SOFTWARE INTEGRATION AND TEST TECHNIQUES IN A LARGE DISTRIBUTED PROJECT: EVOLUTION, PROCESS IMPROVEMENT, RESULTS

P. Sivera, M. Pasquato, ESO, Garching, Germany

Abstract

The Atacama Large Millimeter Array (ALMA) is a radio telescope that is under construction in Chile. The software for the project is being developed by the Computing Integrated Product Team, (IPT), which has the responsibility of realizing an end-to-end software system consisting of different subsystems, each one with specified development areas. Within the Computing IPT, the Integration and Test subsystem has the role of collecting, building, testing the software produced and preparing releases. In this paper, the complexity of the software integration and test tasks is analyzed and the problems due to the high geographical distribution of the developers and the variety of software features to be integrated are highlighted. Different implemented techniques are discussed, among them the use of a common development framework (the ALMA Common Software or ACS), the use of standard development hardware and the organization of the developers work in Function Based Teams (FBT). Frequent automatic builds and regression tests repeated regularly on Standard Test Environments (STE) are also routinely used. Advantages, benefits and shortcomings of the adopted solutions are presented.

ALMA PROJECT OVERVIEW

The ALMA radio observatory is being built with the participation of institutes distributed over three continents, namely Asia, Europe and North America. There are more than 14 different sites involved in the software development, grouping a total number of, at present, 65 FTEs. Since not all developers work full time for the ALMA project, the actual number of people we have to deal with is much higher, often by a factor of two. Within the Computing IPT, the ALMA software developers are partitioned in subsystems. A subsystem is:

- a group of people and its leader, who have to develop a portion of the ALMA Telescope System
- an area in our configuration control tool (CVS) repository where the software developed within that subsystem is checked in.

There are also subsystems whose primary scope is not to develop but, for example, to design the overall software architecture, to define the software engineering practices or, finally, to take care of the integration and test of the whole software produced. The Integration, Test and Support (ITS) subsystem is in itself spread out around the world: two testers in Japan, 4 in Germany, 2 in the United States, (in total 5.1 FTEs) each of them with different experiences and skills, so that the first integration work had to be done among the members of the integration team! A second challenge for the ITS subsystem is the variety of the software to be integrated and tested, which includes the Graphical User Interface (GUI) for the astronomers to prepare an observation proposal, the software to actually execute the observation, the control software characterized by real-time behaviour and used to command the antennas, the receivers and the correlator, the tools to produce data and reduce them, the archive system, the telescope operator GUIs.

STANDARDS AND TOOLS

One of the first decisions which has been taken in within the Computing IPT was to define a standard set of tools accompanied with common rules to be followed at every site. ESO had already successfully adopted such a strategy in other big projects, such as the control system for the Very Large Telescope (VLT) project and ALMA has benefited from the VLT project expertise. First of all, the supported platforms have been identified, for both high level and real-time developments. Then, the following development tools and infrastructures have been adopted:

- Usage of the Alma Common Software (ACS) at every site. ACS has been the subject of other papers presented at ICALEPCS, for example see [1]. Briefly, it consists of a set of applications built on top of CORBA. The purpose is to simplify the development in each of the supported languages adopted in the ALMA project, providing the developers with common services like logging system, error handling, alarm generation and monitoring, configuration database, archiving and, at the same time, hiding the complexity of the CORBA middleware. The basic concept is a component/container model, where each container implemented in one of the ALMA development languages handles the lifecycle of one or more components implemented in the same language and mediates the services listed above. Being thus relieved from system programming concerns, the developers only need to write the application components.
- Usage of a standard development environment. ACS not only provides the necessary tools to develop, but also a set of environment variables already prepared in order to access those tools in the correct way. All these environment variables are put in a single file, in a standard location and every developer must source that unique file to be

able to work correctly in the ACS/ALMA environment.

- A common tool for version control (CVS): the same repository is used for all sites participating in the ALMA project; the software under every subsystem is organized in packages and every package is made up of modules, each with a predefined directory structure. Developers are responsible for one or more software modules. The modules must be prepared using templates which create the correct set of subdirectories of the module. And those directories are mandatory so that the ALMA build system can work properly.
- Common rules to build and install every module. A common Makefile (called acsMakefile) based on gnu make is provided together with ACS and every module must contain a Makefile which includes the acsMakefile. The acsMakefile defines the rules to build in the different supported development languages used in the ALMA project. In this way, the make rules (including compiler and linker options) are the same for every developer.
- A standard test environment (STE) has been prepared. The STE consists of a set of machines installed by the ITS team according to very well established standards and conventions, which span from the number of rpms installed on the machines to the users' definition and environment. It is primarily used by ITS to integrate and test the overall ALMA software, and it is being installed at the operational site as well, where it will work with the real hardware. At the development and integration sites, it should be considered as a model which tries to reproduce the operational environment, using software simulators when the lack of hardware has to be overcome. Developers are also invited to test their software releases against the STE, in this way the peculiarities due to the personal development workstations are eliminated.
- A standard tool called "Tool for Automated Testing", (tat) developed at ESO, is also being provided with ACS. This is a framework which helps running a test suite with only one command and reports the result of the test suite in a simple and clear way, printing to the standard output the word PASSED or FAILED. It is a general rule that ALMA developers must supply a (unit) test suite for every software module they develop. The tool is also being integrated with the different Junit, Pyunit and C++unit frameworks and recently we have begun integrate it into the DejaGnu project, thanks to the cooperation with colleagues from the VIRGO project.
- Finally, since the very beginning of the ALMA project, a tool for bug tracking (Jira) has been adopted, so that every software problem or change request is kept under control.

ALMA SOFTWARE RELEASE CYCLE

A major ALMA SW release is prepared once a year, using the subsystems releases delivered the 30th of September every year. After the subsystems deliver their code, ITS has got two months time to deliver the integrated ALMA SW release. A minor ALMA software release is prepared in the same way 6 months after the major release.

How do the different subsystems deliver their software to the integration group? Which integration strategy is most suitable for a project like ALMA? And how can we ensure that the integration of such diverse software will be successful in the time frame allocated to the integration activities? We tried to answer these questions already in the very preliminary phase of the project, even before the first lines of code were produced. This does not mean that we got it right! But the fundamental idea was to try to avoid a big bang integration, a few days or weeks before the major or minor release. We wanted to approach as much as possible the concept of continuous integration.

Continuous Integration – Monthly Integration

One of the cornerstones of the project's development philosophy was to enable the ITS team to run the software end to end (involving all, or almost all subsystems) from the very first integration. We first tried to find a formal way to get the software delivered from the different developers spread out around the world. We wanted to be able to retrieve all the ALMA software from CVS in an automatic way and keep track of the different integrated software baselines at every specific date in the software development cycle. We required the subsystems to tag their software in CVS by the end of each month with a monthly tag according to some established naming conventions. ITS was tasked with building the software and producing an end-to-end running system with the same periodicity, once a month. In this way, we could approach the release deadline with the hope that the monthly integrations would have uncovered most of the problems.

There are two basic shortcomings with this approach. One external problem was due to the fact that the integration of the software was just at its inception, therefore we had to face, at every monthly integration, at least for the first eight months, a huge quantity of build problems, so that the time left to run integration tests was always too short and often we were approaching the following monthly integration without being able to really finish the previous one. Another big problem was that every subsystem was working in isolation; developers within a subsystem were supposed to develop a certain number of features according to a formal software plan by the release deadline, but there was not enough communication across subsystems. This had as a consequence that the integration was always very painful and most of the integration time was spent in trying to sort out miscommunication problems.

We understood very quickly that it was necessary to put together people from the different subsystems, organizing face to face meetings at least at every major and minor release. Still, the release integrations for the first couple of years were barely managing to deliver a partially running system.

Continuous Integration - Function Based Teams

To improve the quality of the software delivered and to provide our Antenna Test Facility (ATF)* with some usable software, we adopted a completely different approach which has now been in use for almost two years. At the beginning of every release cycle, the needed functionality is identified: for example, by a certain project deadline (read: release), the antenna at the test facility should be able to do optical pointing, holography and should work with a monitor database system. The development of these three overall features of the ALMA software requires contributions from different subsystems. We then organize Function Based Teams (FBTs), (proposed by our colleague Dave Clarke from ATC, UK) consisting of developers from the involved subsystems as well as a leader whose main goal is to bring the FBT to successfully deliver the planned feature on schedule. Every FBT normally works on a specific branch, not on the CVS main trunk, so that this basic principle is respected: the HEAD of the software in CVS should always be stable.

The life time of an FBT is normally of the order of two months. There should be in average 3-4 FBTs per release cycle. In every group, at least one representative from ITS takes care of writing test plans and test cases already during the development of the feature. At the same time, the software of the branch is built and tested, basically every night, in the STE.

During the life of a FBT, there are some milestones that have to be respected, when the software produced should be delivered and tagged following new agreed naming conventions for the branches. These milestones are: the end of the development of the feature, the phase of the merge back to the trunk of CVS and the validation of the software after the merge, which is done by ITS only, running all available regression tests.

This approach based on FBTs has been very successful and the major benefit for ITS is that we can really follow the development of the feature and immediately think about how to test it. The only difficulties are in maintaining the deadlines for the delivery of the functionality. Very often compromises have to be made, like dropping some functionality for a specific release or adjusting the release cycle to the deliveries of the FBTs, maybe delaying the release date by few weeks. The organization of face to face meetings is still necessary for properly concluding the work of a FBT, and sometimes more than one of such meetings per FBT has to be planned.

* The ATF shares the site of the Very Large Array (VLA) in New Mexico where the first ALMA prototype antennas have been assembled

SUMMING IT UP

The introduction of the FBTs was a major step forward in the way the release cycle is organized and greatly helped us to integrate more smoothly, and deliver better tested and more usable releases.

Testing the integrated system end to end from the start is a practice that has both advantages and disadvantages. It is probably good, in particular when dealing with object-oriented software, to immediately test the interfaces among the components. This has the drawback, however that the testing effort is done on an ever changing system, so that much time is spent in consequently changing the test cases.

The adoption of an independent test group is considered as an advantage and a good practice, but a peculiar mistake done in ALMA was not to require development skills for the testers. Instead we have learned that to be able to write better and automated tests, we do need to have knowledge in the languages used in the ALMA project.

Referring to the levels of testing maturity identified by Boris Beizer [2], here is where we are:

- Level 0: there is no difference between testing and debugging. This still happens too often, in particular in many of the face to face meetings during the development phase of a FBT.
- Level 1: the purpose of testing is to show that the software works. We normally reach this level during the validation phase at the end of the work of a FBT. This also means that we do not manage to do more tests than proving the software works for the functionality it was supposed to deliver. The reasons are: time constraint vs number of features to test; often the merge phases are not that smooth and leave a lot of open problems that we have to face during the validation phase.
- Level 2: the purpose of testing is to show that the software does not work. For the last few release cycles, we have begun to think about negative tests as well.
- Level 3: the purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value. Only recently we began to try to really understand the quality of the software under test and to evaluate it from the point of view of its deficiencies and impact on the users if the system is shipped in its present state.

There exists also a fourth level which focuses on making software more testable from its inception. This level and the rigorous adoption of the test maturity model (TMM) should be considered like a desirable evolution of the testing practices within the ALMA project.

REFERENCES

[1] G. Chiozzi et al., “The ALMA Common Software ACS - Status and Developments”, ICALEPCS 2005.

[2] L. Copeland, “A Practitioner’s Guide to Software Test Design”, Artech House Publishers, 2007.