A GENERIC SOFTWARE INTERFACE SIMULATOR FOR ALMA COMMON SOFTWARE

D. Fugate[1], G. Chiozzi[2], A. Caproni[2], B. Jeram[2], H. Sommer[2], S. Harrington[3]
[1]University of Calgary, Calgary, Alberta, Canada, [2]European Southern Observatory, Garching, Germany, [3]National Radio Astronomy Observatory, Socorro, New Mexico, United States of America

ABSTRACT

The generic software interface simulator framework for Atacama Large Millimeter Array (ALMA) Common Software (ACS) provides ALMA developers with an easy means to create and configure the behaviour of interfaces that have been defined using Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL). ACS consists of a set of application frameworks built on top of CORBA and provides the glue which binds other ALMA software subsystems together [7]. In short, ACS provides an implementation of the component-container design pattern via CORBA. Using the simulation framework, one can choose to predefine the behaviour of a simulated component by embedding simple Python commands within a section of the XML-based ACS configuration database (CDB). The option to configure simulated components' behaviour at run-time is also a possibility using a provided graphical user interface (GUI) or application programming interface (API) executed within the context of an interactive Python session. Additionally, if the means above have not been utilized to setup the components' behaviour the framework will dynamically provide an implementation of the entire component with a randomized behaviour. This framework is especially useful to ALMA developers for two reasons. On one side it allows developers to test their own component, which is dependent upon other types of components that have been defined via IDL interfaces, but not yet implemented. On the other side this tool has proven itself to be quite valuable because it allows developers to connect clients such as graphical user interfaces to (simulated) components encapsulating hardware devices. Not only can the physical hardware devices be absent in this type of scenario, but the software representing the hardware need not be available either. The end result here is that clients and components can be developed and tested in parallel completely independent of each other. This paper discusses the design, implementation, and current usage of the simulator framework within ALMA software as well as future improvements to be made.

INTRODUCTION

ALMA is an international project to build the largest and most sensitive millimetre wavelength telescope in the world at Llano de Chajnantor, Chile.

ALMA Common Software

ACS is a software infrastructure for the development of distributed systems based on the component/container paradigm and also includes general-purpose utility libraries [4]. 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. The usage of ACS extends from high-level applications such as the Observation Preparation Tool that will run on the desks 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.

Component/Container Background

As mentioned previously, ACS is based on the component/container design pattern [5]. For those unfamiliar with the component/container model, it's defined as follows: a component is a piece of software that "lives" within a container yet is decoupled from the container. The container manages the lifecycle of components and provides them with a set of common container services. Examples of the component/container design pattern are Enterprise Java Beans (EJB), CORBA Component Model, and Microsoft .Net technologies.

The ACS group has implemented this model entirely in CORBA using IDL and provides a complete implementation of the container. Additionally, ACS has implemented a base component interface leaving other ALMA developers to simply extend this, adding methods useful to what they're doing along the way. These *ACSComponent*-derived IDL interfaces make up the core of ALMA software functionality and are shared between subsystems. As an all too brief example, one container deployed on a PC could have a scheduler component used to schedule observations and an antenna mount component used to move the antenna within it:

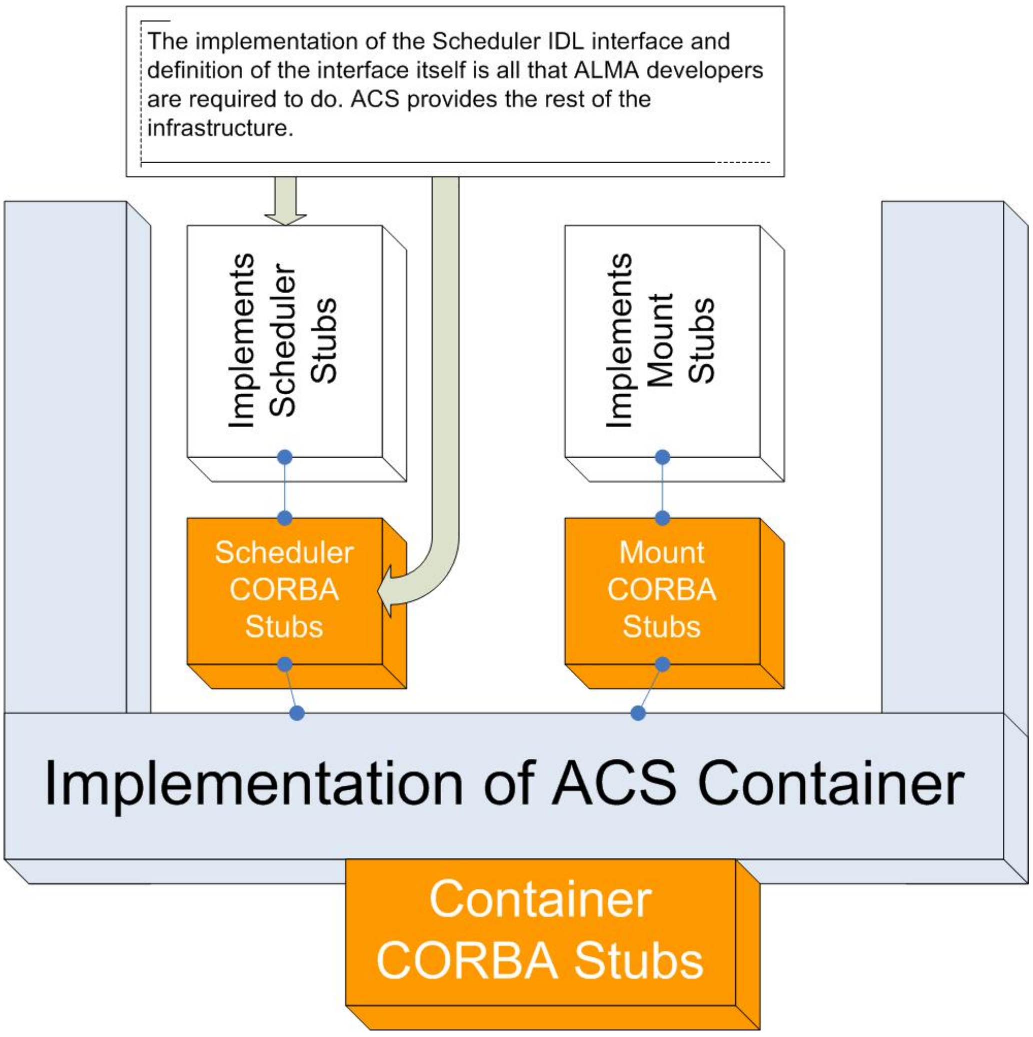

The implementation of the Scheduler IDL interface and definition of the interface itself is all that ALMA developers are required to do. ACS provides the rest of the infrastructure.
Implements Scheduler Stubs
Implements Mount Stubs
Scheduler CORBA Stubs
Mount CORBA Stubs
Implementation of ACS Container
Container CORBA Stubs

Figure 1: High-level overview of the ACS container/component model within ALMA software

SIMULATOR NECESSITY

In September 2003, during the normal ALMA software review process, we came to the conclusion that it was necessary to make available to the subsystems a simulation framework. The main reason for this was to support the Integration, Testing, and Support (ITS) team, responsible for the periodic release of the integrated ALMA software. Essentially our development is iterative and at any intermediate integration some pieces of code contributed by the various subsystems are only partially implemented. It also happens that the intermediate code does not perform according to specifications. It is therefore very difficult to get the integrated (but partial) system working. It is also very difficult to identify the subsystems responsible for bugs and work around them to proceed with the integration tests. It would have been much quicker to get the complete system exercised if the capability to fake the missing software functionality existed.

Due to the fact that only IDL interfaces can be seen by clients of components and not the actual implementations, we concluded that the most effective means of simulation for ALMA is at the component level. That is, it should be possible to specify to the container that the implementation for a given component is a simulated component factory. Also, because of the very nature of CORBA and IDL interfaces, clients using the component will never know they are not using the real deal. Component implementations are hot-swappable within the ACS framework.

REQUIREMENTS

During the ACS development cycle in which the simulator framework was created, concrete requirements came in and these were incorporated into the design:

- The simulator must be able to generate complete implementations of all IDL methods and attributes without input from the end-user.
- Enumerations, used largely by the Control subsystem to specify hardware states, will be fully supported.
- If an interface defines a CORBA Object attribute or a method that returns a reference to another CORBA Object, the simulator should then create the CORBA Object and be responsible for its lifecycle. Nil references are unacceptable.
- A simulated component should behave in the same manner as a real component. That is, simulated components shall have access to the container services and implement the non-IDL lifecycle methods. Additionally the simulator should take advantage of real object implementations where applicable.
- Users will have the option to specify a timeout value for methods. When the simulated method is invoked it will sleep for a period of time defined by the timeout and then return control.
- Read/write attributes should have some form of "memory" to store the value in if it is being set.
- It may be necessary to simulate the crashing of a component.
- A GUI shall be implemented allowing developers to set return values, timeouts, etc. for each attribute a component defines. If the developer does not set these parameters via the GUI, the infrastructure should then search the ACS CDB and if the values cannot be found there either, they will be generated on the fly.
- The GUI will be a dumb client for all intensive purposes. In other words, the intelligence of the simulation will reside in an API available to developers and the GUI will just make requests of the API. This will be used to facilitate the simulator's use in modular tests.

PRELIMINARY DESIGN

There were quite a few possibilities that were tossed around during the initial design phase of the simulator. While none of these proposals could meet the demands of ALMA on their own, the final design ended up being a melting pot of the following concepts:

- The simulator would be more of an interactive IDL interface compiler than anything else. In this way, the implementation of the ACS container would not have to be changed and a so-called simulated component would behave identically to real components. The downside is the developer would be constantly harassed with questions like, "*what values should the 'xyz' method return*". The simulator could also try to predict reasonable return values on its own.
- The developer would run the container interactively and manually set the return values for all component attributes/methods to be simulated. The main thing this has going for it is complete control over each simulated implementation and even the ability to dynamically change what methods do. The simulator/container could even be setup to import a user-defined module from the command-line which sets these automatically at start-up. The upside to this is minimal time is required to implement it. It would not even be necessary to subclass the CORBA skeleton classes because Python has some very useful functions for dynamically attaching methods to classes (the "new" package). It might even be possible for the end-user to simply invoke a function like *defineMethod(MOUNT_ACS_POA.Mount, "moveAntenna", "return 1.23")* meaning the *moveAntenna* method of the *Mount* IDL interface returns a constant double value.
- A subclassed container would be used which does all of the work for the developer. It would read a string from the ACS CDB to evaluate each method/attribute and return that. The output of

methods would almost certainly have to be static. Not as flexible as the previous two alternatives, this would be most user-friendly.

Decision to use Python

Since the simulator should be able to emulate components where the interfaces are not known ahead of time, a dynamic programming language seems like the logical choice for the implementation. Python is both dynamically scoped and typed, supports dynamic inheritance, and most importantly allows developers to dynamically redefine methods at run-time. ACS already provides a Python container which makes Python the ideal language for the simulator's implementation.

FINAL DESIGN

The final design for the simulator ended up being a combination of the three main contenders yielding an extremely powerful framework. In short, the accepted design allows developers to configure the behaviour of simulated components in four different ways – completely self-implementing components, configuration files found in the ACS CDB, a GUI, and an API. It has some of the following characteristics:

- Using the CORBA IDL Interface Repository (IFR), a CORBA service which stores and retrieves IDL, it is possible to accurately create method return values for the developer without their input.
- The Python container can be executed from an interactive Python session. This implies the developer can swap out entire method/attribute implementations with ease.
- Instead of simulating components at the interface level where all component instances of a given IDL type behave identically, we simulate at the named component instance level. This means that each simulated component of a given type can be configured to behave uniquely which is different from the three proposals.
- Using native Python methods, it is possible to dynamically create the implementation of any IDL interface. This implies simulation could indeed occur at the component level without making modifications to the container.
- Using native Python methods, it's possible to read method/attribute return values in the form of XML strings from the ACS CDB.

Configuration Database

The usefulness of defining simulated component behaviour before run-time is especially important for an extremely complex software system such as ALMA. For example, perhaps the end-user wants to find out what happens when the return value of some method is fixed. Productivity may be hampered because of the time spent changing return values each time the simulated component is created. For reasons like this, the characteristics of a simulated component can be retrieved from the ACS CDB if the user does not explicitly set them by some other means. The CDB entries are placed in the /alma/simulated/ section. The current implementation of the XML schema describing simulated components allows setting method timeouts among other things.

Application Programming Interface and Graphical User Interface

At times it can be quite useful to change the behaviour of a simulated method or attribute at run-time. For example, a regression test can require testing the client of a component with many possible return values for the same call performed. We therefore need a way to instruct the simulated component to behave in a specific, but different way for each call received by the client being tested.

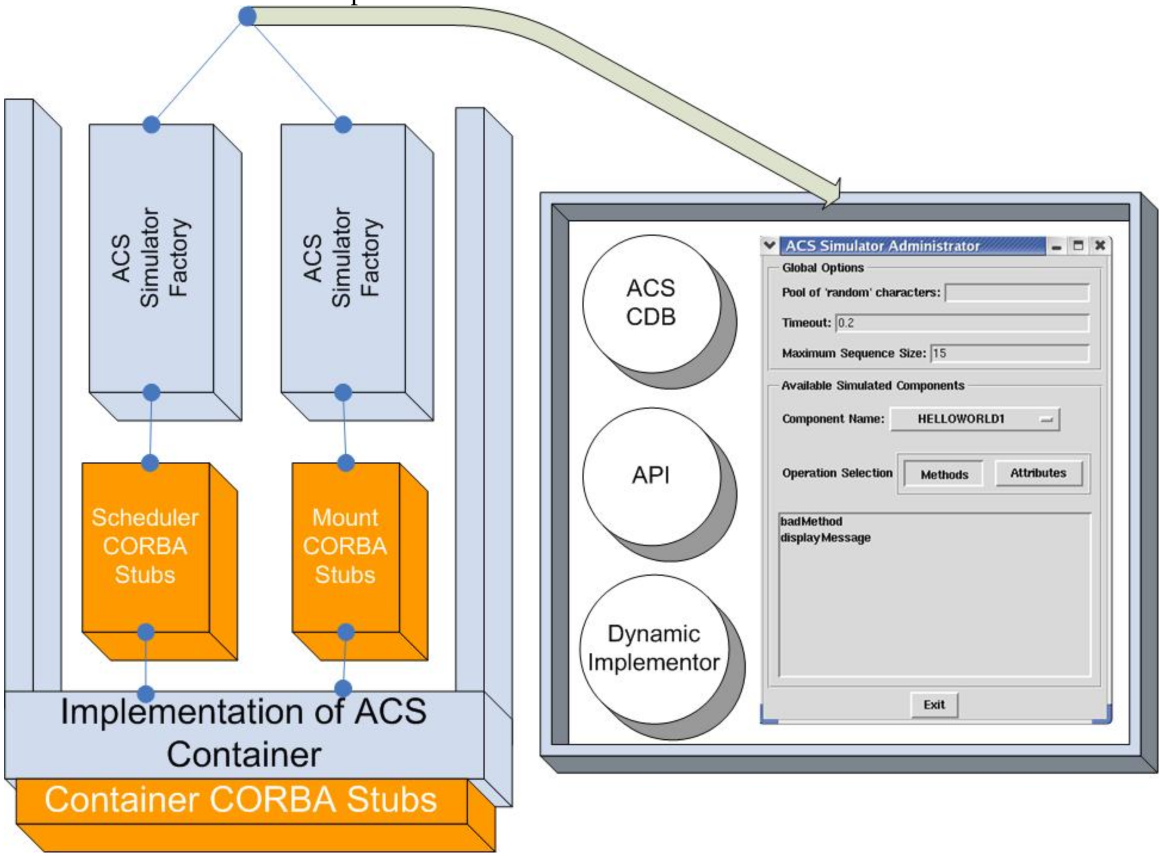
This is possible by running the ACS Python container within an interactive Python session and then manipulating the component(s) with an easy-to-use API. The API methods provided are generally in the format *setXyz* where *Xyz* is some configurable data. Some of the configurable items are:

- A standard timeout for all methods and attributes dynamically implemented by the framework
- The maximum sequence size for CORBA sequences
- Associating a new timeout, function to be executed, etc for a particular component's method or attribute

This API is also used by a GUI which is spawned by the first simulated component started within a container. The purpose of the GUI is to make the API much simpler to use for end-users who potentially have no programming background.

Self-implementing Components

Last but definitely not least, a simulated component will implement its own methods and attributes completely autonomously if the end-user fails to utilize any of the other three mechanisms to modify the behaviour of simulated components.



Pool of 'random' characters:
Timeout: 0.2
Maximum Sequence Size: 15
Component Name: HELLOWORLD1
badMethod
displayMessage

Figure 2: Diagram depicting how simulated components function within a container

RESULTS

The simulator framework is currently being used by a few ALMA subsystems. The Executive team is working on creating an operator GUI which in turn is a client of many different components created by other ALMA subsystems. The problem Executive is experiencing is that while all of the IDL interfaces describing these components have been implemented, the components themselves have not been completed. In situations like this, the simulator is proving itself to be an invaluable tool. A few other groups have begun using the simulator as well, with the primary use intended to be decoupling their modular tests from the implementations of components provided by other subsystems.

CONCLUSIONS

Overview from a User's Perspective

From an end-user's point of view, the simulator can be downright trivial or fairly complicated to use depending upon the functionality desired. Right out of the box, all one has to do is modify a configuration file describing the component to say it will exist within a Python container and then change the implementation library name to that of the simulator component factory. Simply performing the steps listed above gives the user access to a component which automatically implements all methods its interface defines and returns fairly reasonable values. For those demanding a more realistic simulation, they can provide their own logic in the form of XML configuration files or input this information at run-time using the GUI and/or API.

This approach makes it very easy to implement simple behaviour, but we have seen from our users that there are many cases where we have complex simulation needs. For example, we might need to link the value of attributes to the current value of other attributes. Consider the right ascension and declination in the sky of a telescope; they depend on the azimuth, elevation and time. Implementing these and more complex relations with snippets of Python code embedded in the XML definition files is rather complex and difficult to debug. The main request from our community is to make such complex simulations easier to implement.

Future Improvements

There have been a number of proposed enhancements for the simulator that are currently being implemented. First, most people will agree the framework's major shortcoming is that the XMLs do not support complex Python language constructs such as loops. End-users choosing to utilize the CDB are limited to very simple Python operations and code that depends upon the orientation of white space is out of the question. Support for this will be added with the release of ACS 5.0 this Fall and we will also include the capability of adding methods to the simulated component from the CDB. Aside from what's mentioned above, the following improvements will be made:

- Sophisticated support for receiving/sending events will be added
- An area in the CDB will be added in which users can define XMLs defining the behaviour of all IDL interfaces of a given type

REFERENCES

[1] OMG. "CORBA 3.0." IDL Syntax and Semantics. May 26, 2005. <http://www.omg.org/technology/documents/formal/corba_2.htm> (15 September, 2005).

[2] OMG. "CORBA 3.0." Interface Repository. May 26, 2005. <http://www.omg.org/technology/documents/formal/corba_2.htm> (15 September, 2005).

[3] Beazley, David. Python Essential Reference. Indianapolis: New Riders, 2001.

[4] G.Chiozzi. ALMA Common Software: a developer friendly CORBA based framework. Paper 5496-23. Glasgow, Scotland: SPIE, 2004.

[5] H.Sommer. Container-component model and XML in ALMA ACS. Paper 5496-24. Glasgow, Scotland: SPIE, 2004.

[6] D.Fugate. A CORBA event system for ALMA Common Software. Paper 5496-68. Glasgow, Scotland: SPIE, 2004.

[7] G.Chiozzi. ALMA Common Software (ACS): status and developments. Geneva, Switzerland: ICALEPCS, 2005.