Automatic Code Generation for Attitude and Orbit Control Systems Using Domain-Specific Languages

Pedro Azevedo Isidro
pedro.isidro@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
November 2014

Abstract

The Attitude and Orbit Control System (AOCS) is the spacecraft subsystem responsible for determining and controlling the vehicle’s orbit and orientation. Similarly to other kinds of embedded systems, its software has been continuously growing in size and complexity. However, very few satellites are produced each year, when compared to other industries. The consequence is an insufficient level of automation in the development process, which leads to low software reusability, driving up the costs. The proposed solution to this problem is to develop a Domain-Specific Language (DSL) using the Xtext language workbench. The language contains tailored abstractions that allow a simple system model to be created, and is bundled with a specific editor, a model validator and a code generator. The generated C++ code is then customized to implement low-level behavior. A proof of concept centered in the telecommand handling functionality is developed to prove the feasibility of applying the solution to the whole subsystem. Its design and implementation is based on an analysis conducted on the source code of the TET-1 satellite of the German Aerospace Center (DLR). The resulting DSL-based framework is tested with an example model and target code customization, showing its ease of use and proving that it behaves as expected.

Keywords: Attitude and Orbit Control System, Model-Driven Software Development, Domain-Specific Language, Automatic code generation, Xtext, Eclipse.

1. Introduction

The size and complexity of a satellite’s On-Board Software (OBSW) has been continuously increasing, with satellites from the European Space Agency (ESA) now containing hundreds of thousands of lines of code [1]. Even though very mature tools with code generation capabilities exist to model control algorithms, these algorithms represent only about 20-30% of the AOCS software [2].

Developing complex software systems at a low level of abstraction presents several problems. First of all, having human developers performing repetitive low-level tasks is very unproductive. Plus, a human programmer is prone to make occasional errors, thus decreasing the reliability of the software. It also affects software reusability, recognized by the major stakeholders in the space industry as one of the main factors driving development costs [3], since the reusable elements are mere objects or routines, instead of more abstract and portable concepts.

In order to create a high-level model of the AOCS, it is necessary to explore the specificity of the problem, using a methodology called Domain-Specific Modeling (DSM). Source code can then be automatically generated from these high-level models.

This thesis proposes to study the feasibility of using Domain-Specific Languages (DSLs) to automate the development of AOCS software. The goal is to hide repetitive tasks from the developer, while providing appropriate abstractions to express the model of a system.

The first step is to find which parts of the AOCS software are general and which are application-specific, by studying the literature on the topic and by directly analyzing the source code of a specific implementation. The next goal is to transform this understanding into a domain model which adequately represents the AOCS.

After evaluating what can be automated or abstracted and how, a proof of concept is to be developed. The goal is to end up with a Domain-Specific Workbench (DSW), including a modeling language (a DSL) and a code generator.

2. Background

This section provides background knowledge on the relevant topics and concepts, being fundamental to a proper understanding of this work.
2.1. Attitude and Orbit Control System
The AOCS is the spacecraft subsystem responsible for determining and controlling the position and orientation of the vehicle during all phases of a mission. It is an embedded, mission-critical system, with hard real-time constraints\(^1\) [4].

The core task of the system is the enforcement of the control algorithms, typically done autonomously in closed-loop, but with the possibility of being overridden by ground commands [5]. Measurements are collected from the sensors and used to calculate the control signals to be sent to the actuators.

The system must also manage a two-way data exchange link with the ground station. It must receive and process telecommands via the uplink and provide general status information, or Housekeeping Data (HKD), to be sent to the ground via the downlink. Telecommands are used to influence the behavior of the satellite, while HKD is used for monitoring purposes.

Other than that, the AOCS is usually responsible for managing the operational mode, executing attitude and orbit change manoeuvres and detecting and recovering from failures. Figure 1 summarizes the functions of the system.

---

It has been shown that about 70% of faults are introduced early in the development process, and 80% of those caught only at the stage of integration testing or later, where the cost of fixing them is much higher [7]. By validating this model, which is directly linked to the target system, many of these faults can be eliminated early in the development process.

The approach of this thesis work is to use a domain-specific textual modeling language. A Domain-Specific Language (DSL) can be created by capturing the common features of an application domain, thus offering powerful concepts and notations tailored to the domain, while allowing the specification of a concrete system [8].

The code generator maps an abstract input model into executable target code. This process is nothing more than a specific case of so-called model transformations. Template languages are ideal for code generation, as they offer a good syntactic mix of model traversal code and to-be-generated code [6].

It is usually not feasible to create a model which fully and completely specifies a system, in which case it becomes necessary to integrate generated and manual code. They must however be kept separate to prevent the generator from overwriting manually written code. The most widely accepted way to introduce code separation is the Generation Gap Pattern (GGP), which uses the concept of inheritance\(^2\) for that end [9], as shown in Figure 2.

---

Figure 1: AOCS software (adapted from [2])

Due to the limitations in performance and memory of embedded space systems, the needed high-level abstraction mechanisms should be resolved at compile time [6]. Such can be achieved, for example, using domain-specific models which are automatically translated to low-level code.

2.2. Model-driven software development
Model-Driven Software Development (MDSD) is a software development methodology which focuses on creating and exploiting models, putting them at the core of the development process. A model is nothing more than an abstraction of some aspect of a system, i.e., a simplification of reality. Another important concept is that of a metamodel. A metamodel defines the language used to describe a model [6].

The higher level of abstraction not only results in quicker development, but it also provides an opportunity for an early verification of the system design.

---

\(^1\)The input-computation-output process must meet strict deadlines.

\(^2\)Used in object-oriented programming

\(^3\)A stub is a temporary replacement for code that still has not been developed.
2.3. Domain-specific languages
Unlike general purpose (programming) languages like C/C++ or Java, DSLs are limited to, but optimized for, use in applications within a given domain. DSLs allow system design to be conducted at the level of abstraction of the domain, which yields numerous advantages: shorter and more readable programs which can be used for communication between developers and domain experts, and more reliable software with earlier validation. Additionally, the created model can always be used to implement further automation. The result is a major improvement in productivity and reliability, with several electronics manufacturers reporting productivity increases of 300-1000% and a 50% decrease in number of program errors, when compared to manual coding [12].

The grammar consists of a set of rules which determine how a script written in the language is converted into an actual model of the system [9]: the semantic model, usually represented as a so-called Abstract Syntax Tree (AST). the semantic model is usually just a subset of the domain model, as not all of the domain concepts are best handled by the language [9].

A complete DSW also contains a reference architecture and an execution engine (code generator or interpreter) which can complete the model, thus allowing it to be simpler. The elements or features which are invariable belong to the reference architecture, those that can be derived by fixed rules from an input model belong to the execution engine, and only the remaining set of variable features should be included in the DSL, which must in turn provide the adequate abstractions to express them.

Figure 3 shows a typical DSL processing workflow.

2.4. Project context
The AOCS DSL was developed at the Department of Simulation and Software Technology (SC) Braunschweig, of the German Aerospace Center (DLR). SC has been responsible for the development of AOCS software for several DLR lead missions under the German space program. The latest of those satellites is TET-1, depicted in Figure 4. It is a technology demonstration satellite and it was launched on July 2012 [13], under the German national On-Orbit Verification (OOV) program [14].

The repetition of low-level code in the development of AOCS software for this and previous projects lead the developers to believe that there was room for automation. The possibility of using a DSL-based solution to solve the problem stems from another one of SC’s research projects, called LAMBDA (Language fo Metadata Based Applications). Its product is a new domain-specific framework to model metadata for knowledge management tools. The LAMBDA DSL was developed in the Xtext language workbench, which is why it is chosen for this thesis work.

3. Design
This section summarizes the design process of the AOCS DSW, including the definition of the scope for implementation and the semantic model and to-be-generated target files extracted from from the analysis of the TET-1 AOCS software.

3.1. Scope definition
The AOCS is a big and complex system, even for small satellites. Therefore, it is necessary to define a specific functionality of the system which is to be implemented as a proof of concept. Since this implementation aims to demonstrate the possibilities of applying DSL technology to the entire system, it is also fundamental to recreate the general structure around the implemented feature, up to the top-level elements of the system. The proof of concept can then be branched at any point to implement support for other functionalities, reusing (and adapting, if necessary) the higher-level constructs of the language.

By consulting with the AOCS specialists at SC, the telecommand handling functionality has been chosen for this purpose. The analysis of the system consequently follows a bottom-up paradigm, starting with the C++ classes corresponding to the command handlers. The objective is to end up generating a partial, but compilable, collection of target files. Stub classes and methods are to be used, when needed, to replace code that falls beyond the defined scope.

3.2. Semantic model
From the analysis conducted on the TET-1 source code, it is found that the desired semantic model is a simple hierarchic structure. The model is made up of applications, applications contain components, components contain commands, and commands in turn contain parameters. This semantic model is depicted in Figure 5. The simplification of the sys-
tem brought by the use of domain-specific abstractions is obvious.

Figure 5: AOCS semantic model design

Ultimately, the Parameter concept is the most complex in the language. In the analyzed source code, parameters are defined only with a name and a type. They also have a range or set of valid values they can assume, which are coded directly into the validity check performed in the beginning of the command execution. Therefore, an additional feature to specify the valid values of a parameter is added, allowing to automate the validity check. Although not currently used, another feature to specify the physical quantity that it represents and the units in which it is expressed can be used to document the code, and possibly to implement additional model validation and formal verification in the future.

3.3. Target files
The framework is to follow the GGP and the generate once policy to separate automatic and manual code. The idea is to put purely generated files in a folder named src-gen and to have files which are generated only once in a different folder named src, configured to allow the user to modify them.

A base handler class called CommandHandler works as a superclass for each specific handler. Although technically it is a default implementation, i.e., it can be instantiated, it acts mostly as an interface. Its handle method receives as input a pointer to the (location in memory of the) parameters for the execution of the command. A set of parameters is defined as a single command-specific structure (a C++ struct). The method returns the number code (ID) of an error message (of type ErrorMessage::ID), to provide feedback about the success of the command execution.

The default implementation of handle is overridden in the command-specific handler classes (CommandHandlerMyCmd), according to the telecommand that they need to execute. By comparing different handlers, the general workflow of the method can be identified. Some of the steps can easily be automated, like the validity check performed on the received parameters. The actual execution of the command however, is made up of low-level operations and calls to methods of so-called AOCS components, which must be manually implemented.

Figure 6 outlines the handler classes, while also showing the mechanism for code separation. The behavior method, whose implementation is left to the user, is called by the handle method.

Components are units of the system that implement certain functionalities. For example, the EPC (Estimation, Prediction and Control) is the component responsible for all tasks concerning the automatic control loops. Mission control can interact with the components and their functionalities by issuing telecommands. The AOCS DSL must provide the possibility to instantiate components to wrap a group of commands. For each component, there is a class called MyCptCommands which instantiates the command handlers which are grouped under it. This class has a method called registerCommands, which registers the commands at the telecommand interpreter of the application to which the component belongs.

Since components are to be declared at the DSL level and automatically generated, an additional class can be created which holds all component instances and registers them (see Figure 7), similarly to what is already done for the command handlers. This class, named Components, contains an
instance of each component as a private member. Not only does it instantiate the user-defined components, but also the default component `SurveillanceInterface`, and a command interpreter for each defined application. Its only method, `registerComponents`, registers each of these instances at the component manager.

A component has an arbitrary number of members and methods. Without studying AOCS components further, the only option is to let the user define and implement them manually. The consequence is that the generated component classes are practically just empty stubs. Like error messages and commands, components are also indexed by an ID defined in a header file (with extension `.h`). These, however, are unique in the whole system.

The `ComponentManager` class, keeps a table of pointers to components. Command handlers and other components can call its method `getLocation` to retrieve a specific pointer.

Applications are the software modules, and top-level concepts, of the AOCS. Each application is unambiguously identified within the whole spacecraft system by an Application Process Identifier (APID). An enumeration of telecommand IDs exists for each application, listing all the commands wrapped by the components of that application. Therefore, command IDs are unique for each application. However, mission control stations on the ground actually identify all the commands of the spacecraft system with a single number. The enumeration of the commands of an application is, for that reason, accompanied by a constant `OFFSET` integer. This allows the translation of a global identification code to an application-specific one.

The commands received from the ground station are processed and dispatched to the telecommand interpreter of the target application. These interpreters are components instantiated from a template class called `CommandInterpreter`.

**Figure 8:** Surveillance component class

A look into the file tree of the TET-1 source code reveals that both the event and error message IDs reside in the directory of the surveillance component, so that is where they are going to be generated. These ID definitions are created containing only their default members, so they must be manually customizable. Since they are defined in `enums`, and not classes, inheritance cannot be used to provide the desired extensibility. The only solution is therefore to generate the corresponding header files into the `src` folder, with comments instructing the user on how to edit them.

### 4. Implementation

This section describes the implementation of the proof of concept, based on the previously presented design.

#### 4.1. Language modules

A language module named Common was developed with the intent of extending the Terminals grammar, from which all Xtext languages inherit by default. It includes new terminals and basic data types which are not specific to the AOCS language.

A new terminal rule, inspired in Java’s documentation comments, was created to allow the user to document the model. These multi-line comments are parsed and can be included in the target code.

#### 4.2. Grammar

A grammar definition is used by Xtext to automatically generate the parser and infer both the concrete and abstract syntax of the language, *i.e.*, its textual representation and its meaning, respectively. The grammar is constructed based on:

- The presented reference semantic model, which serves as a guideline for designing the abstract syntax.
- The desired concrete syntax. To improve ease of use and acceptance of the tool, it must be as simple as possible and reflect the terminology used in the domain.
- The general good practice of having a loose grammar accompanied by strict validation [16]. This is advisable because the validator can handle issues in the model much more gracefully, providing clear error messages and hints on how to fix them.
The implemented grammar rules yield the following examples of valid parameter definitions:

```java
/**
 * This comment will be included in the
 * target code
 * for boolPar
 */
parameter boolPar is bool array(2)
parameter floatPar is float in range 0.0 to 1.0 with units ms
parameter intPar is int32 in enum ENUM
```

A parameter can be constrained to a range (or to a set of values (enumeration)). The user can reference a range or a set defined elsewhere, as long as it is within the scope of the parameter. Alternatively, a constraint can be declared directly in the definition of the parameter, using the anonymous versions of the rules. Parameter constraints can be defined as

```cpp
class ENUM is (ZERO, TWO=2, THREE)
class RANGE is 0 to 5
```

Parameter types and measurement units can be specified by using a keyword, selected from a set of alternatives.

The cross-referencing mechanism of Xtext was used to extend the designed semantic model. By allowing elements to be defined in an outer scope, the user gains the possibility to better separate definitions by level of abstraction. For example, a parameter can be defined at the component level, and then referenced in a command within that component. This design pattern can be applied as follows:

```java
command cmd is
    parameter par is bool
end command

component cpt is
    command cmd
end component

application app is
    component cpt
end application
```

This also implies that AOCS elements can be defined and not used. At the model scope, for example, applications defined are automatically a part of the model, while any other lower-level definition will only be included when referenced in an appropriate inner scope.

Component names have a special notation. They can be given by a qualified identifier instead of a simple identifier. This is meant to be used solely for organizing the generated files. Within the resulting software system, components are treated only by their last name, which must therefore be unique.

4.3. Model validator

One great advantage of DSM is that is allows validation to be performed at the model level. When using an Xtext-based DSL, the validator is automatically integrated with the editor. It continuously checks the code for validity, providing marks and custom messages of different kinds: errors, warnings and information. Xtext already provides a couple of validators. One of them is programmed to check for the uniqueness of object names within each scope of the model. When it is activated, Xtext adds the `NamesAreUniqueValidator` to the abstract validator class.

A few custom check methods were implemented in the `ParameterValidator` class to verify the consistency between the declared parameter type and the other attributes. Error messages are shown if a value constraint or a measurement unit is declared for a boolean parameter, or if the user tries to limit a floating point parameter to a set of discrete values. Figure 9 shows how one of these error messages looks like in the AOCS editor.

![Figure 9: Invalid parameter constraint](image)

Specific validator classes have been implemented for `Range` and `Enumeration` as well. The mentioned `NamesAreUniqueValidator` will prevent the instantiation of objects with the same name within a scope. However, due to the referencing pattern used, it will still allow, for example, for a command to be defined and referenced within the same component definition. Therefore, a validation method was implemented at each level to avoid that erroneous input.

The fact that telecommands must be unique for each application, which is not their immediate container, demands for a specific. Similarly, components must be checked for uniqueness at the model level.

The last kind of validity checks implemented, at the top level of the model, concerns unused objects. As stated before, any element defined out of place is only a part of the system once it is referenced in an appropriate inner block. With that in mind, validation methods were developed to warn the user of unused objects. As an example, Figure 10 shows how the warning looks like for a defined but unused parameter.

![Figure 10: Warning for an unused object](image)
4.4. Code generator

The AOCS code generator is configured to output files into two different folders, as to implement the GGP:

**src-gen** Purely generated files are written to this folder. All the resources in it are cleaned and re-written every time the code generator runs. Additionally, the files are marked as “derived”, so that the user is warned by Eclipse when trying to edit them.

**src** This folder receives all the files which are to be generated only once. They are not overwritten or deleted, and the user may edit them at will.

Many kinds of names or identifiers are needed by the classes involved in code generation. They are all represented within the generator as Java Strings. Some of these names are used for output configuration, like folder and file names, and file extensions. Others contain the name of files of the C++ standard library which need to be included. But most of those strings represent C++ concepts like class, method, or variable names, as well as data types, *enum* labels, namespace names and debug flags.

They are used as parameters for the code generation templates. In order to make it as easy as possible to configure the code generator, none of these strings is coded directly into the templates. Static names, which are independent of the input AOCS model, are defined with constant static strings. As an example, Listing 1 shows the definition of the name of the file containing the command interpreter template class, which happens to reuse other such strings.

```java
val public static
  FILE_commandInterpreterTemp =
  CLASS_commandInterpreter +
  EXTENSION_definition

Listing 1: Definition of a static name
```

Other names which must be dynamically created are given by static methods. Listing 2 shows one of these methods. This one in particular infers the name of the specific command handler class from the name of the command.

```java
def public static
  getHandlerClassName(Command command) {
    return CLASS_commandHandlerBase +
    command . name . toFirstUpperCamelCase
  }

Listing 2: Method to derive a dynamic name
```

File creation is commanded by so-called *generate* methods. For each C++ class to be generated, two files are typically compiled: a header (or definition) file and an implementation (exceptions are abstract classes and concrete classes with simple default implementations). Class-level *generate* methods abstract away this detail. They are called from a higher-level method and then order the generation of each individual file.

Methods prefixed with *compile* (*compile* methods) are put in charge of creating the contents of the target files. These methods are made to return a string or a character sequence (Java *CharSeq*), created using the template capabilities of the Xtend language.

A folder named *resources/library* exists in the main Eclipse plug-in project of the AOCS DSL. A method called *generateLibrary* copies each file found in that folder (and sub-folders) into the *src-gen* target folder. In this way, existent files can be easily included in the generated code. To prove its workings, a header file *debug.h* has been created, containing basic support for printing debug messages.

The entry point to the code generator is the method *AOCSGenerator.doGenerate*, which is defined for compliance with the *IGenerator* interface. From this point on, the generation process takes the following steps:

1. **Surveillance:** Error message and event IDs (two header files).
2. **Base classes:** Component and command handler base classes (two header files).
3. **AOCS interfaces:** Surveillance interface class, telecommand interpreter template and component manager class (three header and two implementation files).
4. **Applications:** Telecommand IDs for each application (one header file per application).
   ```
   (a) **Components:** Component IDs and instances (two header files and one implementation file).
   
   i. **Commands:** Class to hold command handler objects (one header and one implementation file per component) and command handlers (two header files and two implementation files per command).  
   
5. **Library:** Copy all files in the library folder.

Remember that there can be unused definitions of AOCS elements. It is to filter out these definitions that the generation of all the custom files concerning applications, components and commands is done hierarchically.
4.5. Editor support
Several other features can be implemented/customized with Xtext, mainly intended to improve the user experience.

Content assist features are intended to help the user better understand the elements of the language and what can or should be written in each part of the program. Furthermore, automatic code completion also increases coding speed and efficiency, by instantly inserting typo-free code.

Depending on the context of the model, the user can select from a list of suitable templates, for parameters, commands, etc. The general structure of the chosen element is then automatically inserted, allowing the user to navigate between the customization fields. When navigating the input fields, the user can again request completion proposals. These simple proposals work on a token-by-token basis and are automatically provided by Xtext. In the case of parameter types and measurement units, which are to be chosen from a given set of values, it was deemed important to have a drop-down list automatically appear when the user navigates to that field, as shown in Figure 11.

Figure 11: Parameter type drop-down list

The only customization regarding the syntax coloring in the AOCS editor was the differentiation of parameter type and measurement unit codes from other keywords, for the same reason as above. Xtext already provides default highlighting of grammar keywords, and the different coloring for documentation comments is implemented in the Common language module.

The last feature implemented is a custom text formatter. It is relatively simple to implement and allows the user to easily and quickly format the textual model.

5. Demonstration
The best way to show the capabilities of the developed tool is to perform a demonstration, guiding the reader through the steps of a simple example using the Eclipse Integrated Development Environment (IDE).

5.1. Project setup
Since no specific project wizard got to be developed for the AOCS DSL framework, an AOCS project must be set up manually. The first step is to create a new C++ project and a model file in that project. Eclipse will automatically recognize the file extension (.aocs) and integrate the DSL tools with the project. At this point, the Xtext-based DSL is already working. However, a few additional steps are required for a successful compilation of the target code. First the compiler must be instructed on where to look for included header files. Then an entry point to the program must be provided. In C++, the entry point is the so-called main function.

Now the project can be built, either by direct command or by re-saving the model file. The C++ toolchain will create an executable file in the project folder, as can be seen in Figure 12.

Figure 12: Project folder after compilation

5.2. Example AOCS model
A simple model was developed to show the capabilities of the DSL. It consists of a single application named app, with a single component named my.cpt. This component contains two commands: cmd and cmdTest. The first has several example parameters. The second is empty to allow testing without worrying about parameter constraints.

5.3. Target code customization
All generated classes, methods and enumerations needing (or allowing) customization contain a TODO task tag. These tags are recognized by Eclipse, and can be seen in the Tasks view. Figure 13 shows a screenshot of this view after the steps taken in the previous sections. The view shows which task must be performed in which file, and the exact line of code where the tag is inserted. These customization points can easily be accessed by simply double-clicking the corresponding task.

The first modification in this demonstration is to add a custom error message. Upon opening the file ErrorMessageIds.h, the definition of default IDs can be seen, accompanied by instructions on how to edit it (see Figure 14). An error message was added, being assigned the value ‘3’.

Next, a boolean method named exampleComponentMethod is defined and implemented in the Opt class files. The method does nothing more than returning the boolean value false.
The logical next step is to use the component *cpt* and the implemented custom method within the execution of a command handler. The file *CommandHandlerCmdTest.cc* contains a method stub for the implementation of the execution behavior of the command *cmdTest*. This example implementation starts by getting a pointer to the component, resorting to the component manager. It then calls the method *exampleComponentMethod* within a condition. Since this method is expected to return *false*, the command handler is expected to return the custom error message defined earlier.

5.4. Testing

The first step towards testing the resulting software is to add the command handler debug flag to the compilation command (“-DAOCS_TEST,COMMAND_HANDLER”). Simply hovering the handler class name in the *behavior* method implementation shows the appropriate flag. After building the project again, the resulting executable file will include instructions to print debug messages to the standard output (the Eclipse console, in this case).

The only thing left to do is to code a test routine in the previously created *main* function. It essentially orders the execution of *cmdTest* through the command interpreter, to simulate a telecommand received from a ground station. The project is subsequently re-built and run in Eclipse. Figure 15 shows the resulting message printed in the console. The message confirms the expected behavior.

6. Conclusions

This thesis work was intended to answer the problems of low software development automation and consequently low reusability, which translate into higher costs and longer time of development. It was proposed to do so by using DSM to create high-level models from which software code can be automatically generated.

The scope for implementation of a proof of concept was chosen to be centered in the telecommand handling functionality. Then, based on the system analysis, a preliminary design was developed, including a language module common to all satellite subsystems, a simple hierarchical semantic model and a plan for integrating generated and manual code.

The *generate once* pattern was chosen to provide the separation of manual and automatic code in the file system. Some of the customizable software elements laying in the boundary of the defined scope for implementation were chosen to be created as mere empty stubs, so that the developed solution does not present any drawback when compared to manual development.

The implementation of the proof of concept was based on the analysis performed and the resulting design of the framework. The Common language module, wrapping concerns not specific to the AOCS, includes the definition of a type of comment intended to be parsed and possibly included in the target code for documentation, thus increasing the understandability of the generated files.

The user experience was further improved by implementing editor features like content assist, syntax highlighting and code formatting, making the framework quite easy to use.

By comparing the extremely simple input model with the generated code, it can be concluded that this tool indeed removes a big workload from the software developer.

An example AOCS model was used for demonstration. The generated code was seen to include task tags in the customization points, giving the user easy access to these points. Next, simple modifications were then done to implement custom behavior. Finally, a test routine simulating a telecommand sent from a ground station was developed to prove the correct functioning of the resulting software system. The program was run, yielding successful results.
7. Future work
The creation of an expression language which reflects C/C++ low-level operations could possibly allow the development of the whole AOCS software to be carried out in a single editor. However, it would be a relatively big venture. For now, the presented solution satisfies the requirements.

The growth in size and complexity of the code generator was underestimated in the planning and design phase of the AOCS DSL. Although its files are still perfectly readable due to their organization, looking back it seems like an intermediate model-to-model transformation would have made the framework more modular. An interim model could be inferred which would be as close to the generated code as possible.

Another possible improvement to the AOCS DSL framework would be to create a specific project wizard to automate the process of setting up an AOCS project, described in the product demonstration in Section 5.1 of the product demonstration. This would significantly improve the ease of use of the tool.

Acknowledgements
I would like to thank Meenakshi Deshmukh for giving me this opportunity to work in and learn about the field of software development at the DLR, and for continuously being available to help me with anything I needed, along with all other colleagues at SC Braunschweig. I would also like to thank Prof. Dra. Alexandra Moutinho, for orienting me at IST and specially for the quick and detailed feedback during the writing of this thesis.

References