New Architectures Support for ALMA Common Software: Lessons learned

Camilo E. Menay[a], Gabriel A. Zamora[a], Rodrigo J. Tobar[b], Jorge A. Avarias[c], Kevin R. Dahl-Skog[a], Horst H. von Brand[a] and Gianluca Chiozzi[b]

[a] Computer Systems Research Group, UTFSM, Valparaíso, Chile;
[b] European Southern Observatory, Garching bei München, Germany;
[c] National Radio Astronomy Observatory, Socorro, NM, USA

ABSTRACT

ALMA Common Software (ACS) is a distributed control framework based on CORBA that provides communication between distributed pieces of software. Because of its size and complexity it provides its own compilation system, a mix of several technologies. The current ACS compilation process depends on specific tools, compilers, code generation, and a strict dependency model induced by the large number of software components. This document presents a summary of several porting and compatibility attempts at using ACS on platforms other than the officially supported one. A porting of ACS to the Microsoft Windows Platform and to the ARM processor architecture were attempted, with different grades of success. Also, support for LINUX-PREEMPT (a set of real-time patches for the Linux kernel) using a new design for real-time services was implemented. These efforts were integrated with the ACS building and compilation system, while others were included in its design. Lessons learned in this process are presented, and a general approach is extracted from them.

Keywords: ALMA, ACS, CORBA, framework porting, multi-platform

1. INTRODUCTION

The creation of full cross-platform compatible code is not trivial, even when this is a problem that has been studied before. The complexity is high, and it is even worse when the framework itself does not consist in a few set of files, but of a large amount of code written in several languages, lot of scripts, building patterns, testing tools, code generations, and much more. The work presented in this document is focused on such a framework: ALMA Common Software (ACS). ACS is a distributed control framework based on CORBA intended to be used as a standard for all the software and subsystems that conform the ALMA project.[1,2] Thanks to its open LGPL license, ACS can be used in any other astronomical project, particle accelerators and also allows the contributions of many users and researchers, thus making it possible to change the source code if needed for specific needs.

The official release of ACS is only supported on Scientific Linux and Red Hat Enterprise Linux. This limitation can be a disadvantage when thinking on using ACS for other projects, as all the tool development environments and expected hardware are limited to the the ones supported by the mentioned operating system flavors. Since the first releases of ACS, several efforts have been made to open it to new platforms and architectures, but due to different constraints this has not been always an easy task. From the most important strict limitations with some base tools software versions (i.e.: `GCC` and `glibc`), to the small or non existent support on other completely different architectures, like embedded systems or the widely used Microsoft Windows. Since ACS has always been focused to a few Linux distributions, and also because all of the development and changes made in the meantime, this relation is now stronger, making it harder to break. As explained in the following sections, ACS is built on several base operating system tools, and some of its external products have low level communication with the underlying platforms, making the task of porting ACS to a non-officially architecture or platform non trivial .

These constraints and difficulties have been addressed by the ALMA-UTFSM Group during a long timespan, which has generated a large amount of knowledge and experience on how we can extend a large complex framework such as ACS to new uses. It is also important for frameworks like ACS, and projects like ALMA, which have a long expected life-time, must be constantly adapting to technology changes; otherwise, they will become quickly obsolete, or will become increasingly hard to support.

In this document we want to share the lessons, experiences and conclusions that we have acquired during this work. These have been (and will be) important to achieve the objective of adding support to new architectures and platforms, not limited to ACS, but also to any other large software framework.

1.1 The ALMA Common Software

ALMA[3] is a radio-astronomy project conducted by the European Southern Observatory (ESO), the National Astronomical Observatory of Japan (NAOJ) and the National Radio Astronomy Observatory (NRAO). It will consist in 66 antennas, located on the Chajnantor plateau (in the North of Chile), that will be used simultaneously to make radio-interferometry, being able to observe and study phenomena never studied before.

This project implies a large number of challenges, starting from the geographic location: antennas will located at 5000 [m] above sea level, while the Operations Support Facility (OSF) is being based at an altitude of 2800 [m]. This is the place where main operations are carried out, and people will not be present frequently at the high site.

To cover the distributed nature of the problem, ALMA Common Software[2] (ACS) has been developed. ACS is a distributed control framework developed to be used not only by ALMA, but also by any other project that might want to use it. ACS is based on CORBA technology, and includes open-source ORB implementations for C++, Java and Python (TAO,[4] JacORB[5] and OmniORB[6] respectively). It is distributed under the LGPL license, allowing it to be used in other projects, and facilitating its development by including people outside the project.

Apart from being based on CORBA, ACS uses the Container/Component architecture for software development/deployment. Developers write *Components*, which in turn are run in *Containers*. Containers can run on different hosts, and are available for the three mentioned programming languages. The entire set of running containers that conform a given deployment is orchestrated by a *Manager*. There are also other services and facilities, such as a centralized Configuration DataBase (CDB), Logging Service, Alarm Service, Notification Channels, and more.

ACS provides a standard environment which must be loaded for getting access to all its components. When developing under ACS, this environment also provides a set of `Makefile`s, which are the main actors in the ACS building system. This environment is heavily based on UNIX tools, and currently ACS is only officially supported on Red Hat Enterprise Linux and some versions of VxWorks.

2. WORK DESCRIPTIONS

The following is an explanation of the development of different tasks performed in this work, detailing methodologies, tools and techniques used, in addition to pointing out the disadvantages and degrees of success in these tasks

2.1 Porting ACS to different Linux distributions and newer GNU compiler versions

A first attempt to port ACS to a different Linux distribution was done back then when ACS version 3 was released (2003). This effort was done by the Observatorio Astronómico Nacional (OAN), Spain*. The porting was done using Debian 3 as Linux distribution, which is quite different from Red Hat Linux. The main problems were the compatibility of the GNU C compiler (GCC) suite used in Debian (version 2.95). In contrast, Red Hat Linux 7.2 (the supported platform for ACS at that time) used GCC 2.96. This version of GCC was an experimental branch, a pre-release of GCC 3. The two GCC versions generated incompatible binaries and the code gave strange errors when compilation was tried on Debian 3.

When ACS 4 was released (2005), it used GCC 3.2 for building. This time the compatibility between ACS 4 and Debian 3.1 was very straightforward, because the binaries were compatible between Red Hat Linux 8, the Linux distribution supported at this time, and Debian 3.1. To use ACS it was only required to install the binary files directly, fix a couple of symbolic links and set the environment variables needed by ACS properly. This situation was maintained until ACS 5 (2006).

*`http://almasw.hq.eso.org/almasw/bin/view/ACS/PortingAcsToDebian`

By the time of the ACS 6 (2007) release, the computers' hardware had changed, sometimes incompatible with the Linux distributions that ACS supported. For ACS to run on new hardware, it was necessary to use a newer Linux distribution built with the GCC 4 compiler, or port the needed drivers to the kernel used by the official supported distribution. This task was not always possible, and, as had happened before, GCC 3 generated binaries incompatible with GCC 4. Therefore, this time it was necessary to port the entire ACS code to the new compiler. The ALMA-UTFSM Group[7] had previously done a successful ACS porting from Red Hat Enterprise Linux (RHEL) 4 to Fedora Core 6 (base for RHEL 5).[8] During this task the major problem was the change of the C++ compiler from one distribution to another: RHEL 4 uses GCC suite version 3.4, while Fedora Core 6 used GCC version 4.1. This version difference caused that the source code of ACS could not be built directly, basically because the newer version of GCC is more strict with the syntax and handling static global variables. This new compiler version also added new features which were, in some cases, incompatible with the source code of ACS written for GCC 3.4. Thanks to the work done by the ALMA-UTFSM Group, ACS now can be used on different Linux distributions that use newer GCC versions. Currently, ACS runs on Ubuntu Linux 7.10, 8.04, 8.10; Debian Linux, OpenSuse Linux 10 and 11. The official support for ALMA has started using, in testing and production environment, RHEL 5.3 with ACS 8 (2009), which includes the changes done in this porting process.

After the porting process of ACS to the version 4.1 of the GCC compiler, a second porting process was undertaken by the ALMA-UTFSM Group to compile ACS under the 4.3 version of GCC. This newer version is even stricter that its previous version in the syntax of the code, specially respect to omitted `#include` statements and namespace specifications. This second porting process resulted in a set of patches not only for the `CommonSoftware`, but also for the `ExtProd`, `Kit` and `Tools` packages, and enabled the usage of ACS in even newer Linux distributions, like Fedora 10 and Ubuntu 9.4. These patches have not been yet included in the official ACS release.

An effort to port ACS to Windows XP was conducted in the past by Observatorio Cerro Armazones (OCA) [†] during ACS releases 2 and 3. The main goal was to use a subset of the C++ functionality of ACS, an ACS C++ simple client, together with LabVIEW 7.1. To port the required parts Cygwin was used as a Linux compatibility layer, avoiding most of the complications of porting ACS code from the POSIX API to the Windows Native API. But this also introduces incompatibilities with the loading of shared libraries when ACS is used together with LabVIEW. Finally, some of the issues were fixed or a workaround was found for them, providing a functional subset of ACS on Windows.

2.2 Porting ACS to Windows

One of the latest efforts undertaken by the ALMA-UTFSM Group has been to port the ACS framework to the Microsoft Windows XP operating system. Although some work has been done in this direction, there is still much work left to do, and that is being analyzed at the time of writing. The current finished work includes the porting of the most important runtime scripts and utilities, part of the compilation system, and the Java-coded portions of the framework. The Java-coded portion of the framework was tackled first because, thanks to Java's portable nature, it was the easiest starting point. Please refer to[9] for more detailed information about the work referred in this section.

Porting ACS to Windows opens a huge range of opportunities that otherwise would be unreachable by only having ACS running on Linux. It would allow, for example, to use Windows-only available libraries and/or tools, as the .NET framework or some features of LabVIEW that perform better under Windows. Simultaneously, it would open the possibility to communicate with a number of devices that have only drivers for Windows. Finally, a considerable number of users might feel more comfortable using ACS under Windows to keep their work environments with them.

To perform such a task, a UNIX-like environment was urgently needed. For this reason we used Cygwin,[10] because it offers implementations of all the most important basic programs needed to work in a UNIX-like environment (e.g., `bash`, `ksh`, `grep`, among others). These tools are needed by the compile system (based in `make` and scripts) and the runtime environment of ACS. This way, we avoid rewriting the entire runtime

[†] `http://almasw.hq.eso.org/almasw/bin/view/ACS/PortingAcsToWindowsXP`

environment and compilation system that comes with ACS, and we can reuse it with some minimal effort. For compiling the Java code, we installed the Windows version of the Java Virtual Machine. Some basic external products are also needed by ACS, such as Python, TAO and JacORB. We compiled these using the Microsoft Visual C++ Studio compiler, in the case of C/C++ sources, and with the Java compiler, in the case of Java sources.

ACS makes all its environment available by sourcing a large file with definitions of paths, libraries and environment libraries. This was the very starting point of the work. A successful source of this file introduces the user to the ACS environment, where all the tools are available, and compilation can take process. Nevertheless, this environment is heavily based on scripts, which sometimes had some UNIX-specific values set into variables. An example of this is the `CLASSPATH` separator: While on UNIX it it usually a colon (`:`), in Windows it must be a semi-colon (`;`). Cases like these, which repeated along several scripts, and even through the `Makefile` infrastructure, were solved while maintaining the support for UNIX-like environments.

The complete compilation process was mainly: Base tools, then external products, Kit and finally ACS Common Software. As we were using Cygwin as environment, all the base tools were already covered (e.g., `bash`). As stated before, the external products were compiled using the native Visual C++, as most of the source code ships with a Visual Studio solution file. For the Common Software, we had to check the dependencies as several modules must follow an strict order. This information can be found in the `Makefile` and needs to be adjusted manually. As Java is multi-platform, we did not expect too much trouble, but ACS being such a large framework, there are lots of scripts, environment variables, `Makefile` and others, that must be correctly set to work on a specific platform. Also, ACS compiles the CORBA IDL files for the three supported languages, but as we are only trying to compile for Java, we used some flags that the ACS `Makefile` provides for this cases, (i.e. `MAKE_ONLY=Java`).

After fixing a large number of files and adding support to the Windows platform in the needed scripts, as the nature of ACS is to be a distributed framework, the next step was to test if our development machine with Windows XP installed was integrating correctly with the Linux ones that had the entire ACS framework running. In particular, it was necessary to check if the Java side of ACS was communicating correctly with the ORB services that are in charge of all the distributed tasks, and with other important entities like the CDB DAL and the Manager. The test performed was to run the ACS fundamental services on a main computer. This is the same approach used for Linux machines not running ACS, but wanting to connect to a main computer and execute Containers or applications that need ACS. Due to the ACS architecture it was possible to run a Java container on a Linux machine and use the Java components there. If the porting process was successful, then the same should work with a Windows machine using our ACS port and running Java only containers.

Aside from containers, ACS ships with a set of useful GUI applications. Most of them are written in Java and present important features that are heavily used, like *jLog*, an application that allows to check all the logs of different systems with level filtering and *objexp* useful to check and try the different components that are running in a given ACS Manager.

In figure 1, you can see the actual test performed. It shows ACS working in a distributed environment, several machines working together with Linux and MS Windows communicating with a main computer that is running the more important services needed: ACS Manager, the ORB services and DAL. It is possible to see that there are several containers running for different languages. Running a container means that you can execute all the components written in that specific language and also in the case of Java, if we have complete support is possible to run all the useful GUIs.

The correct functioning of the environment that figure 1 shows validates the porting process and its results, as everything is working correctly as it would in a Linux machine. It was possible, for example, to follow the logs of the ACS Manager and all the Containers from different machines running there with *jLog*.

Finally, the concrete result of all this work was a set of patches that we applied locally to our source code and that we then compiled once again in Linux, to check that our patches were not breaking anything. We also produced a simple installation guide [‡] that allows a user to download a package with the binaries already

[‡] `https://csrg.inf.utfsm.cl/twiki4/bin/view/ACS/AcsWindowsInstallationGuide`

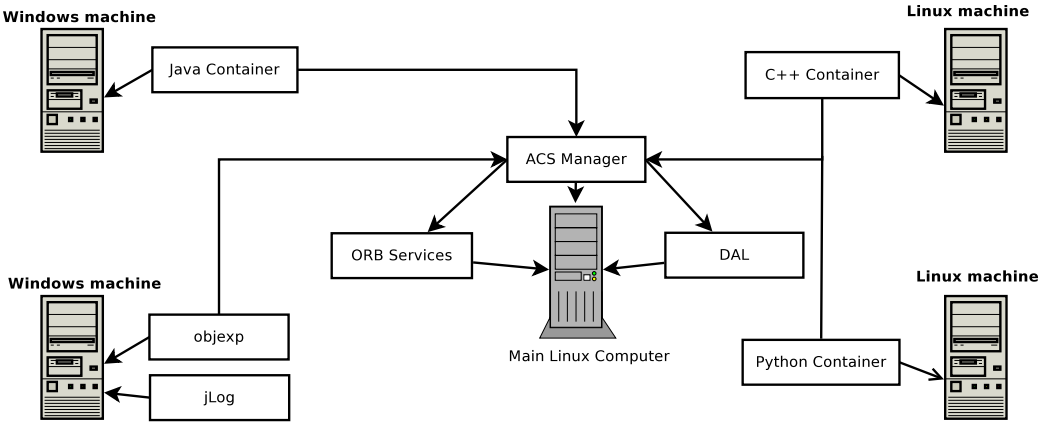

Figure 1. ACS in a distributed environment

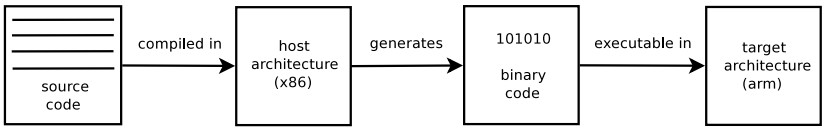

Figure 2. Cross-compilation process overview

compiled and with some short instructions to reproduce the Java ACS environment ready to work. Since our changes are compatible with the original ACS sources, and have been tested also against Linux compilations, applying them to the ACS official sources is very possible. This work has been already been discussed with the ACS development team, and will be undertaken in the following months. Also, C++ and full Python support is being currently studied by the ALMA-UTFSM group at the moment of writing.

2.3 ACS ARM Porting

One of the ALMA-UTFSM Group research tasks was to port ACS to a different architecture than the one on which it was developed. The election was the ARM architecture, because of its great use on the embedded systems world. This work had several objectives: Test the flexibility of ACS's code, check intercommunication between different architectures in a distributed environment, add to ACS the capability to be deployed on small machines, making them part of a greater distributed control system, and taking advantage of their capabilities, like minimum use of energy and real-time features.

Due to the huge number of lines of code that ACS comprises, the time it would take to compile the software on an ARM machine would be problematic. Actually, in some cases it could be impossible due to restrictions of the hardware, since most ARM-based machines have very limited resources. An alternative to this is to use tools that allows compilation of code for a given architecture A, but on a different architecture B, generally with more processing power. This technique is called cross-compilation. In the case of ACS, we aimed to cross-compile ARM code on a normal x86 computer (figure 2).

To carry out a cross-compilation it is necessary to provide a software environment within the host architecture which allows all the processes that are usually run on the target architecture to run transparently and smoothly on the host architecture, thus being able to produce and even test run code of the target architecture within the host computer. Such an environment consists of compilers, libraries, and tools, which together provide an abstraction of the target architecture (figure 3).

In the following subsections we present several cross-compilation environments and tools which were tested, and with which we obtained different results at different levels. For testing purposes we have an ARM embedded machine with an XScale PXA270 processor (ARMv5).

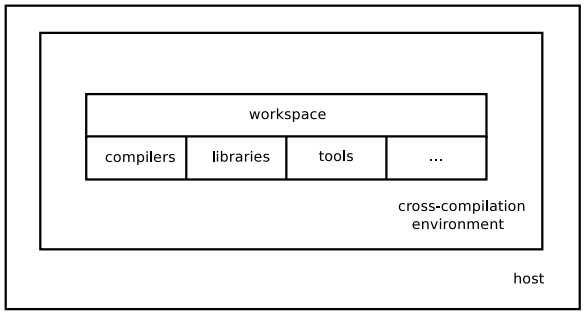

Figure 3. Cross-compilation environment

2.3.1 GNU-ARM toolchain and libc for ARM

The first approach for compiling the C++ modules of ACS for ARM consisted on making use of the GNU-ARM GCC-4.1 toolchain,[11] which includes C and C++ cross compilers. GNU-ARM toolchain source code was compiled and installed under Debian Lenny for i386. At first there were problems in the compilation of the toolchain, most probably because there haven't been new releases of this toolchain since 2006. Regardless, minimal changes in the source code were enough to solve these problems. Once installed, it was successfully tested, compiling a simple C program and generating an ARMv5 binary that could be executed on an ARM machine. The problem is that just compiling code is not enough for an ACS build for ARM machines, and running the executables in some other architecture than ARM is not possible just by having versions of libc for ARM installed on the system. This is not the right way of building complex systems like ACS, where pieces of the system just built are used to build further parts. A better way would be using a virtual machine that runs an ARM processor within it, but without the hardware limitations that would make it slow. The ultimate solution is to reorganize the build process for cross-building, but that would be a huge undertaking.

2.3.2 Emdebian alternative

An alternative solution tested was the Emdebian[12] (Embedded Debian) cross compilation toolchain, which, unlike ARM-Linux tools, is very easy to install. The process consists on adding a Debian repository, and then install the tools using the `apt` installer. Once the packages are installed and configured, the cross compilation of C or C++ code for ARM is possible.

2.3.3 Scratchbox

Scratchbox[13] is a cross-compilation toolkit designed to make embedded Linux applications development easier. It provides tools to configure a cross-compilation environment where the user can define a specific target architecture (in our case it was ARM). It includes a complete set of compilers (e.g. GCC 3.3) and runtime libraries (e.g. glibc 2.5). Since we have an ARM XScale PXA270, Scratchbox was configured for that purpose. Before any compilation it was necessary to install some tools on which ALMA software depends, such as `ksh`. Also, minor problems were fixed, very similar to those which arose during the porting to the Windows platform (see section 2.2), such as symbolic links for the shells, the `echo` command, among others.

After successfully setting up the cross compilation environment, we proceeded to compile the software on it, starting with the External Products. In this regard, the Tcl/Tk compilations presented problems because they depend on Xlib headers. As for the compilation of ACS' Common Software, the major problems that could not be solved were the fact of not having a Java Runtime Environment (JRE) for the target architecture, since Sun's version is only available for newer versions of the ARM architecture like ARMv6. A JRE is needed by several compiling steps, including code-generating and IDL compilation. To solve this problem, we tried to cross-compile the OpenJDK,[14] an open source version of Sun's Java Development Kit (JDK), but this presented problems due to the age of the compilers and libraries provided by Scratchbox (GCC 3.3 and glibc2.5).

In summary, in order to compile ACS on this architecture we would: (i) had to skip any Java-related compilation, or (ii) run the Java-related steps in a separate x86 machine, and copy the generated files by hand

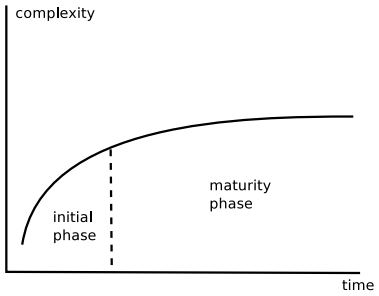

Figure 4. ACS maturity process: Control of complexity

to the target environment. This is a complicated and error-prone process, and it would become more complex while compiling more and more modules, since new intermediate Java-dependent steps might arise unexpectedly.

2.3.4 Reducing the size

Many ARM based machines have very limited disk space. For this purpose we used the `strip` tool, which removes information from executable binary programs and object files, thus potentially resulting in significantly less disk space usage. To illustrate, consider all shared libraries (`*.so`) from ACS that are in the directory `$ACSSW/lib`, which originally used 447 MB on disk. After using the `strip` tool on all these shared libraries, the size was reduced to 81 MB, making the elimination of about 81% of unnecessary information in the first instance, which indicates that it is an important point to note in situations which we find ourselves with limited resources against large pieces of software.

3. CONCLUSIONS

With the passing of time, ACS has gone through several levels at the time of the various porting processes explained above, making it clear that as new features are added to ACS the more complex the task of porting the framework becomes, both compiling the software on different platforms and architectures, and the replication of the standard environment. These tasks complicate due to changes in standards of compilers, libraries and the hardware itself. A very important point to note for large and long-term living projects, such as ALMA, is the need of constantly adapting to new technologies, which often become obsolete after only a few years. For example, the current `GCC` version is 4.5, `GCC` version 2.95 is no longer used or supported nowadays, while versions 3 are being phased out.

While the complexity of the maintenance has been rising when updating both platforms and architectures, it has been gradually decelerating. This is due to several factors, the most important one being the experience of the developers about ACS (figure 4), obtained thanks to years of working on these problems, methodologies and techniques for achieving better results, and changes in technology and standards. On the other hand it is important to note that due to the nature of the software itself it was possible to create a collaboration of different groups; in our case the ALMA-UTFSM Group has contributed in the porting tasks, among others. This implies that knowledge about it can be distributed between different institutions collaborating with the ALMA project, achieving lower complexity curves involved these tasks.

In the technical corner, we can say that one of the main points to note is the fact that the code is not the most important part of a framework, but just one component. This came into evidence several times, when difficulties arose not because of incompatible code, but because of an invalid runtime/compiling environment. The environment is then crucial when it comes to take a piece of software and port it to non-supported platforms. Sometimes it is easy to replicate the standard environment (e.g., when porting from one Linux distribution to another), but in other cases the support must be explicitly added into the framework (as in the case of the Windows port). Other solutions might arise with time, but these two have been the ones that we have used until now.

Regarding the ARM port, this was hampered by poor documentation and lack of support for basic tools used by ACS as a base, specifically referring to Java, in addition to the complex and rigid compilation system that

ACS uses. Nevertheless, some of these problems are specific to the ARM version used (ARMv5). As a future work, it might be interesting to continue this work for newer versions of the ARM processors, for which there is a Java Virtual Machine available. The build system is also an addressable problem, and studies of several improvements with respect to the rigidity of the compilation system can be done, allowing more flexibility to the developers, while trying to making minor changes to ACS.

In the case of the Windows port, work is being done by the ALMA-UTFSM group on finishing the Python and C++ portions of the framework. On the other hand, the ACS development team has taken the previous Java porting work, and is applying these to the HEAD of ACS with the intention of including them in the release of ACS 9.0. The Python and C++ porting efforts are also intended to be included in the mainstream. This will open a new wide range of projects that might want to use ACS, provided that it runs over the Windows platform.

Acknowledgments

This work is part of the ALMA-UTFSM Group, which in turn is part of the CSRG Group at the UTFSM. This work has been supported by ALMA-CONICYT grant #31060008. Horst H. von Brand's work was also supported in part by Centro Científico-Tecnológico de Valparaíso (CCTVal) grant FB0821. Thanks to Heiko Sommer, and specially to Gianluca Chiozzi, for their interest in this work.

REFERENCES

[1] Chiozzi, G., Gustafsson, B., Jeram, B., Sivera, P., Plesko, M., Sekoranja, M., Tracik, G., Dovc, J., Kadunc, M., Milcinski, G., Verstovsek, I., and Zagar, K., "Common Software for the ALMA project," in [*Proceedings of ICALEPS*], (2001).

[2] Chiozzi, G., Sekoranja, M., Caproni, A., Jeram, B., Sommer, H., Schwarz, J., Cirami, R., Yatagai, H., Avarias, J. A., Hoffstadt, A. A., Lopez, J. S., Grimstrup, A., and Troncoso, N., "ALMA Common Software (ACS), status and development," in [*Proceedings of ICALEPS*], (Oct. 2009).

[3] Tarenghi, M., "The Atacama Large Millimeter/submillimeter Array: Overview & status," *Astrophysics and Space Science* **313**, 1–7 (Jan. 2008).

[4] Schmidt, D., "The ACE ORB webpage." `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[5] Spiegel, A., "The JacORB web page." `http://www.jacorb.org`.

[6] Grisby, D., "The omniORB web page." `http://omniorb.sourceforge.net`.

[7] Brand, H. H. v., "Software development for ALMA in Chile: The ACS-UTFSM Group," in [*VI Reunión Anual de la Sociedad Chilena de Astronomía (SOCHIAS)*], 53–+ (2007).

[8] Avarias, J., "Repackaging ACS for embedded systems," Tech. Rep. 12/2007, Departamento de Informática, Universidad Técnica Federico Santa María (2007).

[9] Tobar, R. J., Menay, C. E., Avarias, J. A., and Brand, H. H. v., "Porting a large distributed control framework to the Windows Platform," in [*XIII Workshop on Parallel and Distributed Systems*], (Nov. 2009).

[10] Cygnus Solutions, "Cygwin webpage." `http://www.cygwin.com`.

[11] GNU ARM, "GNU-ARM project." `http://www.gnuarm.com`.

[12] Emdebian, "Embeded Debian project." `http://www.emdebian.org`.

[13] Scratchbox, "Scratchbox, a cross-compilation toolkit." `http://www.scratchbox.org`.

[14] Sun Microsystems, "OpenJDK." `http://openjdk.java.net/`.