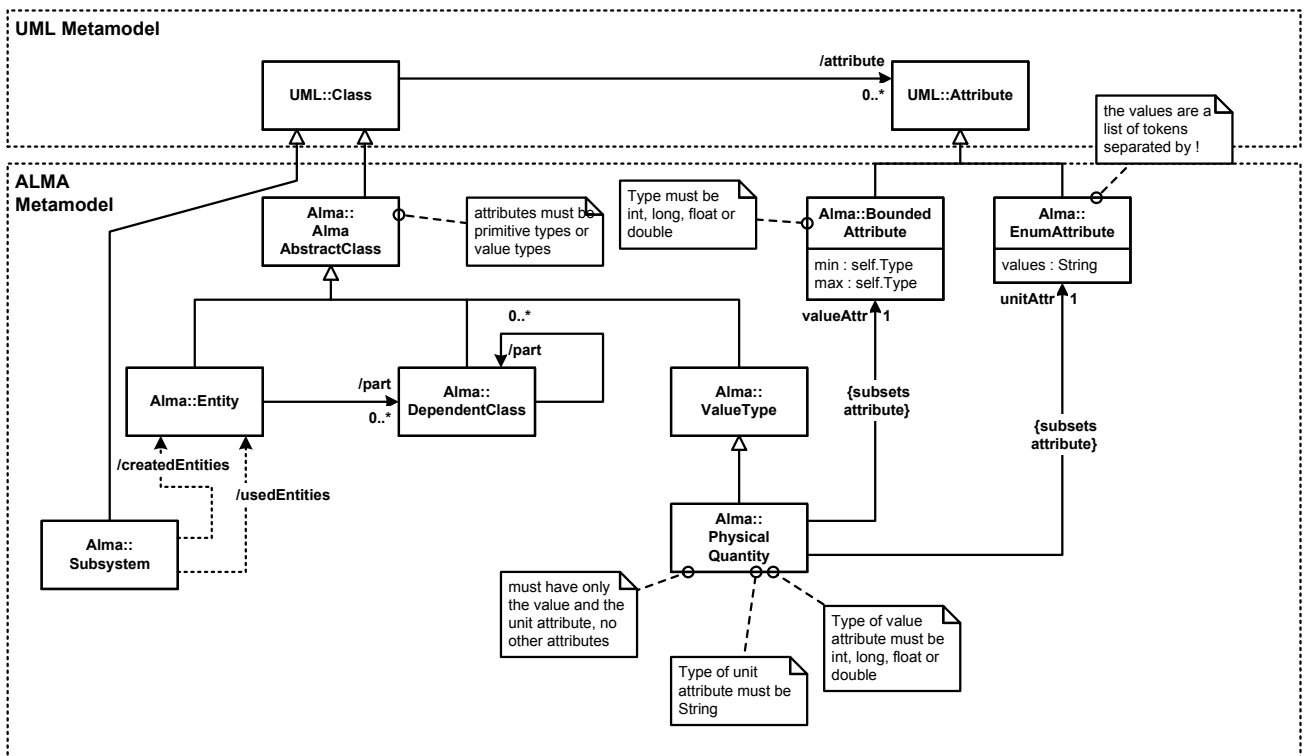


Figure 6. The ALMA Metamodel.