A code generation framework for the ALMA common software

Nicolás Troncoso[a,b], Horst H. von Brand[b], Jorge Ibsen[c], Matias Mora[a,b], Victor Gonzalez[a], Gianluca Chiozzi[c], Bogdan Jeram[c], Heiko Sommer[c], Gabriel Zamora[b], Alexis Tejeda[d]

[a]Associated Universities, Inc. (AUI), Santiago, Chile;
[b]Universidad Técnica Federico Santa María (UTFSM), Valparaíso, Chile;
[c]European Southern Observatory (ESO), Garching bei München, Germany;
[d]Universidad Católica del Norte, Antofagasta, Chile

ABSTRACT

Code generation helps in smoothing the learning curve of a complex application framework and in reducing the number of Lines Of Code (LOC) that a developer needs to craft. The ALMA Common Software (ACS) has adopted code generation in specific areas, but we are now exploiting the more comprehensive approach of Model Driven code generation to transform directly an UML Model into a full implementation in the ACS framework.

This approach makes it easier for newcomers to grasp the principles of the framework. Moreover, a lower handcrafted LOC reduces the error rate. Additional benefits achieved by model driven code generation are: software reuse, implicit application of design patterns and automatic tests generation. A model driven approach to design makes it also possible using the same model with different frameworks, by generating for different targets.

The generation framework presented in this paper uses openArchitectureWare[1] as the model to text translator. OpenArchitectureWare provides a powerful functional language that makes this easier to implement the correct mapping of data types, the main difficulty encountered in the translation process. The output is an ACS application readily usable by the developer, including the necessary deployment configuration, thus minimizing any configuration burden during testing. The specific application code is implemented by extending generated classes. Therefore, generated and manually crafted code are kept apart, simplifying the code generation process and aiding the developers by keeping a clean logical separation between the two.

Our first results show that code generation improves dramatically the code productivity.

Keywords: ACS, Code Generation, ALMA, Distributed Systems, UML

1. INTRODUCTION

Chile has huge potential for astronomical observation, because it is the home of the majority of the most important observatories in the world, and the favorite place for further projects planned to be constructed during the next decade. With the clearest skies in the world, Chile is becoming an astronomical power. Some of the already installed international observatories are La Silla (NTT, ESO-3.6), Paranal (VLT), Cerro Pachón (Gemini, SOAR), Cerro Tololo (Blanco), and Las Campanas (Magellan). They are mainly financed by European and North American public organizations.

In addition, the currently largest astronomical project is under construction on the Chajnantor plateau in the Atacama desert: The Atacama Large Millimeter/submillimeter Array (ALMA).[2] This will be a radioastronomy observatory, formed by at least 54 12-meter diameter antennas and 12 7-meter diameter antennas, which will act together as one big interferometer. ALMA is a partnership between Europe (ESO), North America (NRAO), and Japan (NAOJ). Early science is expected to start the second half of 2011.

Further author information: (Send correspondence to Nicolás Troncoso)
Nicolás Troncoso: E-mail: ntroncos@alma.cl, Telephone: +56 2 4676165

As part of the cooperation with the Chilean government, ALMA is funding scientific development at Chilean Universities with around half a million dollars annually. In this context, the Computer Systems Research Group (CSRG) at Universidad Técnica Federico Santa María is collaborating since 2004 with the development of ALMA software. This work was financed by the ALMA-CONICYT fund, and is done in close coordination with the ALMA ACS and Control subsystems development.

The ALMA software is built on top of the ALMA Common Software (ACS)[3,4] infrastructure framework. The ACS framework consists of a collection of common patterns, components, and services. The heart of ACS is an component/container model based on Distributed Objects implemented as CORBA[5] objects. ACS provides common CORBA-based services such as logging, error and alarm management, configuration database and lifecycle management. It is based on the experience accumulated with similar projects in the astronomical and particle accelerator contexts, reusing concepts and technology used previously, like the VLT Common Software.[6] Although designed for ALMA, ACS can and is being used in other control systems and distributed software projects, since it has been implemented to be a generic control framework using generic design patterns and components off the shelf.[7] Through the use of standard constructs and components, non-ACS developers can easily understand the architecture of software modules. This makes maintenance affordable even on a very large software project like ALMA.

In Section 2, one of the current problems when developing software projects of the magnitude of ALMA. Section 3 is focused on exploring the current code generation techniques and its impact in software development. Section 4 proposes a solution for the problems presented in section 2. Section 5 reviews how the solution was verified. Finally, section 6 concludes what lessons have been learned so far and what has been achieved with this initial implementation.

2. PROBLEM APPROACH

Modern software control systems involve multiple subsystems and devices. In the domain of implementation for large scientific experiments, initial requirements are not well understood and the system design will be subject to an evolution and significant changes. As[8,9] adequately state, coding a large codebase requires an army of programmers. A better approach is to use model-driven programing, where the code is generated automatically starting from a model. This directly impacts the errors per line of code (since it takes over many routine, error-prone tasks) and helps enforcing a single coding style for a large portion of the code base. Benefits of such an approach are that it makes it easier to accommodate model changes, and it improves productivity since developers can focus on business logic instead of details of implementation. Besides the above benefits, improvements in the generator translate directly into improvements in the whole generated code base.

ACS (Alma Common Software) could do with more automatically generated code. It should be possible to write software components that use the framework in an easier way than its done today. Having this in mind, the author proposes a solution that accomplishes:

1. Code generation starting from a UML[10] model (and/or text representation). UML (Unified Modeling Language) is a software modeling specification maintained by the Object Management Group (OMG)[11]
   (a) Create IDL[12] files. This implies creating a full implementation of the interfaces in the model as IDL files so it may be compiled by IDL compilers.
   (b) Create a base class implementation starting from a class diagram. Base classes will be functional as they implement the relevant interfaces of the ACS framework.
   (c) Define the type mapping between the UML model representation and the specific language implementation.
2. Integrate design patterns with the code during the generation process.
3. Generate code for one of the ACS supported languages; as a start, Java.
4. Finally, validate the proposed solution through external opinions, current experience, and generating example implementations of a variety of models through a prototype implementation.

3. CURRENT TECHNOLOGY

Program transformation has applications and uses in many areas of software engineering, including compilation, optimization,[13] refactoring, program synthesis, software renovation, and reverse engineering.[14,15] The aim of program transformation is to increase programmer productivity by automating programing tasks, thus enabling programing at a higher-level of abstraction, and increasing maintainability and re-usability.[16]

Program transformation may be divided roughly into the areas of translation and rephrasing. A *translation* involves transforming a program from a source language into a "different" target language. Examples of such transformations are compilers such as GCC,[17] TAO IDL,[18] OmniOrb IDL,[19] jacOrb IDL.[20] Others are model to code transformations, like the ACS Error definition system.[21]

Rephrasings are transformations that take an input program and produce an output in the same language. Rephrasing tries to say the same thing using different words. The main scenarios where this is used are optimization, refactoring, and renovation.

The ALMA Common Software as a framework supports code generation to some extent. CORBA stubs and error definitions for all languages supported by ACS are automatically generated. Also, ACS provides a state machine code generator which to some extent makes ACS a generated code framework. Having all the previous generation facilities, the core programing is done in the software components. Nevertheless, writing software components is still a repetitive task, the overhead is the same for all components.[22] To circumvent the overhead of repetitive tasks some ACS code generators have been proposed, like the ACS generator described in 3.2.

3.1 ACS Error System Code Generator

The ACS Error System[21] is based on XML (Extensible Markup Language) and XSLT (Extensible Stylesheet Language Transformations)[23] code generation. Error types and error identifiers (error codes) are defined in XML files in an ACS language independent way. The code generation will the xml files to produce the corresponding Java, C++ and Python implementation. For each error definition type, a unique XML file must be written by the developer. This has to be done since corba has no way of translating exceptions from one language to an other. The generated ACS exceptions use the error codes to reconstruct the exception when it has been delivered from trough a CORBA interface. This code generator is considered to be complete and at this time it is being used in production in ALMA software.

3.2 ACS Generator

The ACS generator is a community code contribution to the ACS project. It is based on bdsGenerator[24] which was created to use ACS software with the Hexapod Telescope.[25] The bdsGenerator system is a C++ hardcoded template based code generator that parses IDL files and creates a C++ ACS component module. The ACS generator further extends the bdsGenerator work providing more advanced code generation. At this time the ACS generator only creates C++ component modules making it complementary to the work in this paper. The work proposed in this paper aims at solving some limitations of the ACS generator, such as one interface per module, and simple template extension. The lessons from the ACS generator should be used when extending the generator this paper proposes to the C++ implementation in an effort to merge the functionality of both.

3.3 Impact of Code Generation

During the past decade there has been an upsurge in the use of code generators in an attempt to deliver enterprise software to the market quickly.[26] Even though automated code generation is not a new concept, it had received relatively little recognition until this time from application developers.

Expertise is an intangible but unquestionably valuable commodity. People acquire it slowly, it takes a long time for a novice to become an expert. Design patterns[27] are a way to capture expert knowledge and convey it to novice developers. The drawback with this approach is that a design pattern is only know-how and not an implementation per se. Budinsky et al. describe an automatic code generator[28] which addresses this issue by generating code using a specified design pattern. This approach ensures consistency throughout the development.

Support of software evolution and maintenance is at hand when generating code from a formal definition language. Such a definition may be appropriately translated to cope with underlying platform changes without

major developer interaction.[29] A framework change is possible without cumbersome refactoring of all the software built on top of the framework, since the change can be made in the generator.

Some of the mayor concerns at hand when working with code generation is keeping the generated code clean and humanly readable. This is important for this project since the way defined by ALMA software development to handle problems in generated code is to debug the generated code, manually tweaking it until the problem is found and fixed, and the fix is then introduced into the generation process. This debug process can be extremely complex if the generated code is not well structured. Another important pitfall of the code generation is that if a bug is introduced in the generated code this bug will be present all over the generated software. At the same time, since the generation process is common, fixing a bug fixes it in all the generated software. On the other hand, if the generator itself is opaque, there will be a tendency to fix the generated code instead of the generator. These issues should be kept in mind when working with code generation.

4. PROPOSED SOLUTION

Following the industry standard and a requirement from the ACS software group, UML (Unified Modelling Language)[10] was chosen as the model definition language. As OAW[1] has been already selected in ALMA for data model code generation, with very good results, and has been therefore integrated in the ACS distribution it was natural to use this tool to build the ACS code generator.

The greatest challenge to achieve a proper code generation for the ACS framework was to properly translate an UML model into generated code. The overall architecture proposes to start from an UML model in XMI 2.0 UML[11] format, and use Open Architecture Ware to transform this model into an actual implementation. The generated implementation consists in the Java component code, the corresponding IDL interface, an XSD Schema file and an example configuration database usable for immediate testing. The generated Java component is a ready component in the sense that it may be instantiated by the ACS container without need of extension by the developer. This provides and easy and first verification that the generated code is working properly.

The general architecture of this code generator allows the developer to generate ACS components and characteristic components. The former are the simplest case and the later add additional complexity when generating code since they require access to the BACI interface. BACI, the Basic Control Interface is a general purpose control framework. It does not specify any specific control system, but it restrict the space of definable objects within a specific design and guidelines defined in the BACI interface.[30] It is not required to have components and characteristic components in different models. The ACS framework is container/component based. Hence ACS components are the basic functional software element that can be created. In case the control interface is needed it is possible to define characteristic components instead of simple components: this will give the software implementation access to BACI. The ACS code generator expects that classes identify their behavior, either as components or as characteristic components, using stereotypes and thus instructing the code generation on how to proceed.

4.1 UML Model Specification

When an input model is created for the ACS code generator, the UML model make use of new stereotypes not specified in the UML standard but specific for ACS. These stereotypes are enumerated in table 1. Additionally, it should be possible to specify data types which are not defined by the UML specification, for example the `Sequence` data type. This type is declared and used extensively in the ACS framework. If this non-standard data type is used in the model it should be used automatically in the generated code without any special intervention. Additionally the UML package name will be used by the code generator to name the `jar` file, and the directory structure. An important feature in the generation process is to define some class as `<<NOGenerated>>`, this instructs the code generator not to generate any implementation for it. This is significant when doing complex extensions, were it is possible that the developer does not want to generate code for a particular class, but wants to keep the model complete so the generator may define extending classes appropriately. At the time when this paper was written this feature was under discussion and most probably it will evolve into generating only the classes that have ACS specific stereotypes and not generating the others. Keeping with the spirit of having a complete model of the application.

Stereotype	Description
Class Stereotype	
`<<CharacteristicComponent>>`	Indicates if the component is characteristic.
`<<NOGenerated>>`	Indicates that this class must not be generated.
`<<IDLStruct>>`	Indicates that this class must be treated as an IDL struct.
Type Member Stereotype	
`<<ReadOnlyProperty>>`	The type uses the read-only BACI interface for properties.
`<<ReadWriteProperty>>`	The type uses the read/write BACI interface for properties.
Function Member Stereotype	
`<<Asynchronous>>`	The method is implemented as an asynchronous method.

Table 1. Stereotypes

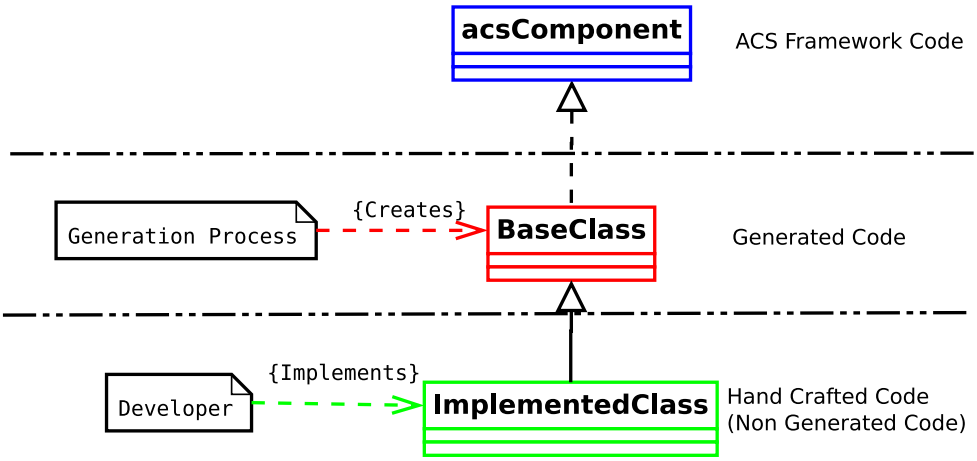

Figure 1. Code Separation

4.2 Non Generated Code

It is not possible to fully generate the logic within the software component. To achieve the full functionality of the component the developer must extend the generated code. This scheme has been shown to work in the ALMA CONTROL software code generation[8,9] where the generated code is extended by class inheritance. The same approach is used in the ACS Code Generator: a minimal working component is automatically generated, but the internal logic must be added by the developer by means of class extension.

The main benefit of this approach is that it isolates the generated code from the user written code. The side effect of this is that code may be regenerated multiple times without affecting user implemented code. This is known as a code separation pattern as seen in figure 1.

During the development cycle the base (generated) classes are created, then the developer extends these classes and implements complex functionality within the extensions. This approach is used with the generated IDL and generated Java files. It is also possible to use the same approach with the generated XSD files.

OAW provides a way to tell the generator which parts it must generate, by allowing the developer to delimit certain regions of code as protected. The code separation scheme described above was preferred over region delimitation since the later adds complexity to the generator and to the generated files themselves.

4.3 Generation Workflow

This steps taken in the generation workflow in order to convert an UML model into an ACS system, see figure 2 for a general layout of the steps to accomplish the end to end generation task.

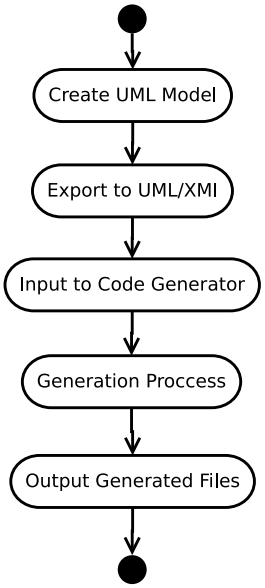

Figure 2. General Generation Workflow

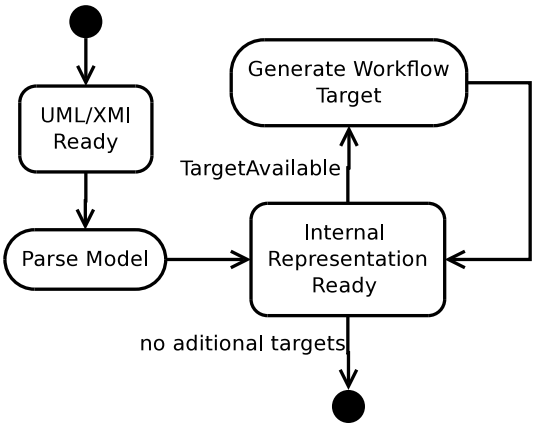

Figure 3. Generation Process Workflow

Task Name	Description
IDL tasks	
`genidlCommon`	Generate common IDL files. Contains structs and enumerated types.
`genidl`	Generate IDL files for every ACS component.
Java Tasks	
`genjava`	Generate Java Helper and Java Base classes.
ACS module tasks	
`genschema`	Generate Schemas in case of ACS characteristic component.
`gencdb`	Generate test CDB.
`genmake`	Generate Makefile.

Table 2. OAW Tasks

Figure 3 shows the internal steps taken by the code generation, it is a state machine that will transform the input model into an output for every target that has been specified. The targets described in table 2 are tasks taken by the ACS code generator in order to generate the ACS software module.

The tasks described in table 2 show the different OAW targets which are processed in order to generate an ACS software module. Every task is independent of the others, so they may be interchanged, eliminated or new steps could be added.

5. VERIFICATION

For testing purposes code from two models was generated. These models cover the full spectrum of code that can be generated by this work. Both models cover a basic component, and more complicated structures as enumerated in table 1.

At the time when implementation was begun, the setup chosen was:

- **Operating System**: A virtual machine with Red Hat Enterprise Linux 4.4 (interchangeable with Scientific Linux 4.4).
- **ACS**: ACS version 8.0.0.

- **OAW**: Version 4.1.3 with the Itemis Distribution*.
- **Modeling Tool**: Magic Draw 16.5

After executing the ACS code generator as described by the workflow in section 4.3, the output files were tested initially by compiling and installing the new software module. Compilation is achieved using the generated `Makefile`. A strict requisite is that the generated code must compile correctly and it should be possible to instantiate all classes. This is manually checked during the verification process, a test suite should be generated to automate this process.

Manual testing using the tools provided by ACS demonstrated that the generated module is fully functional. It was possible to instantiate and interact with the newly generated software module without any modifications to the generated code.

5.1 External and Peer Review

This work was followed closely by Gianluca Chiozzi, Head of the Control & Instrumentation Software Department at ESO, who had the opportunity to test preliminary versions of the ACS code generator.

This work was also presented in the Advanced Track of the 6th ACS Workshop, which took place at UTFSM, on November $13^{th} - 19^{th}$ 2009. Positive feedback was received during the presentations, including requests for new features which should be addressed as future work.

The model used for testing the ACS Code Generator was originally designed for the basic tutorial track of the 6th ACS Workshop, but it evolved into a validation suite for this project. A portion of the generated code, specifically the IDL files, were used during the workshop and the participants implemented the user code complementing the generated code.

6. CONCLUSIONS

This paper presents the result of the design and the test of a working implementation of a Model Driven generator for the ACS framework. The development process was based on an iterative process with hands on experience and real examples that made shortcomings and modifications needed in the design evident. The shortcomings detected were addressed in the evolution to the actual implementation, achieving a maturity such that the generated IDL files were used for the 6th ACS Workshop Basic Track Live Example (see section 5.1).

Some of the achievements of the ACS Code Generator are:

- The generated module is ready for compilation and usage as generated. No further development is required for initial testing.
- A pluggable module design was achieved, in the sense that supporting an additional implementation language or target does not disrupt, nor cause great impact, in the generation framework.
- The version of OAW used during development is the same as the one shipped by ACS-8.1. No porting efforts are required for integration.

This ACS Code Generator provides and excellent starting point (training) for a new comer to the ACS Framework. Its possible to have a working software module in under 10 seconds (time the generation process takes). The novel ACS user has only to fill in the stubbed functionality. The alternative is to adapt an ACS Example module, but this has always been error prone and new developers will take longer to achieve a simple working implementation.

*This is the distribution that integrates the necessary OAW 4.1.3 plugins. This is no longer necessary since now OAW is part of the eclipse project.

REFERENCES

[1] openArchitectureWare, "openArchitectureWare web page." www.openarchitectureware.org.

[2] Project, A., "The ALMA webpage." http://www.almaobservatory.org.

[3] 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 [*Accelerator and Large Experimental Physics Control Systems*], Shoaee, H., ed., 439–+ (2001).

[4] Chiozzi, G. and Šekoranja, M., *ALMA Common Software Overview* (2006).

[5] Object Management Group, "CORBA 3.0 – formal/02-06-01," (2002).

[6] P.Sivera, "VLT common software — Overview." http://www.eso.org/. VLT-MAN-ESO-17200-0888.

[7] Chiozzi, G., Bridger, A., Gillies, K., Goodrich, B., Johnson, J., McCann, K., Schumacher, G., and Wampler, S., "Enabling technologies and constraints for software sharing in large astronomy projects," in [*Proceedings of SPIE*], *Advanced Software and Control for Astronomy* **7019** (June 2008).

[8] Farris, A. and Juerges, T., *Device Driver Code Generation Framework*. ALMA (May 2007).

[9] Farris, A., Marson, R., and Kern, J., "Generating software modules using model driven software development," in [*Astronomical Data Analysis Software and Systems XVI*], **376**, 523 (Oct. 2006).

[10] Object Management Group, "Unified Modeling Language." http://www.uml.org.

[11] Object Management Group, "Object Management Group." http://www.omg.org.

[12] Object Management Group, "CORBA 3.0 – OMG IDL syntax and semantics chapter)." http://www.omg.org/cgi-bin/doc?formal/02-06-39.

[13] Grune, D., Bal, H. E., Jacobs, C. J. H., and Langendoen, K., [*Modern Compiler Design*], John Wiley (2002).

[14] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K., "Reverse engineering: A roadmap," in [*Proceedings of the Conference on the Future of Software Engineering*], 47–60, ACM, New York, NY, USA (2000).

[15] van den Brand, M., Klint, P., and Verhoef, C., "Reverse engineering and system renovation." http://www.cs.vu.nl/ x/reeng/REanno.html (July 1996).

[16] Visser, E., "A survey of rewriting strategies in program transformation systems," in [*Workshop on Reduction Strategies in Rewriting and Programming*], *Electronic Notes in Theoretical Computer Science* **57**, Elsevier Science Publishers (2001).

[17] Project, G., "GCC, the GNU Compiler Collection web page." http://gcc.gnu.org.

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

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

[20] Spiegel, A., "The jac ORB web page." http://www.jacorb.org.

[21] Jeram, B., Chiozzi, G., Plesko, M., Fugate, D., and Roberts, S., *ACS Error System*. ESO (Oct. 2002).

[22] Milcinski, G., Sekoranja, M., Lopez, B., Fugate, D., and Caproni, A., *ACS C++ Component/Container Framework Tutorial*. ALMA (may 2005).

[23] W3C, "XSL transformations (XSLT)." http://www.w3.org/TR/xslt.

[24] Community, A., "ACS contributed code." http://almasw.hq.eso.org/almasw/bin/view/ACS.

[25] Bochum, R.-U., "Hexapod - telescope." http://www.astro.ruhr-uni-bochum.de/astro/oca/hpt.html.

[26] Stephens, M., "Automated code generation." http://www.softwarereality.com.

[27] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., [*Design Patterns: Elements of Reusable Object-Oriented Software*], Addison Wesley Professional (1994).

[28] Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S., "Automatic code generation from design patterns," *IBM Systems Journal* **35**(2), 151–171 (1996).

[29] Floch, J., "Supporting evolution and maintenance by using a flexible automatic code generator," in [*Proceedings of the 17th International Conference on Software Engineering*], 211–219, ACM, New York, NY, USA (1995).

[30] Plesko, M., Tkacik, G., and Chiozzi, G., *ACS Basic Control Interface Specification*. KGB Team, Jozef Stefan Institue (Sept. 2005).