TÉCNICO LISBOA

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

**Pedro Azevedo Isidro**

Thesis to obtain the Master of Science Degree in

**Aerospace Engineering**

Supervisors: Prof. Alexandra Bento Moutinho

MSc. Meenakshi Deshmukh

**Examination Committee**

Chairperson: Prof. Fernando José Parracho Lau

Supervisor: Prof. Alexandra Bento Moutinho

Member of the Committee: Prof. João Carlos Prata dos Reis

**November 2014**

Dedicado aos meus avós, Zé e Graciete.

# Acknowledgments

I would like to start by thanking 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.

Additionally, I would like to express my appreciation for all the amazing people that have accompanied me throughout the last five years, in Lisbon, in Delft and in Braunschweig. My childhood friends from the Azores, Técnico's aerospace engineering class of 2009, the ones who made Marcushof an unforgettable place, and those with whom I shared some great times in the kitchen of Wendenring 35. It was a great and successful experience, thanks to you.

Last but not least, I want to thank my family, who showed me nothing but support and pride, pushing me and expecting from me no less than I was capable of.

## Resumo

O Sistema de Controlo de Atitude e Órbita (AOCS) é o subsistema responsável por determinar e controlar a órbita e orientação de uma nave espacial. Tal como outros tipos de sistemas embarcados, o seu software tem crescido continuamente em tamanho e complexidade. No entanto, em comparação com outras indústrias, muito poucos satélites são produzidos a cada ano. A consequência é um nível insuficiente de automação no processo de desenvolvimento, que leva a uma baixa capacidade de reutilização de software, elevando os custos. A solução proposta para este problema é desenvolver uma Linguagem de Domínio Específico (DSL) usando a plataforma de desenvolvimento de linguagens Xtext. A linguagem contém abstrações adequadas ao AOCS, que permitem a criação de um modelo relativamente simples de um sistema. É acompanhada de um editor específico, um validador de modelos e um gerador de código. O código C++ gerado é então personalizado para implementar funcionalidades de baixo nível. Uma prova de conceito centrada no processamento de telecomandos é desenvolvida para provar a viabilidade de aplicar a solução a todo o subsistema. A sua concepção e implementação baseia-se numa análise realizada ao código-fonte do satélite TET-1 do Centro Aeroespacial Alemão (DLR). A Plataforma de Domínio Específico (DSW) desenvolvida é testada recorrendo a um modelo-exemplo e uma simples personalização do código-alvo, mostrando a sua facilidade de uso e comprovando que se comporta como esperado.

**Palavras-chave:** Sistema de Controlo de Atitude e Órbita, Desenvolvimento de Software Guiado por Modelos, Linguagem de Domínio Específico, Xtext, Eclipse.

## 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 Domain-Specific Workbench (DSW) 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.

# Contents

# List of Figures

# List of Listings

# List of Acronyms

| | |
|---|---|
| **AOCS** | Attitude and Orbit Control System. |
| **API** | Application Programming Interface. |
| **APID** | Application Process Identifier. |
| **AST** | Abstract Syntax Tree. |
| **DLR** | German Aerospace Center. |
| **DSL** | Domain-Specific Language. |
| **DSM** | Domain-Specific Modeling. |
| **DSW** | Domain-Specific Workbench. |
| **EBNF** | Extended Backus-Naur Form. |
| **EMF** | Eclipse Modeling Framework. |
| **EPC** | Estimation, Prediction and Control. |
| **ESA** | European Space Agency. |
| **GGP** | Generation Gap Pattern. |
| **GPL** | General-Purpose Language. |
| **HKD** | Housekeeping Data. |
| **IDE** | Integrated Development Environment. |
| **M2M** | Model-to-Model. |
| **MDE** | Model-Driven Engineering. |
| **MDSD** | Model-Driven Software Development. |
| **OBSW** | On-Board Software. |

|  |  |
|---|---|
| **OO** | Object-Oriented. |
| **SC** | Simulation and Software Technology. |
| **TET** | Technology Experiment Carrier. |
| **UI** | User Interface. |
| **UML** | Unified Modeling Language. |
| **URI** | Uniform Resource Identifier. |

# Chapter 1

# Introduction

A spacecraft can be placed into orbit for a number of purposes, like observation, communication or research. Once in orbit, the spacecraft becomes an artificial satellite. A satellite is composed by a bus and a payload. The payload can include various instruments or experiments. The satellite bus, on the other hand, must perform many mission-critical tasks, which are divided into separate subsystems. The way that the subsystems are defined depends on how these tasks are distributed among them. However, combining information from [1] and [2], one can infer a general subsystem configuration for unmanned spacecraft:

**Structure subsystem:** Physical structure of the spacecraft, launcher adapter and other moving parts.

**Power subsystem:** Generates, stores, distributes and regulates electrical power supply to the whole spacecraft.

**Thermal control subsystem:** Maintains all equipment within allowed temperature ranges.

**Telemetry, tracking and command subsystem:** Used to track, monitor, and communicate with the spacecraft from the ground. Also called “communications subsystem”.

**Command and data handling subsystem:** Decodes, validates and distributes incoming telecommands. Gathers, processes and formats telemetry data to be sent.

**Attitude and orbit control subsystem:** Determines the vehicle’s orbit and orientation using sensors, and controls them using actuators which apply the required forces and torques. The signals sent to the actuators are calculated by the control laws embedded in the flight software. Sometimes these tasks are split into two different subsystems.

## 1.1 Motivation

The size and complexity of a satellite’s On-Board Software (OBSW) is steadily increasing, with satellites from the European Space Agency (ESA) now containing hundreds of thousands of lines of code [3].

Traditional software development methods based on manual coding are no longer suitable.

Even though the AOCS is inserted in the more general field of embedded systems, very few satellites are produced each year, when compared to other industries. The consequence is that the level of automation in the development process is lower than that observed in, for instance, the automotive industry. Very mature tools for automatic code generation of control algorithms already exist (*e.g.* Simulink). However, considering that those algorithms represent only about 20-30% of the AOCS software [4], there is still much room for improvement. Another issue is the discrepancy in the evolution of hardware and software. In recent years, new processors have been space-certified, bringing a significant improvement in terms of memory and processing capabilities, thus eliminating previous constraints on software complexity. Now the main limitation is the mentioned lack of high-level methods for developing more complex software.

Currently, AOCS software (as well as that concerning other subsystems) is still mostly re-written from scratch for each mission. This low-level design and development has many drawbacks, which become increasingly significant as the complexity of the system grows. Firstly, having human developers carrying out repetitive low-level tasks is highly unproductive, since they are focusing on implementation details instead of design decisions. Secondly, any human programmer is prone to make occasional errors, as opposed to computers, which only produce systematic errors which are easier to trace. Lastly, the fact that software development is carried out at a low level leads to low reusability, since the reusable elements of the system model are mere objects (if Object-Oriented (OO) methodologies are used) or even routines or sub-routines, instead of more abstract and portable concepts. In a space application, changes at this level imply the complete re-qualification of the module or routine [5], increasing the development time and thus driving costs even higher.

The described issues call for the creation of methodologies and tools which can raise the level of abstraction of the design process. By exploring the domain-specificity of the problem, one can use a DSL to create high-level models of the domain-specific software, from which low-level code can be automatically generated.

Model-Driven Software Development (MDSD) puts models in the core of the development process. It has for years been used in the branch of embedded systems as a mean of increasing productivity and reliability and decreasing time to market. Even though many modeling tools with code generation ability already exist, they are mostly oriented towards controller design, like the widely used Simulink and SCADE [6, 7]). In the space industry in particular, where the major stakeholders recognize reusability as one of the main factors influencing development costs, efforts are being made in standardizing interfaces and components, concerning both hardware and software (see Section 3.2). Even though the DSL approach has already been prescribed [8], it is a relatively new development which has not yet been properly explored in the domain of satellite software.

## 1.2 Research goals

This thesis proposes to study the feasibility of applying the DSL methodology to automate the development of AOCS software, with a focus on all the aspects of the system except for the controller itself. Such research yields interesting results, mainly to developers of space systems software and space agencies. The elimination of repetitive tasks and consequent coding errors, decrease in cost and increase in reliability are just some of the benefits that can be attained. Naturally, showing the applicability of DSLs to such a complex and constrained system would mean that it could be applied to other areas as well.

The first step towards this is to analyze both the literature on the AOCS in general and the software of a specific implementation, in order to find common functionalities, design patterns and repetitive tasks. This will allow to create a domain model representative of the system's structure and behavior and to identify which parts of the implementation can be automated or abstracted.

The development of a proof of concept follows. The product will be a DSL tailored to populate part of the formulated domain model. The idea is to create a domain-specific framework capable of facilitating the development and maximizing the maintainability. Aside from editor support for the developer, the framework must also feature a code generator which outputs readable target code.

Lastly, the proof of concept must demonstrate how the method can be extended to the remaining AOCS elements and possibly even the whole satellite software.

## 1.3 Outline

After an introductory chapter, this thesis is made up of seven other chapters, organized as follows:

**Chapter 2** provides background knowledge on the relevant topics and concepts and explains how they affect the project, namely the AOCS, MDSD and DSLs.

**Chapter 3** presents related solutions and efforts, both completed and under development, their achievements and shortcomings, and how this project fits into the state of the art.

**Chapter 4** introduces the tools used in the development of the AOCS DSL. It describes the workings of Xtext and related software.

**Chapter 5** describes the proposed solution. It presents the context of this thesis work and defines the objectives. Then, from an analysis of existent AOCS software, a design for the system model and the code generator is outlined.

**Chapter 6** provides details on the implementation of the proof of concept: developed features, challenges encountered and solutions applied.

**Chapter 7** demonstrates the capabilities of the developed software, through an example *AOCS* project.

**Chapter 8** summarizes the results and findings of this project, identifies eventual unresolved issues and recommends possible solutions and further improvements.

# Chapter 2

# Background

This chapter provides background knowledge on the topics and concepts relevant to the project, being fundamental to a proper understanding of the following chapters.

First, the AOCS is introduced, in order to understand all of its relevant features concerning structure, functionality and requirements. Next, MDSD – the software engineering paradigm on which this thesis is based – and its fundamental principles are explained. By exploring domain-specificity in the realm of model-driven approaches, one encounters DSLs, the last topic that this chapter addresses.

## 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 [9], *i.e.*, the input-computation-output process must meet strict deadlines.

In the past, orbit control was done in open-loop, with commands sent from ground stations [10]. For this reason, it is sometimes decoupled from the attitude control system, which is then referred to as the Attitude Determination and Control System (ADCS).

### 2.1.1 Units

The term ‘unit’ is used to refer to the hardware components of the AOCS. These include sensors, actuators and the processor[1]. Passive sensors, which do not have an internal processor, include Sun and Earth sensors, magnetometers and gyroscopes. On the other hand, active sensors like star sensors or GPS receivers can have an internal software matching the complexity of the AOCS [10]. Attitude control is achieved through actuators like magnetorquers and reaction wheels, which impart torques on the spacecraft. To control the orbit, thrusters (or *delta-V* actuators[2]) are used.

[1]AOCS computations can also be distributed across multiple processors [9]. For the sake of simplicity, the most common case is assumed here.

[2]Delta-V actuators are those which produce a change in velocity ($\Delta V$)

### 2.1.2 Functions

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 [10]. Measurements are collected from the sensors and used for estimation of the state of the spacecraft. Then the necessary control signals are calculated and sent to the actuators. The component responsible for the whole control signal computation cycle is called the EPC – Estimation (of the current state), Prediction (of the next state), and Control. However, due to the nature of space missions, the AOCS software must implement much more than mere control algorithms.

The system must also manage a two-way data exchange link with the ground station, as depicted in Figure 2.1. This communication links usually run through another spacecraft subsystem called Command and Data Handling (C&DH) system.

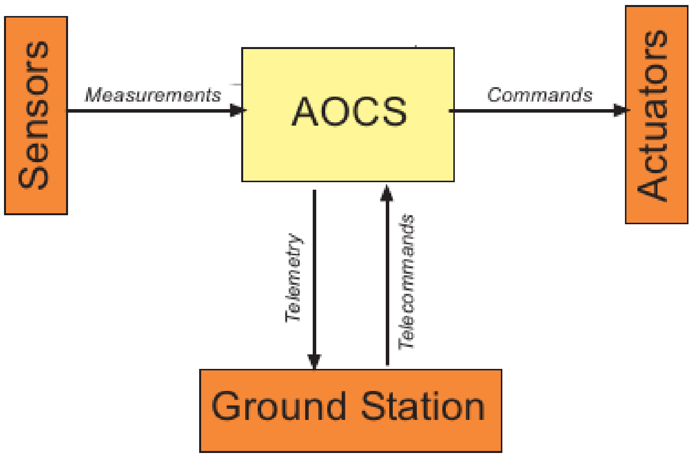


Figure 2.1: AOCS software external interfaces (adapted from [4])

The AOCS must receive and process telecommands from the ground via the uplink. These commands are used to influence the behavior of the satellite, possibly overriding internal decisions of the OBSW. On the other hand, the AOCS must provide Housekeeping Data (HKD), *i.e.*, general status information, to be sent to the ground with the remaining telemetry data of the spacecraft (*e.g.* science data), via the downlink [10]. But managing the data for uplink and downlink are only two of the functions typically carried out by the system. Other than that, the AOCS is usually responsible for managing the operational mode, executing attitude and orbit change manoeuvres and detecting failures. Figure 2.2 shows this schematically.

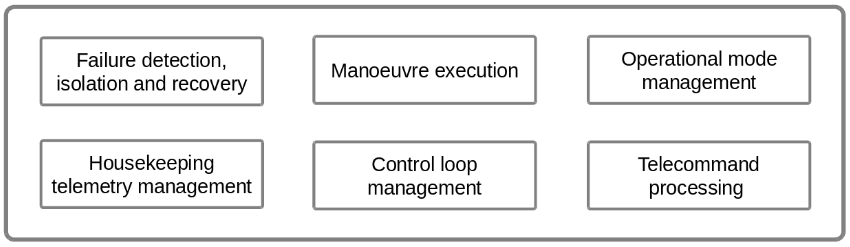

Failure detection, isolation and recovery
Manoeuvre execution
Operational mode management
Housekeeping telemetry management
Control loop management
Telecommand processing

Figure 2.2: AOCS software (based on illustration from [4])

The operational mode determines the nominal attitude and the precision at which it must be kept, the control algorithms used, and other settings. It must be managed internally because there is not a one-to-one correspondence between the operational modes of the satellite and those of the AOCS [10].

The execution of pre-programmed parameterized manoeuvres, concerning both orbit and attitude[3], is usually triggered by telecommands from the ground.

Another crucial task of the AOCS is to detect and attempt to isolate failures. Since all of the system's units – sensors, actuators and processor – are redundant, once the failure is isolated the faulty unit can be reconfigured to attempt recovery. Alternatively, the whole configuration can be changed or, if the recovery attempts are not successful, the spacecraft can enter a safe mode and wait for ground commands.

### 2.1.3 Software considerations

The design of a system as complex as the AOCS demands the use of high-level abstractions. However, its embedded nature leads to a few constraints on the abstraction mechanisms.

One of those constraints is that, due to limited computational resources and real-time constraints, the abstraction techniques used cannot lead to excessive runtime overhead, in terms of memory usage and processing speed. Object-orientation is a common technique used for raising abstraction, not only through encapsulation of variables and routines, but also through the use of inheritance [11]. Even though all OO applications introduce processing overhead due to the extra level of indirection caused by dynamic binding [10], today's space-qualified processors are fast enough to allow the use of OO languages for programming space systems [9]. Nevertheless, higher-level abstraction mechanisms are needed which can be resolved at compile time, with little or no runtime cost [11]. Such can be achieved, for instance, using DSLs (see Section 2.3).

Also, because it is a critical system, the standard for reliability is very high. Languages which provide the low-level capabilities needed in most embedded applications, like C and C++, are deemed 'unsafe' for giving the developer excessive freedom to introduce bugs in the code. That is why coding standards for embedded software, like MISRA[4] C and C++, were created and are enforced. However, the programs used to check for compliance are loosely integrated with the development environment. Using a DSL, such static verification procedures could be included in the development framework [11].

## 2.2 Model-driven software development

Model-Driven Software Development (MDSD) is a software development methodology which focuses on creating and exploiting models. Unlike Model-Based Software Development (MBSD), which uses models merely as documentation artifacts [12], MDSD puts models at the core of the development process. These models are then used to automatically generate platform-dependent designs, source code, tests, or even documentation [13]. This not only significantly increases software reuse, but also

[3]Attitude change manoeuvres are also called *slew* manoeuvres.
[4]Motor Industry Software Reliability Association

allows the developer to abstract away the implementation details, which are not relevant to the problem domain.

In the realm of model-driven approaches to software development, all the different representations of the system are considered as models, and the core mechanism of the development process is model transformation. The general principles behind model transformation are illustrated in Figure 2.3.

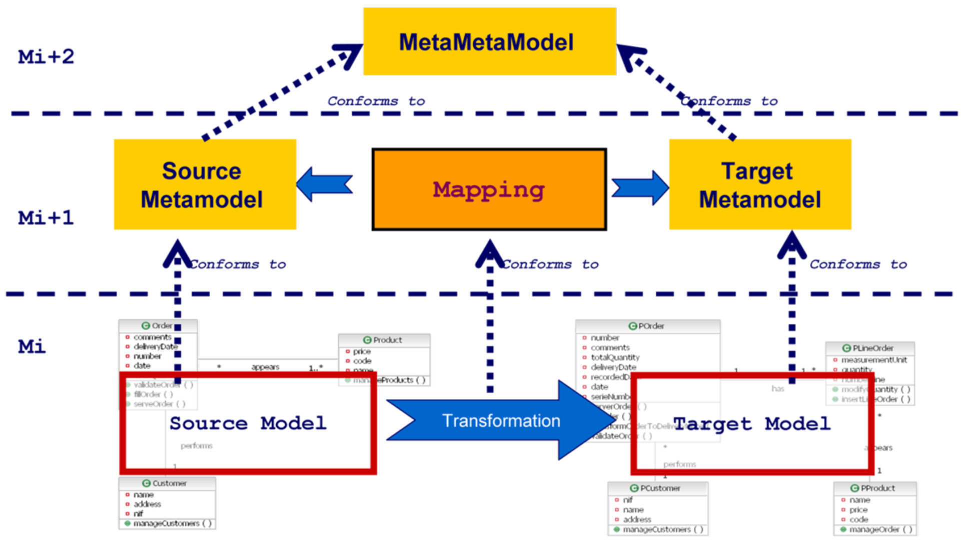


Figure 2.3: Model transformation (credits to INTEROP-VLab Education Committee)

In this thesis work, for instance, the highest level – the metametamodel – is the domain model of the AOCS, from which the source and target metamodels are derived by the developer of the DSL. In this particular case, these are, respectively, the language definition and the target code architecture and templates, while the mapping is programmed in the code generator. On the lower level, the models are the DSL script and the target code.

The higher level of abstraction allows not only a quicker development of complex systems, but it also provides an opportunity for an early verification of the system design. 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 [14]. By validating this model, which is directly linked to the target system, many of these faults can be eliminated early in the development process.

## 2.2.1 Models

A model is nothing more than an abstraction of some aspect of a system, *i.e.*, a simplification of reality. This means that it leaves out some of the details of said system, while preserving its relevant characteristics [11]. Which characteristics are relevant greatly depends on the purpose of the model. Another important concept is that of a metamodel. A metamodel defines the language used to describe a model [11], which is nothing more than an instantiation of the first [15].

A distinction can be made here between descriptive and prescriptive models. A descriptive model is one intended for communication or analysis of a system, while a prescriptive model is used to, fully or partially, automate the construction of the target system [11]. Therefore, when referring to models in the context of MDSD, one means prescriptive models. One fundamental implication of this distinction is that prescriptive models require a higher level of detail (see Section 2.2.2).

Models are created to gain a more meaningful insight into the system, more reusable software, and better integration of design, implementation and testing [16]. So for an abstraction to be truly advantageous, the models used must provide that.

### 2.2.2 Modeling languages

When modeling a software system, a fundamental point of concern is the language used. There are many “flavors” of modeling languages: graphical or textual, general-purpose or domain-specific, and other less relevant distinctions for this case. All of them offer pros and cons, and which one is the best really depends on what it is going to be used for.

The approach of this master’s thesis work is to use a domain-specific textual language. Textual languages use standardized keywords and parameters to construct a computer-readable model [17]. When a language is designed with a focus on readability, textual models can even be more understandable than graphical ones. They also make the task of a version control system much easier [18], since that model is stored in plain text files. The advantages of exploring domain-specificity are discussed in Section 2.3.

For a model to be usable in the context of MDSD, it must be complete enough to contain all the details needed for the generation of target code, whether the production of this target code is partially or completely automated (see Section 2.2.3). At the same time, it must still be simple, because a complicated model defies the very purpose of modeling.

The consequence is that many widely used modeling languages, like Unified Modeling Language (UML), are unsuitable. Due to their general-purpose nature, they don’t provide abstractions powerful enough to contain all the necessary information within the model. Attempts have been made to extend these languages to incorporate behavior modeling, so that the models can be transformed into executable artifacts. One example of that is based on UML: the Extendable and Translatable UML (xtUML). Not only is it meant to support high- and low-level code generation, but it can also be tweaked to support various targets for embedded software development [19]. However, the limitations of a General-Purpose Language (GPL) are still not tackled. The graphical models are created as diagrams of the software classes of the system as core abstractions. When dealing with complex systems, the absence of higher-level abstractions will likely lead to models not being as clear and understandable as they could be.

The solution is then to exploit the domain-specific characteristics of the problem at hand. An application domain comprises a set of software applications with common features. By capturing those features, one can create a domain-specific modeling language (or metamodel) which offers to developers powerful concepts and notations tailored to capture the characteristics of that same domain, while allowing the specification of a concrete system [12]. Bundled with a reference architecture common to all applications within the domain, a code generator, and a library of reusable domain components, a framework is created that lets the developer work at the level of abstraction of the domain [20], with more meaningful concepts. The topic of DSLs is covered in Section 2.3.

### 2.2.3 Code generation

The code generator is one of the essential components of a MDSD tool. It maps an abstract input model into executable target code. This process is depicted in Figure 2.4, and is nothing more than a specific case of the general model transformation shown in Figure 2.3. Code generators are also referred to as compilers.

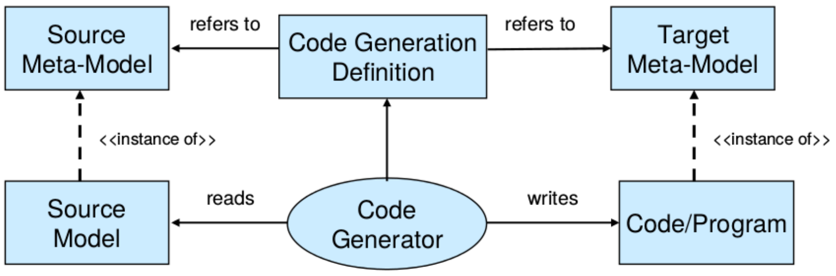


Figure 2.4: Code generation (from [21])

Several types of code generators exist, using very different paradigms. However, within MDSD, only metamodel-based generators are relevant. More important is to summarize the guidelines for its design that can be found in the literature.

First of all, it is important to notice the gap in the level of abstraction between the input model and the generated code. The consequence of this gap is that the code contains much more detailed information about the system. Therefore, the transformation can only be unidirectional, *i.e.*, changes in the code cannot be reflected in the model. So the generated code cannot be edited. Guaranteeing that there will be no need for that is a responsibility of the designer of the code generator. Also, if that gap proves to be so wide that it forces the code generator to be excessively complex, then it is probably a good idea to create intermediate representations using Model-to-Model (M2M) transformations. In practice, a single transformation is sufficient, if needed [22].

However, the fact that the generated code should never be hand-edited does not mean that it should not be readable. In fact, at times developers may want to know how it works, so producing clean code is a key principle [18].

Due to the complexity of the AOCS software, there is a lot of code to be generated concerning the architecture of the system, which is common to all applications in the domain. In that case, when the dynamic code to be generated represents a small amount of the total code, a generation mechanism based on code templates is the most appropriate [18]. Also, template languages offer a good syntactic mix of model traversal code and to-be-generated code [11].

The last consideration concerns testing. Although one of the biggest benefits of MDSD is possibility to automate test generation, it is not addressed in this project, for the simple reason that dedicated software testing methods for the AOCS already exist.

### Generation gap pattern

In very simple domains, it is possible for a system to be completely described with an abstract model. However, for many real-world systems, it is not possible to abstract away some of their features. In this case, code will have to be manually written to complement the output of the code generator, and the integration of automatically generated and hand-written code becomes an issue [11].

The most widely accepted way to introduce code separation is the GGP, which prescribes that generated and hand-written code be separated using inheritance [18]. Obviously, this requires that the target language be Object-Oriented. This pattern is shown in Figure 2.5. The generated classes implement

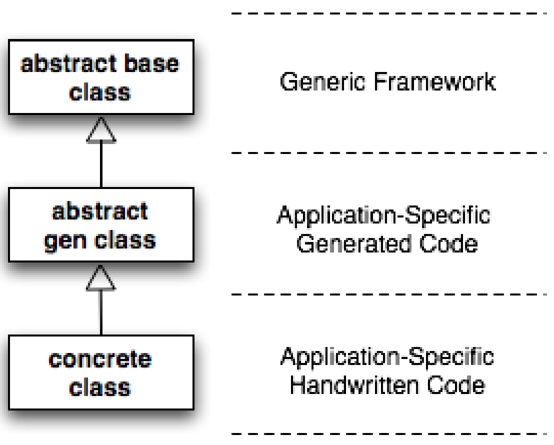

abstract base class
Generic Framework
abstract gen class
Application-Specific Generated Code
concrete class
Application-Specific Handwritten Code

Figure 2.5: Generation Gap Pattern (from [22])

default, and possibly incomplete, behavior, which can be augmented or overridden by a concrete subclass. When using the GGP, a good practice is to make use of the compiler of the target language to guarantee that the necessary manual code is implemented [22]. By creating abstract methods in the generated classes, the compiler will force the developer to create a subclass which implements those methods.

Although this pattern hints on how to split the code in the software structure, it makes no recommendation on how to separate the corresponding files in the project directory. A common approach is to put generated code in one folder (usually named *src-gen*) and have the developer save manual code in another (usually named *src*). However, this requires that the developer looks for which classes to subclass and which methods to override. Further support can be given by automatically generating class and method stubs[5] in the *src* folder. The key here is that, while the content of the *src-gen* folder is regenerated every time the generator is run, files in the *src* folder are only generated if they do not already exist [23]. In summary, stub files are therefore only generated once, with the intent of providing a scaffolding for developer to manually implement the missing functionality. Because of this, the pattern is referred to as the *generate once* policy.

[5]A stub is a temporary replacement for code that still has not been developed. It is commonly used as an implementation of a known interface, when the actual implementation is not known.

## 2.3 Domain-specific languages

A Domain-Specific Language is defined by Martin Fowler as follows [18]:

> Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain.

Unlike general-purpose programming languages like C/C++ or Java, a DSL is limited to, but optimized for, use in applications within a given domain. The goal when building a DSL is to create a programming (or modeling)[6] language with concepts and notation closely related to the domain, in order to meet the way that software developers reason in said domain [24]. It trades generality – most DSLs are not Turing complete [11] – for abstractive power.

### 2.3.1 Benefits

DSLs allow system design to be done at the level of abstraction of the domain, which has numerous advantages:

**Shorter programs:** It has been shown that the amount of code alone introduces complexity in a program [11]. The high-level character of a DSL means that a possibly very elaborate concept can be instantiated with a small amount of code [25].

**More readable models:** Readability is greatly improved by the usage of domain notation in the language.

**Easier communication:** The communication between software developers and domain experts can be a limitation in a project [26]. A well designed DSL can make the work of the developer understandable to the expert [27].

**Higher reliability:** A DSL frees the developer of the clutter of implementation details and repetitive tasks. These low-level tasks are not only a waste of the time and capabilities of a human designer, but they are also error prone. Unlike a computer, a human programmer is bound to make occasional errors like missing references, typos and use of uninitialized variables. This is a result of the often unnecessary degrees of freedom offered by a GPL [28].

**Earlier validation:** A DSL is a metamodel, which not only captures the domain concepts, but also the domain-specific constraints and rules [15]. As such, it can be used to enforce compliance with standards [29], like those seen in the embedded systems industry, and also to apply model validation checks early in the development process.

**Possibility for further automation:** Not only target code can be generated from the model. Automated tests and documentation artifacts are also a possibility [30].

[6]In the general area of Model-Driven Engineering, modeling and (high-level) programming are often considered synonyms.

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 [28].

### 2.3.2 DSL processing

In order to understand how DSLs work, it is important to introduce a few key concepts. The first of those concepts is the grammar. 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 [18]. In other words, it defines the concrete syntax (or textual representation) of the language. With knowledge of this concrete syntax, the parser[7] is able to create a model based on its abstract syntax, *i.e.*, its meaning. This model is therefore called the Abstract Syntax Tree (AST), or alternatively semantic model. It is a data structure that holds the meaningful information expressed in the script [18]. From this model, a number of optional artifacts can be automatically generated, namely target code, which is the end goal of this project. An intermediate model can also be generated via a M2M transformation, to reduce the complexity of the code generator. This whole process is depicted in Figure 2.6.

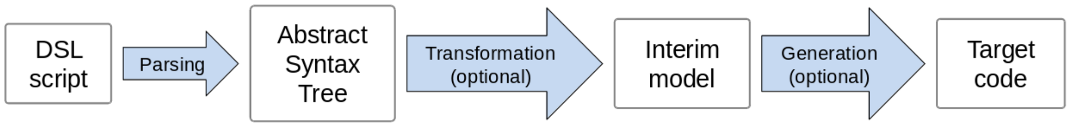


Figure 2.6: DSL processing

An important note here is that 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 [18].

### 2.3.3 Distinctions

Many different types of DSLs exist, and understanding their differences, and the pros and cons of each, is essential for defining how a new language will be designed. Once again, it all depends on the intended application.

**Internal or external:** External DSLs featured their own grammar and parser, while internal (or embedded) DSLs are built within a pre-existent host language. Internal DSLs basically shape their host language for domain-specific purposes [31], by adding domain-specific concepts and by restricting the use of the host language to a subset of its elements (see Figure 2.7). While they present the advantage of reusing all the infrastructure of the host language (editor, compiler, *etc.*), they are also bound to its syntax and constraints. Perhaps more importantly, internal DSLs lack custom content assist and other forms of editing support. The boundary between an internal DSL and an Application Programming Interface (API) is very fuzzy and falls beyond the scope of this thesis, but [11] provides a good distinction between the two. External DSLs, on the other hand,

[7]The term ‘parser’ is used to refer to the entire lexer-parser-linker toolset. More details on that in Section 4.3.

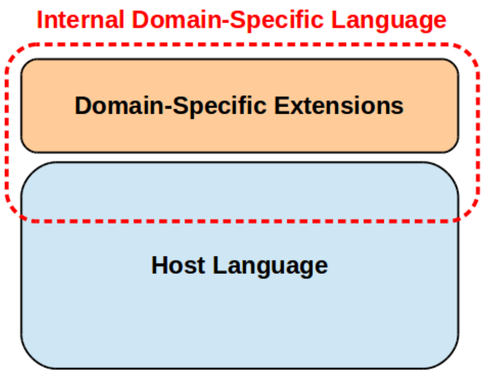


Figure 2.7: Internal DSL

provide the cleanest form of abstraction [11]. Although they require the implementation of a full language infrastructure, a lot of unnecessary clutter can be avoided, as a result of the full control over the design and notation of the language. Also, custom domain-specific Integrated Development Environment (IDE) support can be bundled with the language, which allows to better guarantee that the developer follows rules and conventions, of the domain or of a product family [28]. The downside of creating an external DSL is that it requires the construction of a whole framework, including a parser, a validator and a code generator. The creation of a custom parser (plus the required lexer and linker) alone is already a complex task, only at the reach of language specialists [32]. However, the continuous maturation of language workbenches, which provide support for creating these elements, has been mitigating this problem. External DSLs will be addressed in detail throughout this thesis.

**Graphical or textual:** When designing a DSL, one of the key considerations concerns how to edit a program: graphically or textually. Graphical visualizations can be created from a textual model and a textual representation can be generated from a a graphical model through projection, so the concern really lies on how to edit it. While a graphical model provides a better overview of a system, a textual modeling language is easier to develop [33]. The choice must be based on what feels more natural in the domain. For instance, state machines are usually more intuitive when expressed with a graphical notation (see Figure 2.8). One advantage of textual DSLs is that the concrete syntax of the language corresponds to the way the model is stored, thus improving migration of models to other projects and platforms [22]. Another significant advantage concerns team work, as plain text files integrate much better with version control systems [18].

**Technical or application domain:** DSLs can be categorized according to their targeted users. [11] defends that a technical DSL is meant to be used by a programmer, *i.e.*, someone with knowledge of software development, while an application domain DSL is intended for domain users. In the case that a very simple and complete language can be created for a given domain, this distinction can be made in terms of internal and external DSLs [27]. However, this does not have to be the case, and the AOCS DSL (see Chapter 5) constitutes a good counter-example. With additional development, it is theoretically possible to evolve it into a simple enough language to be used by

```
events
 doorClosed D1CL
 drawerOpened D2OP
 lightOn L1ON
 doorOpened D1OP
 panelClosed PNCL end

resetEvents
 doorOpened
end

commands
 unlockPanel PNUL
 lockPanel PNLK
 lockDoor D1LK
 unlockDoor D1UL
 end

state idle
 actions {unlockDoor lockPanel}
 doorClosed => active
end

state active
 drawerOpened => waitingForLight
 lightOn => waitingForDrawer
end

state waitingForLight
 lightOn => unlockedPanel
end

state waitingForDrawer
 drawerOpened => unlockedPanel
end

state unlockedPanel
 actions {unlockPanel lockDoor}
 panelClosed => idle
end
```

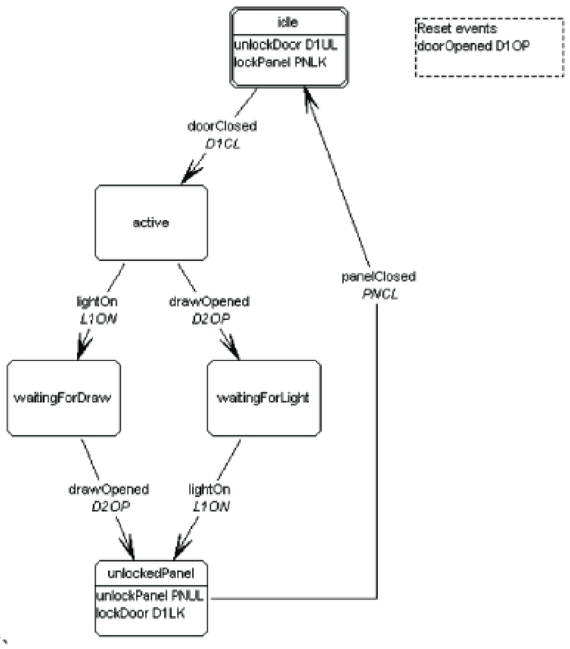


Figure 2.8: Textual and graphical representation of a state machine (from [34])

domain users with relatively little knowledge of programming, but the amount of work required to achieve that may overweight the advantages.

### 2.3.4 Design guidelines

Language design is a cumbersome task, usually performed by specialists. Today's language workbenches provide a valuable help, allowing less experienced, but adventurous developers, to create their own DSLs, either with a specific application in mind, or just for didactic or recreational purposes. However, by following some basic guidelines, the need for trial-and-error may be greatly mitigated. This section presents a compilation of those guidelines, found in the literature.

First of all, it is important to understand that a DSL is just one of the elements of a complete domain framework. The Domain-Specific Modeling (DSM) methodology, which concerns the development of such a framework, prescribes the following ingredients [20]:

- Domain model.
- Domain-Specific Language.
- Code generator (or any other kind of model transformation tool)[8].
- Library of reusable components.

This is important because considerations about other components can (and usually do) influence the design of the language, since the domain framework is developed as a whole in a language workbench.

[8]Alternatively, the model can also be directly executed by an interpreter. Interpreters are often used for simulation of the system, but simulation falls beyond the scope of this project.

One important point is to properly separate the domain concerns. As stated before, the domain model should not be entirely reflected in the language. A Domain-Specific Workbench (DSW) also contains a reference architecture and an execution engine which can complete the model of a system, thus allowing said model to be simpler. According to [11], this separation should be done as follows:

- The elements or features which are invariable, *i.e.*, equal for each program in the domain, belong to the reference architecture.
- Those that can be derived by fixed rules from an input model belong to the execution engine, meaning that they will be inferred by the model transformation mechanisms.
- Only the set of variable features that cannot be derived should be included in the DSL, which must in turn provide the adequate abstractions to express them.

When performing this separation of concerns, one important point to consider is that, when using a model-driven approach to software development, a developer must be able to make all decisions concerning design at the model level [29].

Considering that one of the main points of MDSD is to increase reuse of software, one commonly found advice is to reuse existing languages when possible, either by extension or composition. Even if a new language must be created, [33] prescribes that an existing type system be reused in order to reduce the workload of the developer and to promote acceptance by the user. Also with the intent of easing the adoption of the language, the notation used by the domain experts should be present in the language.

DSLs are designed to make the models simpler. As such, unnecessary complexity must be avoided. The language should reflect only the necessary subset of domain concepts. Also the number of language elements (the size of the language [11]) should be kept to a minimum. Excepting a few very simple application domains, this requires either the use of libraries or a modular language [33]. In a modular language, modules would augment, and be translated to, a small core language, adding its own IDE support. Then, other modules can be added and translated to the first-level modules, and so on. A very promising example of a modular language is presented in Section 3.3.3. This approach is related to a programming paradigm described by Martin Ward [25], called Language-Oriented Programming, which is based on creating and extending DSLs to solve each problem at the appropriate level of abstraction. However, this approach has not been followed in this project. Explanations about this and other design choices are given in Chapter 5.

# Chapter 3

# Related work

This chapter provides an overview of what has been done so far that could contribute to solving the problem of reusability in AOCS software. Related work includes developments made towards the same goal or with a similar methodology, since results can be applied and adapted to overcome eventual shortcomings. The following sections will show, through real examples, that this project addresses a real and current problem, and that it complements previous research with a unique approach. This review is also important to establish a basis for comparison and evaluation.

As mentioned in Chapter 1, graphical modeling tools like Simulink (The MathWorks) and SCADE (Esterel Technologies), despite their widespread use in aerospace and other industries, are focused in the development of controllers, and not of whole software systems. Even though they may provide the means to integrate a software architecture described with UML or SysML[1] class diagrams, the system models are created externally and so these tools will not be reviewed here.

Section 3.1 presents a project addressing the exact same problem, but following a different (pre-DSL) paradigm: a software framework. Section 3.2 follows, with a description of the current efforts in the space industry concerning reusability and standardization. Finally, DSL-based solutions already developed for other kinds of embedded systems are discussed in Section 3.3.

## 3.1 The AOCS framework project

The AOCS Framework Project was carried out at the turn of the millennium in the Computer Science Department of the University of Constance, in Germany [35]. It presents a related but different approach to AOCS software development, from before the popularization of DSLs. The documents written with respect to the project also show the most in-depth practical study of the system to be found, and are used as references on the matter throughout this thesis.

[1]Systems Modeling Language.

### 3.1.1 Concept

The concept behind the AOCS Framework Project is that of a software framework. It is natural that applications within the same domain will have similar high-level requirements, which in turn translate to a common structure at software level. The point of a framework is then to create an architecture that captures these common requirements and optimize it for the target domain, thus allowing one to easily and quickly create an instance of such a system [26].

The AOCS Framework intends to build a complete system as a composition of independent components [9]. The key here is how they are connected to the well-defined framework. The framework has specific hooks[2] for each kind of component, and any component which is attached to these hooks to extend (or override) functionalities must implement certain abstract interfaces, *i.e.*, sets of related operations which are declared, but not implemented [26]. This means that implementation of system-level functionalities at component-level is left up to, and demanded from, the component developer. Figure 3.1 shows a schematic example.

This interface-based approach naturally requires the use of an OO methodology[3].

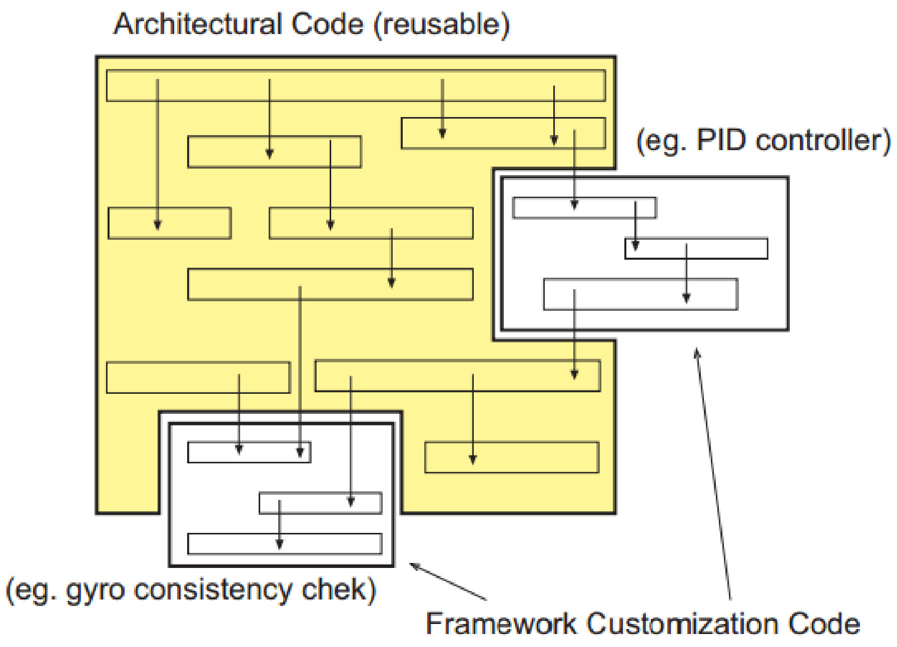

Architectural Code (reusable)
(eg. PID controller)
(eg. gyro consistency chek)
Framework Customization Code

Figure 3.1: The AOCS framework approach (from [10])

#### Customization code

Once the AOCS is properly understood and the framework is created, another concern is how to develop the customization code. Of course, hand-coding the components is always an option, but one of the priorities of this project was to allow integration of automatically generated code. This because it was felt that “autocoding” tools, as the authors call them, have strengths and weaknesses complementary to those of framework technology [5, 10] :

[2]In programming, a hook is a place, usually accompanied by an interface, for a programmer to insert custom code.
[3]The framework was initially developed in C++ and later ported to Java [4]

**Architecture design:** The graphical modeling tools used for code generation do not generally facilitate architecture design, which is captured by the framework.

**Generality:** Frameworks are built specifically for an application domain, while graphical modeling tools feature component-level abstractions of several domains, mostly concerning controller design.

**Simulation:** Code generation tools are typically created for development environments intended for algorithm design and simulation, providing a good integration of development and validation. On the other hand, a framework, being defined at code-level, and representing a usually incomplete system, is not simulation-capable.

The synthesis process proposed consists of the merging of an architectural skeleton provided by the framework and mission-specific components provided by the code generators of dynamical simulators [10].

### Integration strategy

To allow the complete instantiation of a system, a configurable integration of different resources is still necessary. The end goal of the AOCS Framework is then to extend it in order to provide such integration, as depicted in Figure 3.2. Is this example, the widely used modeling software from The MathWorks was chosen to illustrate the strategy.

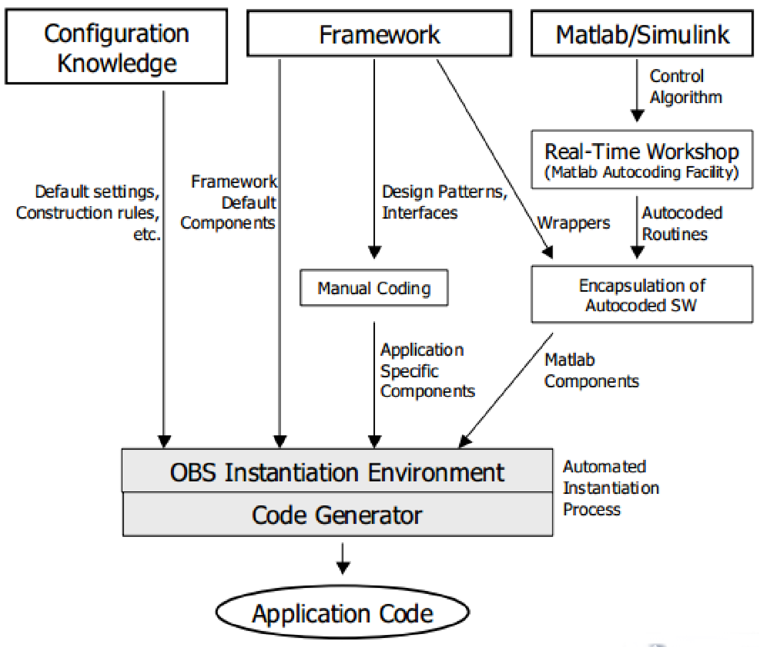

Configuration Knowledge
Framework
Matlab/Simulink
Control Algorithm
Real-Time Workshop (Matlab Autocoding Facility)
Default settings, Construction rules, etc.
Framework Default Components
Design Patterns, Interfaces
Wrappers
Autocoded Routines
Manual Coding
Encapsulation of Autocoded SW
Application Specific Components
Matlab Components
OBS Instantiation Environment
Code Generator
Automated Instantiation Process
Application Code

Figure 3.2: Proposed development process (from [36])

### 3.1.2 Conclusions

To sum it up, this project aims at creating a platform for the development of reliable software systems, by customizing a generic architecture with unit-verified components compliant with a well defined interface. However, despite representing a great advancement in the field of AOCS software development, there is still room for improvement in the AOCS Framework.

First of all, note that a domain-specific architecture is by default embedded into a DSL by the process of domain engineering (see Section 2.3), so the framework approach results in just a subset of what can be achieved with a complete domain-specific development platform.

The customization of the AOCS Framework must be done either at code-level or with separate modeling tools, while a DSL allows further automation of the process to be integrated in the platform, by providing high-level abstractions for the variable part of the system. This leads to very readable models which more clearly show the relation between the fixed architecture (defined by the language structure) and the application-specific parts (specified using language constructs).

In other words, domain-specificity can be explored to automate recurrent design patterns beyond the architecture. In fact, a DSL can even be developed to the point of including low-level concepts, eventually allowing it to completely express systems like the AOCS.

## 3.2 Standardization of space systems

There are ongoing efforts in the space industry to standardize the development of space avionics, concerning both hardware and software. This section highlights the software-related outputs of those projects, to raise awareness of the current problems and needs of the industry and to understand how this thesis proposes to meet them.

As stated before, reusability is a big concern within the space community these days, and the projects presented in the next sections share a common end-goal: to decrease time and cost of development of space avionics, by reaching a development paradigm with the following features [37]:

- A reference architecture which captures the basis of all the subsystems in the domain.
- Standard interfaces which allow easy integration of components.
- Reusable components conforming to the standard interfaces.

The proposed modularity of space systems comes with the downside of affecting performance when compared to highly optimized designs. However, the performance of modern hardware allows that sacrifice to be made. In other words, it is becoming a viable option to produce software systems which can be built much faster, with a ‘good enough’, instead of optimal, performance, as long as the required reliability is assured [38].

### 3.2.1 Satellite plug-and-play avionics

Conducted by the North-American Air Force Research Lab, the Satellite Plug-and-Play Avionics (SPA) project aims at adapting Plug-and-Play (PnP) approaches for use in space systems. Commonly seen in personal computers and other everyday electronics, PnP technology has resulted in the proliferation of cheap and easily integrable devices, inspiring its use in more ambitious applications like critical systems [39]. The principle behind PnP is to design systems based on an interface-driven set of standards. Since compliant components are platform-independent and seen by the system as a black-box[4], their integration stops being an issue and the consequence is quicker development and higher reusability.

On the software side, the project involves developing a middleware layer called Satellite Data Model (SDM). This layer aims at promoting software reuse at the application level [38] by decoupling the application software from the execution platform, as shown in Figure 3.3. The applications then interact with the spacecraft components through API calls to the middle layer. A demo satellite called PnPSat was developed but never launched [40].

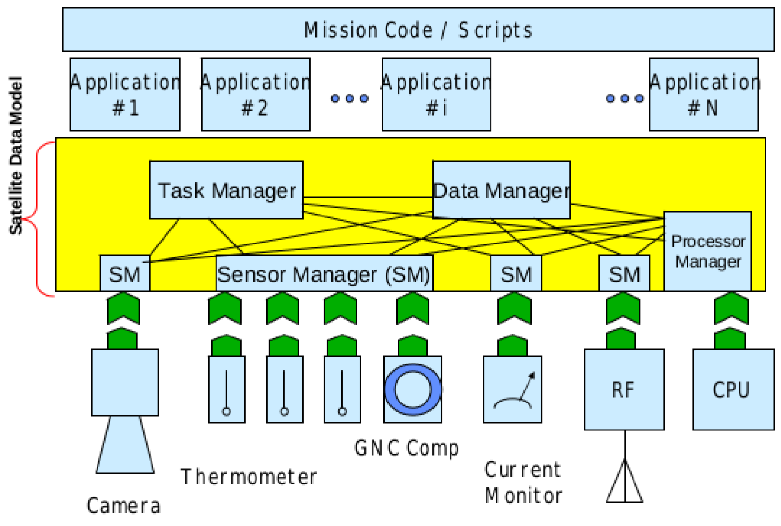


Figure 3.3: SDM in PnPSat (from [39])

In the SPA project, nothing is prescribed on how to develop these components and applications, as the project focuses solely on the interfaces. Therefore the usage of DSLs is not excluded.

### 3.2.2 SAVOIR

Taking inspiration from AUTOSAR, an open and standardized automotive software architecture, the Space Avionics Open Interface Architecture (SAVOIR) is an initiative to improve the way that the European space community builds and develops avionics. It is coordinated by the SAVOIR Advisory Group, which is comprised of three working subgroups and includes representatives from ESA, Astrium, Thales, DLR and other major stakeholders in the community [37]. The subgroup SAVOIR-FAIRE[5] is in charge

[4]A black-box is something that is viewed only in terms of input and output, with no knowledge of its internal workings.
[5]Fair Architecture and Interface Reference Elaboration.

of creating a standardized architecture for OBSW.

Departing form the identification of the main avionics functions of spacecraft, the group proposes to develop a standard interface between them, so that components can be more easily developed and reused across different projects [37]. The result is a software architecture based in the separation of application software and execution platform (like the one seen in Section 3.2.1), named Component-Oriented Development Techniques (COrDeT). Model-Driven Engineering (MDE) (see Section 2.2) and component composition are the other foundational principles of this architecture. The high-level structure of the second version of COrDeT is depicted in Figure 3.4.

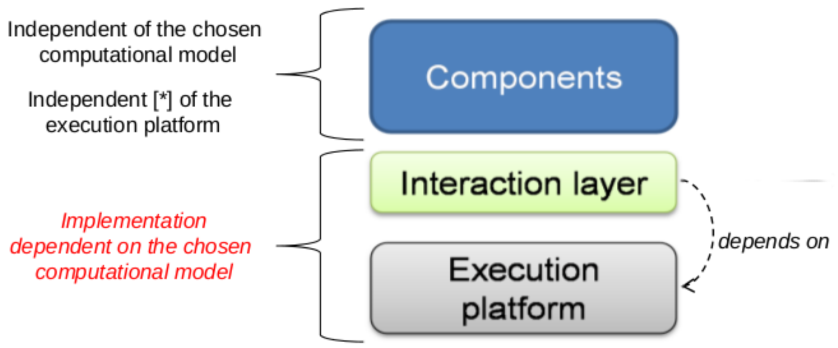

Independent of the chosen computational model
Independent [*] of the execution platform
Implementation dependent on the chosen computational model
Components
Interaction layer
Execution platform
depends on

Figure 3.4: COrDeT-2 high level architecture (from [8])

But SAVOIR goes one step further than its North-American counterpart, to provide a set of possibilities on how to develop the interface-compliant components. One of those possibilities is described in [8] as *Scenario 1.b-2*. It prescribes the use of a DSL and an associated toolset to generate an Ecore-based metamodel (see Section 4.2) with the full specification of the components.

### 3.2.3 Conclusions

The above-presented projects are very similar in their motivations and goals. Both promote the platform-independence of application software, both defend a standardization of the system architecture and component interfaces, and both have as an end-goal the faster development of space avionics by means of component reuse. The difference is that, while SPA relegates the development methodology of the components to their developers, SAVOIR defends the possibility of a DSL-based development, thus confirming the pertinence that this thesis assumes in the current context of the space industry.

## 3.3 DSL-based solutions

With the recent and fast maturation of language workbenches, DSLs are now a relatively easy way to implement a modeling platform tailor-made to address the specific problems faced by each industry or manufacturer. As a result, several renowned institutions and established companies have been turning to DSM, searching for a valuable increase in productivity.

This section addresses the most notable examples of DSL-based solutions, developed and under development, and their results. These include a language for programming and processing magnetic measurements, a language for programming refrigerators, and an ambitious tool for embedded software development based on domain-specific extensions for the C programming language. The purpose of this section is to raise awareness to the most recent developments and set the bar for the contributions that this project aims to make.

### 3.3.1 Magnetic measurements at CERN

The measurement systems at the laboratory of the European Organization for Nuclear Research (CERN), having been developed one by one with traditional software development methodologies, lacked a separation between the generic and specific code, necessary to achieve satisfying levels of software reusability and maintainability. Carlo Petrone addressed this issue in his thesis [15], by creating a DSL to be integrated in CERN's also recent Flexible Framework for Magnetic Measurements (FFMM). The goal was to provide an easy and flexible way to define test procedures, synchronize measurement tasks and configure the involved instruments.

```
    //*************************************/
    //Variable assignement
    //*************************************/
    AcquisitionBufferSize = numberOf_FDI*( SamplePerTurn/2)*4*2;
    //*************************************/
    // Device Definition
    //*************************************/
    DEF ENCODER_BOARD:  Enc_B      WITH ( "1" , "1","CERN" ) ;
    DEF FDI_CLUSTER:    Cluster_1  WITH (numberOf_FDI  );
    DEF KEITHLEY2K:     Mult_M     WITH ( "1", "2", "NI") ;
    .
    .

    //*************************************/
    // Device Configuration
    //*************************************/
    CFG ENCODER_BOARD: Enc_B       WITH ( Encoder_bus , Encoder_slot ) ;
    CFG FDI_CLUSTER:   Cluster_1   WITH ( Cluster_bus , Cluster_slot ) ;
    CFG KEITHLEY2K:    Mult_M      WITH ( Multimeter_intfNum, .
    .
    .
    .

    //*************************************/
    // Device Setting
    //*************************************/
    CMD FDI_CLUSTER: Reset ( Cluster_1, 0);
    CMD FDI_CLUSTER: Reset ( Cluster_1, 1);
    SET FDI_CLUSTER: Params2 ( Cluster_1, spt1, SamplePerTurn, Cluster_abs_gain_,
Cluster_comp_gain_, CONT, 500000, spt2,10);
    SET FDI_CLUSTER: Stop_Source ( Cluster_1, surceStop);
    CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 0, 1.0);
    CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 1, 1.0);
```

Figure 3.5: Portion of a MDSL script (from [15])

The Measurements Domain-Specific Language (MDSL) has been designed for non-programmer users, allowing them to create their own measurement applications. The simplicity of the resulting language can be seen in the code snippet in Figure 3.5. Having added editor support like completion proposals, the author highlights the ease of use of his DSL.

### 3.3.2 Refrigerator programming at BSH

Even though the basic refrigerator cooling algorithms are already well known at Bosch and Siemens Home Appliances (BSH), the continuous search for better efficiency requires some trial and error. However, testing new ideas was time-consuming [30]. So BSH hired itemis, a software company focused on MDSD, to develop a DSL-based solution which would allow the domain experts to quickly test new ideas.

The domain experts (and end users) were involved in the process and the result was a set of three independent but cooperating Xtext-based (see Section 4.3) DSLs: one for defining variants within the product family, one for defining the cooling algorithms and one for testing those algorithms.

The team created an interpreter-based simulator to speed up the testing iterations and a C code generator to automatically implement the algorithms in the form of production code for the target devices [30]. The industrial standards for the embedded code were taken into account when developing the code generator, so there is no need for the experts developing the cooling algorithms to worry about them, instead focusing on design decisions. In addition, the team at itemis provided the platform with the ability to generate additional documentation, like state charts, flow charts and lists of relevant requirements for each product variant.

The project was presented at the EclipseCon Europe 2011 conference, in Ludwigsburg, Germany.

### 3.3.3 Mbeddr

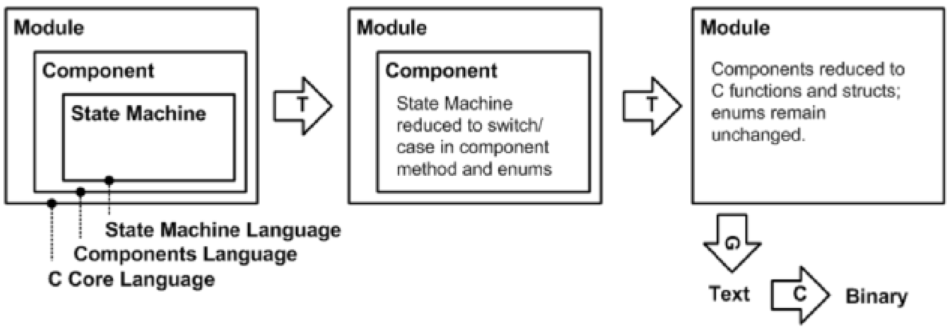
Developed by a team mostly resident at itemis, using the JetBrains MPS language workbench, mbeddr is a tool targeting embedded software development, basically consisting of a stack of DSL extensions to a C core.

Most real systems cannot be completely described with a single tool or DSL, and the integration of hand-written code and code generated by several different modeling tools can be a cumbersome task [41]. With this in mind, the team responsible for mbeddr decided to tackle the challenge by searching for a tight integration of general-purpose code and domain-specific models. By directly embedding the extensions into C, mbeddr proposes to simply remove that integration challenge [11].

Mbeddr has taken a language-oriented approach to become a prime example of a modular language, as described in Section 2.3.4. Each language module is built on top of other modules (or directly on top of the C core), extending them with higher-level constructs and abstractions, syntax, and IDE support. The compilation process therefore becomes a chain of relatively simple reductions from high- to low-level languages, eventually leading to a program completely written in C. The consequence is that all transformations (or reductions) are reused by the language modules above [11], and that no runtime cost is paid for the abstractions. The compilation process is exemplified, with standard extension constructs, in Figure 3.6.

The advantages of giving the developer the possibility to choose at which level of abstraction to solve a particular problem are numerous, including [41]:

- Legacy code written in C can be easily integrated.
- C's constructs can be used to program efficient low-level operations, while the extensions can be used to get rid of implementation details at design-level.
- C, being a low-level, general-purpose, Turing-complete language, is very difficult to analyze statically. Such static analysis can be done at the level of the DSL extensions, which are more restricted



Module
Component
State Machine
T
Module
Component
State Machine reduced to switch/ case in component method and enums
T
Module
Components reduced to C functions and structs; enums remain unchanged.
State Machine Language
Components Language
C Core Language
G
Text
C
Binary

Figure 3.6: Mbeddr compilation process (from [41])

by nature.

Figure 3.7 shows the stack of default extensions. Exceptions to the language stacking principle are the support for requirements traceability and product variability, which are common to all embedded software and provide the same functionality at all levels of abstraction. Additionally, a developer can create his own custom extensions to further expand the stack [11].

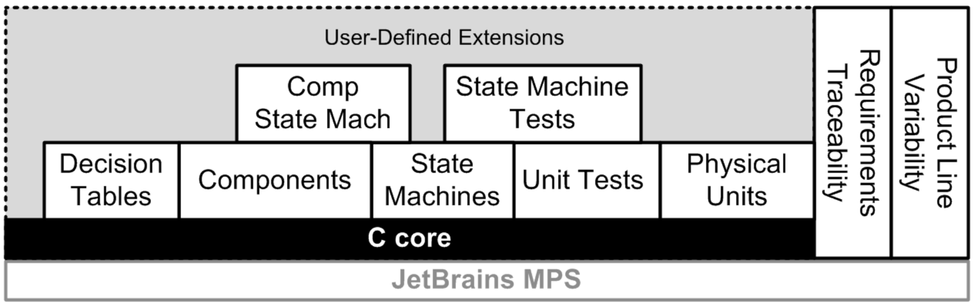


Figure 3.7: mbeddr (from [11])

### 3.3.4 Conclusions

The team behind mbeddr proposes to create a single environment where the complete software of an embedded system can be developed. Mbeddr is a very ambitious and promising project. It already provides a basis for all embedded systems and an extensibility mechanism. In the future, with the development of a library of more specific, higher-level extensions, it might get to see widespread use in embedded systems development. However, for now it only provides some basic extensions, and directly creating a tailored language from scratch may still be the best option in some cases.

Creating and using DSLs for a specific task has been proven feasible and very useful by the examples in Sections 3.3.1 and 3.3.2. But the AOCS is a significantly different and far more complex system, so the approach taken here cannot be quite the same. The AOCS DSL cannot, at least at first, reach the level of simplicity to allow its use by non-programmer domain experts. This challenge is exactly what makes a DSL-based solution for the AOCS a valuable contribution, to space software and to overall embedded software developers.

# Chapter 4

# Development tools

In this chapter, the tools used in the development of all the components of the AOCS DSL are presented. The whole DSW is created using the Xtext workbench. However, since it is integrated in the Eclipse IDE and uses Eclipse Modeling Framework (EMF) to create and manipulate models, an introduction to these software tools is also necessary.

## 4.1 Eclipse

Eclipse is a software IDE with a base workspace customizable by means of an extensible plug-in system. Plug-ins are the smallest units of extension of the Eclipse platform [42]. Existent plug-ins allow it to be used as a development environment for several commonly used programming languages, most notably Java, C/C++[1] and PHP.

One of its key benefits is its ability to integrate several tools in a single environment. For instance, by installing the development tools for Java and C/C++, it becomes a development environment for both languages simultaneously. In order to tag the tools which a certain project uses and requires, Eclipse uses a project nature mechanism. A project can have multiple natures, meaning that it will be shared by the correspondent tools [42]. For instance, a project that uses an Xtext-made DSL for developing C/C++ source code will have the Xtext nature and the C/C++ nature.

The User Interface (UI) of Eclipse is based on three concepts [42]: editors, views and perspectives. An Eclipse window consists of a certain arrangement of editors, used to browse and edit files (or resources), and views, which are visual components used to navigate sets of data. A perspective is just an arrangement of editors and views optimized for a specific task.

Another valuable feature of Eclipse is its easy integration of version control systems and associated team repositories.

[1]Since C++ is nothing more than a superset of C and the development tools for both are usually bundled together, sometimes they will be referred to as if they were a single language (C/C++).

## 4.2 Eclipse Modeling Framework

The Eclipse Modeling Framework is a framework for the development of model-based software. It facilitates the creation of models and metamodels and provides automatic generation of Java code [24], thus simplifying the development of complex applications [43].

The core component of EMF is Ecore [11], which is basically a metamodel for describing models (or lower level metamodels). Ecore models are built from a small set of concepts, each of them directly related to a Java concept. The relations between them are shown in Figure 4.1. An EPackage is a

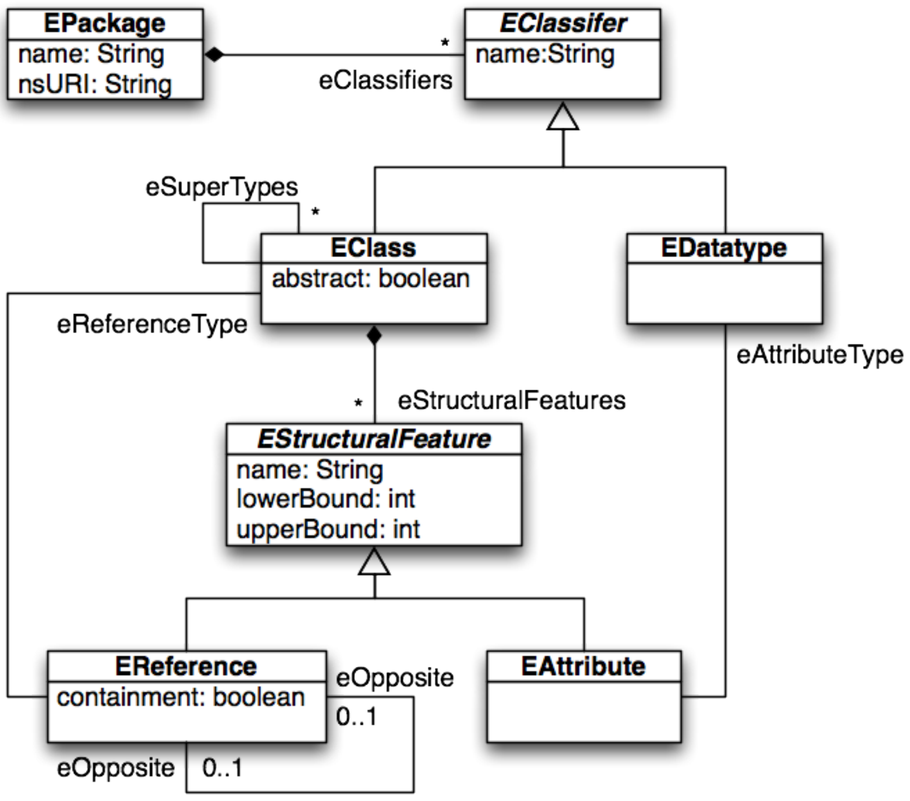


Figure 4.1: Basic Ecore concepts (from [24])

container of information, equivalent to Java's package and C's namespace concepts. An EClass is, like classes in any OO programming language, a template for the creation of objects (EObjects in EMF), containing an arbitrary number of attributes (EAttributes) and references to other classes (EReferences). EStructuralFeature is just a common interface. References can denote association or containment relations. Attributes are defined by a name and a type (EDatatype) [24].

EMF's code generator creates Java classes from an additional metamodel called generator model (with extension *.genmodel*) [24]. This model has the same structure, but its elements have additional properties which facilitate the generator's task.

It is important to note that EMF does not model behavior. The implementation must therefore be provided as hand-written Java code, which can then be included in the model [24]. Despite this limitation, EMF still provides an always helpful separation of concerns.

## 4.3 Xtext

Xtext, as has been pointed out, is an open source language workbench consisting of a set of plug-ins to the Eclipse IDE [24]. It provides the tools for developing and using DSLs[2], alongside with their UI features [43].

The first step towards developing a DSW is to define the abstract and concrete syntax of the language. Both are defined in the grammar, written with an Extended Backus-Naur Form (EBNF)[3] style notation [44] called grammar language (see Section 4.3.3). The whole workflow of the Xtext framework departs from the grammar definition, and is outlined in Figure 4.2.

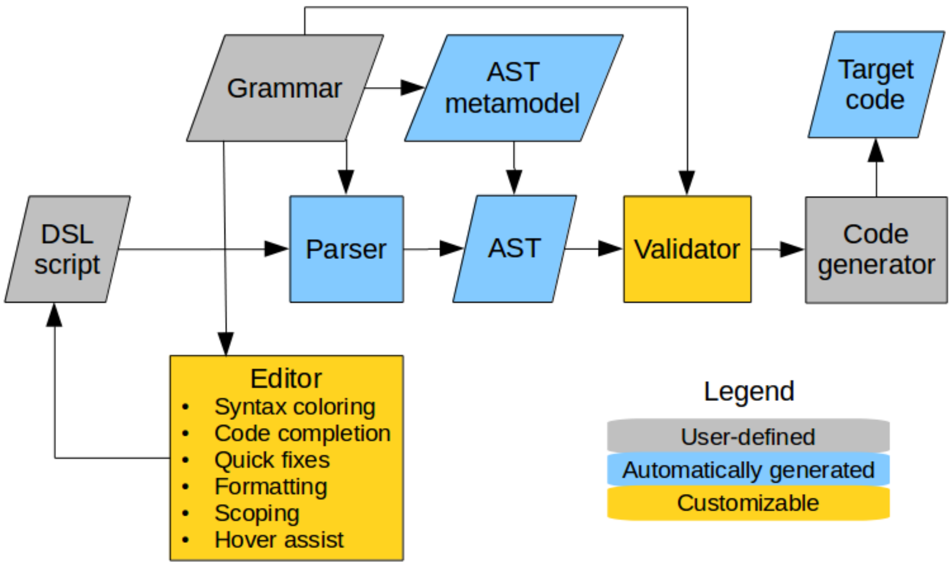


Figure 4.2: Xtext workflow

From the DSL grammar, Xtext derives an AST metamodel, a parser, a very simple default validator, and an Eclipse-based editor with basic support [27, 43]. The AST metamodel is saved as an Ecore model, while the parser is created using the ANTLR parser generator. Subsequently, EMF generates Java interfaces and implementation classes for each EClass in the metamodel, corresponding to language concepts. The implementation class contains getters and setters for each EAttribute, usually corresponding to assigned features within a grammar rule. Since these artifacts are generated, they are saved in the corresponding package in a folder called *src-gen*. [43]. Later, when a DSL script is parsed, the AST is instantiated as a tree of EObjects, *i.e.*, instances of the EClasses in the metamodel [11].

Furthermore, Xtext creates a scaffolding to assist the user in the development of the DSL. Default implementations for the validator and all editor support features are provided. All of these default implementations are customizable, and for the most common of them, customization class stubs are automatically added to the *src* folder. Since the code generator can be used to output any kind of target code, a default implementation cannot be provided. Nevertheless, an implementation class stub is also created.

[2]Xtext, as most language workbenches, can also theoretically be used to develop GPLs. However, doing so would be highly unpractical, so this possibility will not be taken into account here.
[3]EBNF is a type of metasyntax notation for defining grammars.

Xtext provides yet other services to assist the developer, from which the class `MyDslGrammarAccess` must be highlighted. It allows one to programmatically access and use the elements of the grammar.

Xtext follows the GGP and uses a DSL called Modeling Engine Workflow 2 (MWE2) to configure the generation of the above-mentioned components and artifacts [24].

Finally, it is important to mention Xbase. It is a reusable expression language developed with Xtext. It was designed to provide a base for including behavior modeling into new DSLs. However, it is tightly coupled with Java. If the goal is to generate target code in some other language, as is the case here, Xbase cannot be used [43].

### 4.3.1 Customization

The automatically generated default implementations shown in Figure 4.2 can be extended and overridden by the user. Customization is done using the same mechanism that is used to assemble all Xtext components [45]. It is based on the dependency injection pattern, in which the dependencies (or services) are passed to the dependent object (or client). This means that, when developing a component, one only has to declare the dependencies, without worrying about resolving them. The resolution is then handled by the framework, in this case Google Guice. For instance, when creating a component that depends on a scope provider (for cross-reference handling), a developer could inject it into his class with

```
@Inject
private IScopeProvider scopeProvider;
```

A so-called *module* is then used to map types to the implementations that should be injected by means of API methods, like

```
public class MyDslRuntimeModule extends  AbstractMyDslRuntimeModule {
   public Class<? extends IScopeProvider> bindIScopeProvider() {
      return MyConcreteScopeProvider.class;
   }
}
```

This *bind* method, for instance, makes Guice inject a new instance of `MyConcreteScopeProvider` upon declaration of a dependence to the interface `IScopeProvider`. By subclassing a default implementation and binding it, the user can customize nearly everything in Xtext.

To code the custom implementation classes, Xtext encourages the use of Xtend. It is a statically typed language which fully integrates with, and translates to, Java. Xtend eliminates some of Java's syntactic clutter and extends it with some handy features. The most remarkable of those features is probably its template expressions, ideal for templated code generation [44].

To extend the validator, the user has to write validation methods annotated with `@Check` within the validator stub class. The validator will then pass all instances with a compatible runtime type to those methods [43]. An example of how to configure the code generator is provided (commented out) in the generator stub class, using the `doGenerate` method. When the code generator is triggered, this method is called for each model file (also called resource) in the project [11].

### 4.3.2 Testing

For every feature implemented/extended in an Xtext project, as in the development of any other software application, it is of the uttermost importance to test it in order to verify that it works as expected. Automated tests are necessary to continuously verify that the code is behaving as it should, and that newly added features do not interfere with the workings of existent ones [43]. If a problem arises, the failed test(s) will indicate it right away, while also hinting the developer on where the error resides. Note that the effectiveness of testing greatly depends on which scenarios are covered by the test cases, so tests do not guarantee that the code is bug free.

For the purpose of implementing automated tests, Xtext uses JUnit. It is a widely-used framework for unit testing and it is shipped with the Eclipse Java Development Tools (JDT). One can easily create classes in the tests package containing methods annotated with `@Test`. Assert methods (*e.g* `assertEquals` and `assertTrue`) can then be used to check the correct functioning of the developed software. JUnit then provides a helpful report about failed tests. To assist the testing of the grammar, the validator and the code generator, Xtext automatically generates helper classes [43].

### 4.3.3 Grammar and parsing

This section, based on the Xtext user guide [45], presents some of the basic features of the grammar language and their importance in the parsing process. The goal is to provide the concepts to understanding the definition of the grammar of the AOCS DSL. Other concepts will be introduced in Chapter 6 as they appear in its implementation.

#### Preamble

Every Xtext grammar contains a short preamble before the syntax definition, consisting of a language declaration, EPackage declarations, and optional Epackage imports.

The first line in every grammar is the language declaration, and it takes the following form:

```
grammar my.package.MyDsl with org.eclipse.xtext.common.Terminals
```

This example declares a language called MyDsl. It makes use of Java's classpath mechanism to indicate that the grammar is contained in the package *my.package*. The 'with' keyword is used to specify that this language inherits all constructs and features of the Terminals language (*org.eclipse.xtext.common* package). All Xtext grammars inherit from this language by default. Its definition can be seen in Appendix A.1.

The most common, and easiest approach, is to let Xtext infer a metamodel (EPackage), from the grammar. To do this, it is necessary to have an EPackage declaration like

```
generate mydsl "http://www.eclipse.org/my/package/MyDsl"
```

It tells Xtext to generate an EPackage named *mydsl*, with the namespace URI)[4] (nsURI) "http://www .eclipse.org/my/package/MyDsl". The name and nsURI are the only properties needed to create an EPackage.

[4]A Uniform Resource Identifier (URI) is string of characters used to identify a resource.

Existent EPackages can be imported using their Uniform Resource Identifier (URI). A commonly imported package is Ecore. By declaring it as

```
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

the Ecore EPackage is imported with the alias *ecore*. Types defined in this package can then be used in the grammar through a qualified reference like `ecore::EString`. The applications of this are shown in the next sections.

### Terminal rules

The first step of the parsing process is called lexing. The lexer runs through the input character sequence (the DSL script) and transforms it into a sequence of atomic symbols called tokens. A token is a sequence of characters. It can either be a keyword or a match to a terminal rule (or lexer rule). The lexer will always look to match a longer token. In the case that a character sequence matches two or more different terminal rules, it will be attributed to the one defined first. Xtext provides an error message when a terminal rule is hidden by a previous one, *i.e.*, when it cannot possibly be matched. As an example, the terminal rule `INT`, included in the Terminals grammar, is defined as

```
terminal INT returns ecore::EInt:
    ('0'..'9')+;
```

A character range is declared with two dots (..), so `‘0’..‘9’` matches any character between ’0’ and ‘9’[5]. The expected number of occurrences of the expression between parenthesis is defined by the cardinality operator:

- Exactly one (default);
- One or none (?);
- One or more (+);
- Any number (*).

Therefore, the `INT` terminal rule is matched by a sequence of one or more integers from 0 to 9. Although in this case the quotes surround a single character, they can be used to declare other literals, like keywords or even multiple-word strings.

Terminal rules return a value with a certain type (EDatatype), which defaults to EString (equivalent to a Java String). If another type is to be returned, like EInt (equivalent to int), it must be explicitly declared. Furthermore, if the user wishes to have the returned value be anything other than the parsed character string itself, a value converter must be implemented, with two-way transformation methods between value and string.

### Production rules

After the lexing phase, the parser takes the stream of tokens as input. Using production rules (or parser rules), it attributes meaning to the input and constructs a tree of EObjects. The type of each node in the

[5]The order of characters is given by their ASCII code.

tree is inferred by the parser. To demonstrate how production rules are defined and how they influence the parsing phase, lets walk through the Entities grammar (*org.example.entities.Entities*) [43].

The entry point for building the model is the `Model` rule, which merely states that an Entities model consists of a list of entities. Then, the `Entity` rule is defined as

```
Entity:
   'entity' name=ID ('extends' superType=[Entity])? '{'
      attributes+=Attribute*
   '}'
;
```

The parser will expect the 'entity' keyword followed by an identifier (default terminal rule `ID`), which is assigned to the feature `name`. The type of an assigned feature is derived from the return type of the right-hand side of the assignment. In this case, `ID` returns EString, so `name` will be a string. Next, an optional super-type can be declared to specify that the entity inherits from another entity. The square brackets denote a cross-reference to another object of type `Entity`, which will be resolved later, in the linking stage. Finally, there is an assigned call to the rule `Attribute`. Therefore, the parser will process the following token sequence according to that rule. Furthermore, the assignment operator `+=` tells it to expect multiple instances, which makes `attributes` a list feature. The type of the elements of that list will be derived from the return type of the called rule. It defaults to a class named after the rule, but it can be specified to be any other arbitrary class, which would be interpreted as being a superclass.

Continuing down the hierarchy of rules, the `Attribute` rule simply declares that an attribute has a type and a name. More interesting though, is the rule `AttributeType`, since it uses another assignment operator.

```
AttributeType:
   elementType=ElementType (array?='[' (length=INT)? ']')?;
```

This last assignment operator `?=` denotes a boolean assignment. The consequence is that `array` is a boolean feature, which will be assigned the value *true* if the keyword '[' is found.

In the rule `ElementType` there are no assignments, only unassigned rule calls.

```
ElementType:
   BasicType | EntityType;
```

Since the parser creates the EObjects lazily upon the first assignment, this means that `ElementType` will not be instantiated. Instead, it will become an abstract class with two subclasses. The vertical bar (pipe) separates alternatives.

At last, an `EntityType` contains only a cross-reference to an `Entity`, while a `BasicType` lets the user choose the type of the attribute in question from a list of alternatives:

```
BasicType:
   typeName=('string'|'int'|'boolean');

EntityType:
   entity=[Entity];
```

The last stage of the process will be done by the linker, which will resolve the cross-references in the tree model. The scope provider lists the objects with a matching type which are referable, *i.e.*, within the scope of the reference. The linker then tries to match the input token to one of those objects. The most common scenario is to have an `ID` as input and matching it to the `name` feature of the object.

### Data type rules

Like terminal rules, data type rules also create instances of EDataType and do not call production rules or contain any assignments. The difference resides in the fact that they are used by the parser instead of the lexer. The consequence is that these rules are context-sensitive, so they will not conflict with, or hide, terminal rules.

As an example, lets write a rule to define a signed integer:

```
SignedInt returns ecore::EInt:
   ('-'|'+')? INT;
```

In this case, since the sign is optional, an unsigned integer could be interpreted as an `INT` or as a `SignedInt`. If the latter is defined as a terminal rule, there would be a conflict. However, the parser knows from the production rules to expect one or the other, so the definition is valid. For this reason, it is always a good practice to have a minimal amount of terminal rules.

As was the case with terminals, a data type rule must have an associated value converter [43].

# Chapter 5

# The AOCS DSL

This chapter describes the proposed solution for reducing the development time and increasing the reusability of the AOCS software. As already mentioned, it is a DSW developed with the Xtext language workbench.

In the first place, the context of this thesis work is presented in Section 5.1. Next, the scope for the implementation of the AOCS DSL is defined in Section 5.2. Then, Section 5.3 conducts an analysis on the existing AOCS software which provides the basis for this work. Lastly, Section 5.4 presents the design for the language modules, the semantic model and the code generator.

The whole system analysis and design presented throughout this chapter provides a basis for the implementation of the proof of concept, which is the topic of Chapter 6.

## 5.1 Context

The AOCS DSL has been developed at the Department of Simulation and Software Technology (SC) BRaunschweig of the German Aerospace Center (DLR). This work is influenced by a few of the projects, concluded and ongoing, of the department.

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 5.1

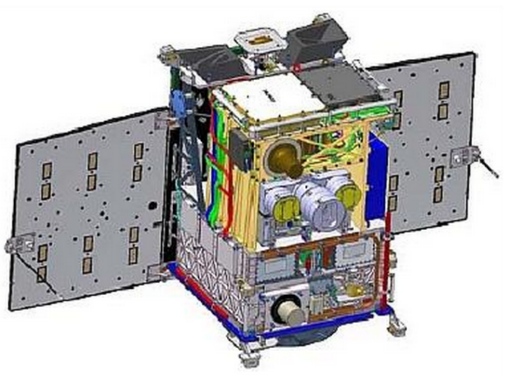


Figure 5.1: TET-1 satellite with deployed solar panels and stripped of its insulation (from [46]).

The Technology Experiment Carrier (TET)-1 is a technology demonstration satellite and it was launched on July 2012 [47], under the German national On-Orbit Verification (OOV) program [48]. The goal of this program is to provide the means for the industry and research institutes to test space technologies in their intended space environment [48], thus bridging the gap between ground testing and intended application. The series of satellites for this program use a generic satellite bus called Technology Experiment Carrier (TET), a modular multi-mission spacecraft bus based on the also DLR-lead BIRD[1] satellite [49]. This satellite was launched in 2001 [50], intended for detection and observation of hot-spots on Earth and also as a technology demonstrator [51].

The coding of repetitive and low-level tasks in the development of AOCS software for the above-mentioned 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 Language for Metadata Based Applications (LAMBDA) [44], and its product is a new DSW to model metadata for knowledge management tools. The framework is intended to be used by the developers of these tools as a mean of ensuring a consistent data model over the set of components. Although not related to AOCS software, the activities in the development of DSLs provide a much helpful insight and base experience for this thesis. The LAMBDA DSL was developed in the Xtext language workbench, which is why it is chosen for this thesis work.

## 5.2 Scope

The AOCS software is a big and complex system, even for small satellites like the BIRD or the TET-1. Therefore, it would be unrealistic to aim for the full implementation of the framework during a master's thesis project. Instead, one has to define a starting point: 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 of the AOCS around the implemented feature. The goal is to show in detail how the support for the development of that specific feature can be implemented, from the bottom level (functionality) up to the top-level constructs of the AOCS software. 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. In order to ensure that a good insight is gained on the handlers, several instances were taken and compared, to locate the common and specific features. Then, the elements on which the command handlers depend are analyzed, and then the elements on which these depend, and so on. The objective is to end up generating a partial, but compilable, collection of target files. The intention is obviously to extend the DSL to possibilitate the modeling of other functionalities. However, the code generated by the proof of concept can theoretically be aug-

[1] Bi-Spectral Infrared Detection.

mented manually, thus representing already an improvement on the current development methodology. In this process, empty stubs are to be used, when needed, to replace code that falls beyond the defined scope.

## 5.3 Analysis

The first step of this thesis work is to analyze the AOCS: its structure, functionalities and behavior. The intent is to study the feasibility of a DSL-based solution, *i.e.*, assess which parts of the software can be automated and how, in order to evaluate what can be achieved. Another goal is to determine the requirements for the language and the code generator, such that a similar system can be instantiated. The theoretical part of this analysis consists in the literature review conducted in Section 2.1. But to understand the software of the system in detail, it is necessary to study the source code of an actual implementation. This more practical analysis is conducted on the source code of the AOCS of the TET-1 satellite, introduced in Section 5.1. This system results of an adaptation and extension of BIRD's attitude control software at DLR's SC[48]. The end result of this section is the specification of the target classes and files to be replicated.

### 5.3.1 Command handlers

When telecommands are received from the ground, they are processed and dispatched to the appropriate command handler, which is put in charge of executing it. Note that terms *command* and *telecommand* are used interchangeably. There is no risk of ambiguities because all commands mentioned in this context are originally sent from the ground.

#### Base class

A base class `CommandHandler` exists which works as a superclass for each specific handler. Although technically it is a default implementation, it acts mostly as an interface[2]. In other words, it is possible to instantiate this base class, but its main purpose is to guide the implementation of specific handlers. The class contains two methods: a virtual destructor and the `handle` method.

- The virtual destructor allows for the destruction of an instance of a command handler using a pointer to the base class. It is a common procedure in C++ programming, to avoid memory leaks by inadequate resource deallocation [52].
- The `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*). Since all commands are processed equally until they reach the handler, the reference is received as a *void* pointer, *i.e.*, a pointer to objects of any data type. It returns the number code of an error message (of type `ErrorMessage::ID`), which is meant to provide

[2]The C++ equivalent of a Java interface is an abstract class. Such a class contains so-called pure virtual methods that must be implemented in its subclasses.

feedback about the success of the command execution to the caller. This default implementation of the `handle` method returns the error message `unimplementedHandler`.

This base class does not need any input from the developer of the AOCS software, so it can be generated with a static template.

In the analyzed source code, some types of commands inherit from a more specific base class, which in turn inherits from `CommandHandler`. However, this specific case will not be covered here. If needed, one can easily modify the code generator to create such intermediate classes.

### Specific classes

In the TET-1 software, the specific command handler classes are named `CommandHandler<MyCommand>`. They override the default implementation in the base class according to the telecommand that they need to execute.

The only method in a command handler is `handle`, which overrides the correspondent virtual method of the base class to execute the command. By comparing the handlers for different commands, it is possible to identify its general workflow:

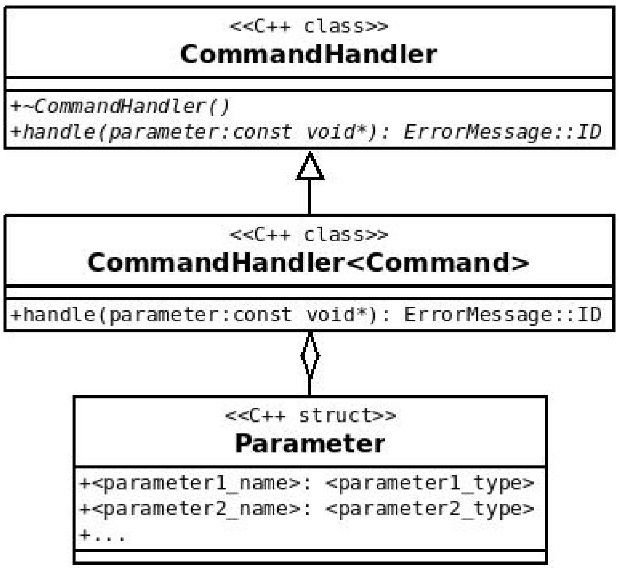
1. An `ErrorMessage::ID`-type variable is initialized to `noError`, to be changed during execution of the command if something unexpected happens. This initialization should be done automatically.
2. The received *void* pointer is casted to a `Parameter` pointer, so that its members can be accessed. The `Parameter` structure is defined privately in each handler. This behavior can easily be generated with no user input.
3. The command is executed, resorting to a combination of low-level operations and calls to methods of AOCS components (see Section 5.3.2). Low-level operations could be easily integrated in the DSL if an expression language like Xbase existed for C++. Since it does not, it is probably easier to have this task performed manually at target code level. Components must be requested from a class called `ComponentManager`. Since the calls to components can be intertwined with other operations, they must be manually coded as well. What can be done to facilitate the user's task is to provide an implementation stub with instructions on how to get the desired components.
4. An `ErrorMessage::ID` is returned indicating the result of the command execution. Therefore, it must also be manually implemented.

Figure 5.2 outlines these classes. It also shows the inheritance relation to the previously introduced base class and the aggregation relation to the defined `Parameter` structure.

The specificity of these classes requires that a parameterized template is used to generate them. The DSL must then provide the abstractions to define the parameters of the command.

### Managing commands

In order to generate the target files revolving around the command handlers as completely as possible, it is necessary to look further into the system to understand how these handlers are used and how they



<<C++ class>>
CommandHandler
+~CommandHandler()
+handle(parameter:const void*): ErrorMessage::ID
<<C++ class>>
CommandHandler<Command>
+handle(parameter:const void*): ErrorMessage::ID
<<C++ struct>>
Parameter
+<parameter1_name>: <parameter1_type>
+<parameter2_name>: <parameter2_type>
+...

Figure 5.2: Command handler class diagram.

fit into its structure.

The AOCS software is organized hierarchically. Commands are grouped into components, and components are grouped into applications, the top-level constructs of the system. Command handler instances are created and held by a class at the component level. The same class then registers the handler instances at the telecommand interpreter specific to the application that owns the component, *i.e.*, at the application level.

The registration of command handlers involves the use of the identification number. Similarly to error messages, each command has an ID defined in an enumeration (a C++ *enum*). The difference is that command IDs are unique only in their application, and not in the whole AOCS. This ID is also used to document the handler class.

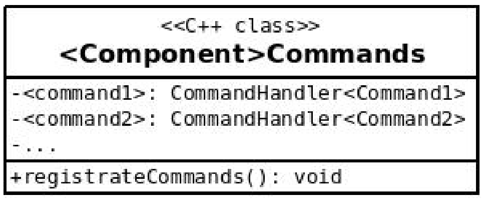
The following sections assess these elements, and others which are related, in more detail.

### 5.3.2 Components

Components are units of the system that implement certain functionalities. For example, the Estimation, Prediction and Control (EPC) 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. Although a command handler can call any component in the system during its execution, each command is hierarchically under a component. In the source code file tree, this means that command handler files are put into a folder corresponding to the component that owns them. Therefore, the AOCS DSL must provide the possibility to instantiate components to wrap a group of commands. For the sake of completeness, all the remaining classes at the component level should be generated, whether their implementation is automated or not.

### Command handler instances

For each component, there is a class `<MyComponent>Commands` which instantiates the command handlers which are grouped under it (see Figure 5.3). Other than having an instance of each handler as a private member, it only contains the public method `registrateCommands`. This method registers the commands at the telecommand interpreter of the application to which the component belongs (see Section 5.3.3), taking no arguments and returning no output. To automatically generate this class, only a list of commands is needed.



<<C++ class>>
<Component>Commands
-<command1>: CommandHandler<Command1>
-<command2>: CommandHandler<Command2>
-...
+registrateCommands(): void

Figure 5.3: Class to instantiate the component's commands

### Component classes

Identically to command handlers, AOCS components have a base class which provides an interface to specific component classes. However, the `Component` base class in the TET-1 software is simply an empty class, foreseeing the eventual need to define a component interface. This class can therefore be generated without any feature being added to the DSL. To replicate the pattern followed by the command handlers, a virtual destructor can be added.

Without going too off-course, a basic stub for the specific classes of the components declared in a model can still be generated. The relevant points about component classes are:

- The constructor of each component in the TET-1 AOCS software is responsible for registering it at the `ComponentManager`. This could easily be automated just taking the name of the component as input. However, since components are to be declared at DSL level and automatically generated, an additional class can be generated which holds all component instances and registers them, similarly to what is done for the command handlers.
- 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 to-be-generated component classes are just empty stubs, with the exception of a virtual destructor in the base class. Nevertheless, Figure 5.4 explicitly depicts these classes.

### Component instances

As already mentioned, the constructors of the component classes in the TET-1 software are responsible for registering the created objects at the component manager. But advantage can be taken of the fact that the C++ code is automatically generated to create an additional class.

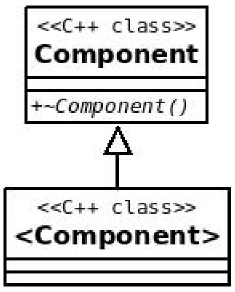
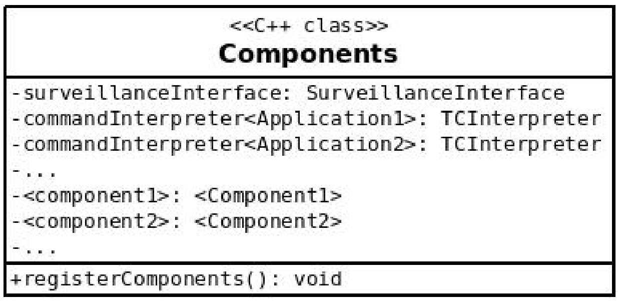


Figure 5.4: Component class diagram

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 surveillance component and a command interpreter for each defined application. Its only method, `registerComponents`, registers each and every one of these instances at the component manager. Figure 5.5 outlines this class.



<<C++ class>>
Components
-surveillanceInterface: SurveillanceInterface
-commandInterpreter<Application1>: TCInterpreter
-commandInterpreter<Application2>: TCInterpreter
-...
-<component1>: <Component1>
-<component2>: <Component2>
-...
+registerComponents(): void

Figure 5.5: Class to instantiate components

#### Component IDs

By now, it is possible to recognize the pattern of having AOCS elements of the same type associated with an identification number (ID) in an *enum*. Components are no different, and again the contents of the target file containing those IDs can be easily inferred.

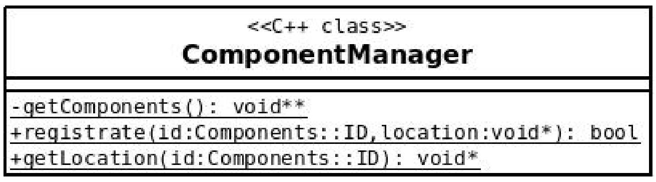
#### Component manager

As already mentioned, components are registered at the `ComponentManager` upon instantiation. This class consists of three static methods:

- The private method `getComponents` is used internally to fetch a table of pointers to all components. This table is defined as a static variable. This means that its “life time” spans the whole execution of the program. On the first call, `getComponents` sets all pointers in the table to null pointers (*0* or *NULL*).
- The `registrate` method receives a component’s ID and a pointer to the location of its instance (again a *void* pointer). The component is then registered by including the pointer in the table. The ID serves to indicate the position at which it should be inserted. It then returns a boolean value to assert that the ID received is valid.

- The last method `getLocation` is used by any entity that wishes to fetch a certain component. A pointer is retrieved from the table by calling `getComponents` and then returned to the caller.

Figure 5.6 summarizes the above. It can be concluded that this class will present the same implementation for any AOCS system, so it can be generated from a static template.



<<C++ class>>
ComponentManager
-getComponents(): void**
+registrate(id:Components::ID,location:void*): bool
+getLocation(id:Components::ID): void*

Figure 5.6: Component manager class

### 5.3.3 Applications

Applications are the software modules of the AOCS. Each application is unambiguously identified within the whole spacecraft system by an Application Process Identifier (APID). It is also used, together with the telecommand code, to document the command handlers in their definition source file. The application's APID can easily be inferred from the order of the application definitions in a model.

As stated in Section 5.3.1, an enumeration of telecommand IDs exists for each application. It includes all the telecommands owned by the components of that application. Command IDs are unique for each application, but mission control stations in the ground actually identify all the commands of the spacecraft system with a single number. For that reason, the enumeration of the commands of an application is accompanied by a constant integer called `OFFSET`. This allows the translation of a global identification code to an application-specific one.

#### Telecommand interpreter

Externally to the AOCS, 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 `TCInterpreter`.

They contain three member variables:

- The variable `commandHandlerTable` is a table of pointers to the command handlers.
- The variable `undefinedHandler` is used to initialize the pointer table.
- The `lastTcCode` is an unsigned integer variable that holds the identification code of the last received telecommand.

Additionally, the `TCInterpreter` template defines and implements the following methods:

- Its constructor initializes the handler pointer table as a table of undefined handlers. Then, it registers the interpreter at the command manager. Again, this task will be relegated to the class that instantiates all components.

- The method `registrateCommand` registers the received handler by adding it to the handler table.

- The method `executeCommand` starts by reporting the received telecommand to the surveillance component (see Section 5.3.4). It then passes the parameters to the appropriate handlller for execution. In the case that some error occurs during execution, the failure is reported to surveillance and the method returns the boolean value *false*.

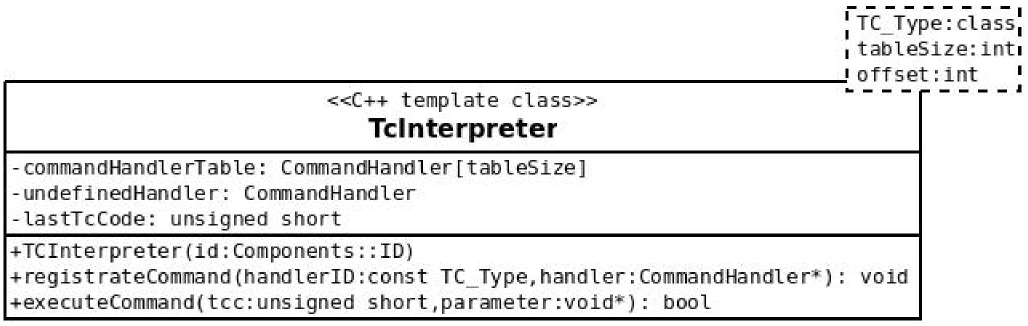

TC_Type:class
tableSize:int
offset:int
<<C++ template class>>
TcInterpreter
-commandHandlerTable: CommandHandler[tableSize]
-undefinedHandler: CommandHandler
-lastTcCode: unsigned short
+TCInterpreter(id:Components::ID)
+registrateCommand(handlerID:const TC_Type,handler:CommandHandler*): void
+executeCommand(tcc:unsigned short,parameter:void*): bool

Figure 5.7: Template class for telecommand interpreters

Figure 5.7 shows a schematic of the described template class. This class can be generated from a static template. Furthermore, if the AOCS DSL allows the user to define applications, it is trivial to derive a component ID and instance name for each application.

### 5.3.4 Surveillance

As seen in Section 5.3.3, an application's command interpreter uses a surveillance component called `SurveillanceInterface` to report events and failures. Some of the command handlers analyzed happened to call methods of this component as well. Also, by looking into the file tree of the TET-1 source code, it can be observed that both the event and error message IDs reside in the directory of the surveillance component. Therefore, this section addresses the `SurveillanceInterface` class, along with the declaration of the identification codes.

#### Events and error messages

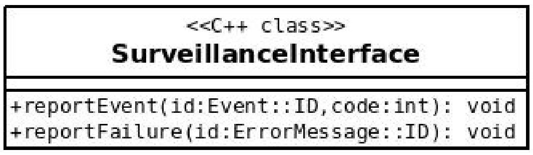
The enumerations for the events and the error messages need to follow the same pattern: *enum* members for default events and error messages must be generated in a way that the enumeration can be extended with new members. How exactly to do this is a problem that will be addressed in Section 5.4.3. The default members that need to be defined in order to assure that the generated code is compilable are:

**Events:** `REC_CMD` (received command).

**Error messages:** `noError`, `commandParameterRange` and `unimplementedHandler`.

**Surveillance interface**

Since the analysis of the `SurveillanceInterface` class would fall beyond the scope of this work, defined in Section 5.2, generating a class with empty stubs for these two methods suffices. Once again, if functionality needs to be added, it can always be manually implemented by the user. Figure 5.8 outlines the to-be-generated surveillance class.



<<C++ class>>
SurveillanceInterface
+reportEvent(id:Event::ID,code:int): void
+reportFailure(id:ErrorMessage::ID): void

Figure 5.8: Surveillance component class

## 5.4 Design

The knowledge about the AOCS software obtained in Section 5.3 is used in this section to outline the design of the AOCS DSW.

First, the composition of the language, *i.e.*, its language modules, is presented in Section 5.4.1. Then Section 5.4.2 describes the semantic model of the DSL based on the analysis conducted. Finally, Section 5.4.3 presents a plan for using the GGP for code generation.

### 5.4.1 Common language module

One of the goals of the proof of concept is to demonstrate how to extend the methodology to other AOCS subsystems. With that in mind, a reusable language model is to be created containing features common to all subsystems. The idea is that any additional DSL developed for another spacecraft subsystem can be built on top of this module.

This language, named Common, extends the Terminals grammar with new terminal rules and some basic data type rules. These rules are naturally accompanied by their respective value converters.

The implementation of this language is presented in Section 6.1

### 5.4.2 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. The telecommand parameter is therefore the most complex concept in the language.

In the analyzed source code, the definition of a parameter contains only its name and type, while the range (or set) of valid values is coded directly into the validity check performed in the `handle` method. Therefore, a feature to specify the valid values of a parameter would be a valuable addition, allowing to automate the validity check.

Additionally, and taking inspiration from mbeddr (Section 3.3.3), a feature to specify the physical quantity represented by a parameter and a unit of measurement in which it is expressed can be added to the parameters. Although this is not currently used in the AOCS software, it can be used to document the target code. Furthermore, it opens the possibility to automate model validation and formal verification.

Figure 5.9 shows the semantic model design as described above. It can already be seen that a system model can be greatly simplified by using domain-specific abstractions. Note that this does not represent the actual semantic model of the AOCS DSL, as implementation details are not taken into account here.

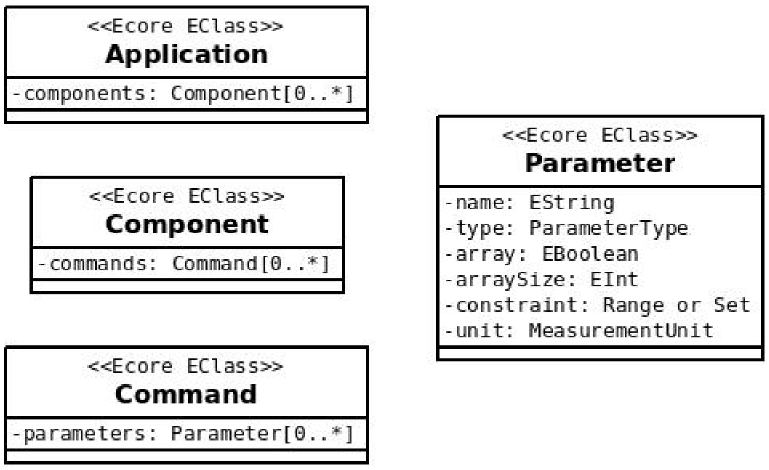


Figure 5.9: Semantic model design for the AOCS DSL

Note that while the `Parameter` defined in the source code corresponds to a structure of parameters, this `Parameter` class represents each one of those parameters. This concept, to be realized by a production rule in the grammar, is consequently translated by Xtext into an Ecore EClass (and later by EMF into a Java class). Because of this, UML class diagrams depicting DSL concepts exhibit Ecore types.

### 5.4.3 Generation gap pattern

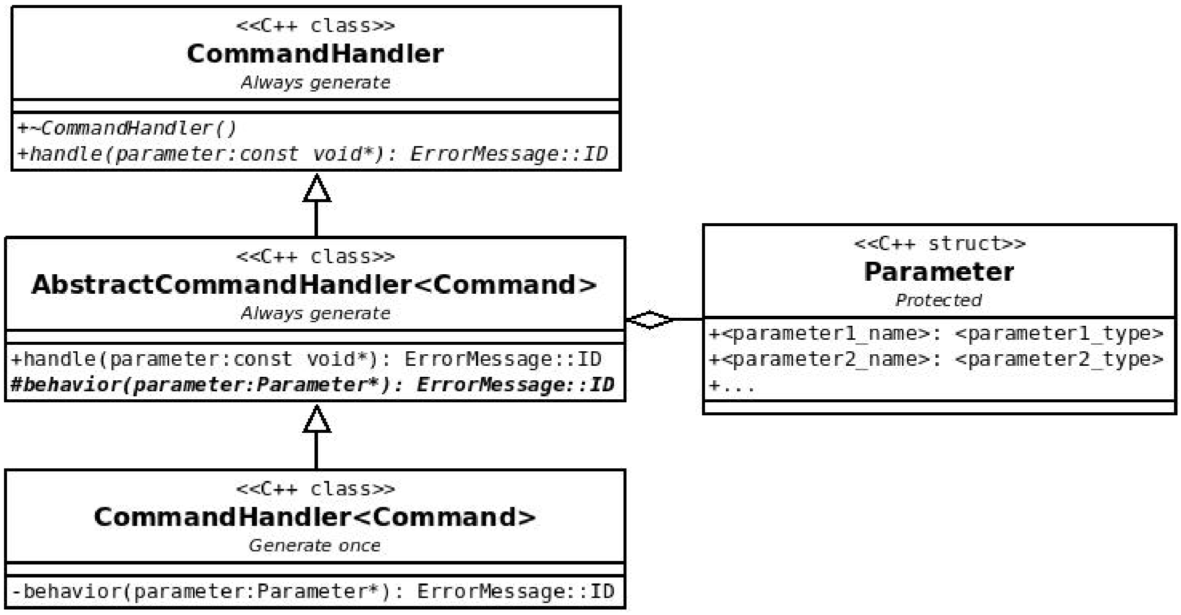
It was seen in Section 5.3 that most of the target code can be automatically generated from the DSL model. But for some of the software components, generated and manually written code must be integrated. There are three such cases: the command handlers, the component classes and error message and event code enumerations.

The framework is to follow the GGP to separate automatically generated 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 (*e.g.* class stubs) in a different folder that allows the user to modify them (named *src*).

The error message and event codes contain only the defaults, so they must be manually customizable. Since these are defined in *enums*, and not classes, inheritance cannot be used to provide the desired extensibility. The only solution is therefore to follow a *generate once* policy (see Section 2.2.3). The files are then created in the *src* folder, where the user can edit them at will. There is a risk that the default definitions are inadvertently deleted, but in that case an error message will be issued by the compiler.

The component classes generated are just empty stubs, so any customization has to be done manually. Consequently, the correspondent files can be simply put in the *src* folder as well.

The command handlers require a more complex solution. It is necessary to automate the parameter definition and range checking, while leaving the command execution code to be manually written. The solution to this is depicted in Figure 5.10.



<<C++ class>>
CommandHandler
Always generate
+~CommandHandler()
+handle(parameter:const void*): ErrorMessage::ID
<<C++ class>>
AbstractCommandHandler<Command>
Always generate
+handle(parameter:const void*): ErrorMessage::ID
#behavior(parameter:Parameter*): ErrorMessage::ID
<<C++ struct>>
Parameter
Protected
+<parameter1_name>: <parameter1_type>
+<parameter2_name>: <parameter2_type>
+...
<<C++ class>>
CommandHandler<Command>
Generate once
-behavior(parameter:Parameter*): ErrorMessage::ID

Figure 5.10: GGP in the command handlers

An intermediate abstract class is introduced which implements the `handle` method. After casting the pointer to the parameter structure and performing the validity check, it calls the `behavior` method, which is defined as a pure virtual method in the abstract class. This forces the user to provide the implementation in the concrete subclass, which is created in the *src* folder.

# Chapter 6

# Implementation

This chapter presents the implementation phase of the whole proof of concept. It begins by describing, in Section 6.1, the developed Common language module. The remaining sections cover the AOCS DSW.

The grammar which defines the concrete and abstract syntax of the language is the topic of Section 6.2. Then the implementation of the model validator and the code generator is explained in Sections 6.3 and 6.4, respectively. Finally, Section 6.5 presents some additional features, mostly related to the UI.

## 6.1 Common language module

A language module named Common was developed with the intent of extending the default Terminals grammar. It includes new terminals and basic data types which are not specific to the AOCS language. The following sections show the definition of these rules and related features. The complete Common grammar can be found in Appendix A.2.

### 6.1.1 Number literals

Since the Terminals grammar only defines a rule for lexing unsigned integers (`INT`), one of the issues addressed is the representation of numbers. Rules have been created to define hexadecimal integers, signed decimal integers and real numbers. These rules are shown in Listing 6.1.

```
Real returns ecore::EDouble: SignedInt '.' INT (('E'|'e') ('+'|'-')? INT)?;

SignedInt returns ecore::EInt: ('+'|'-')? INT;

terminal HEX returns ecore::EInt: '0' ('x'|'X') ('0'..'9'|'a'..'f'|'A'..'F')+;
```

Listing 6.1: Common rules for parsing numbers

Note that the fractional part of real numbers is defined to be mandatory. This is due to the possibility that `Real` and `SignedInt` being used as alternatives for an abstract rule. In that case, a `Real` rule with optional fractional part would shadow the `SignedInt` rule.

A very simple value converter was developed for the `Real` rule. The Java standard method `Double.valueOf(String s)` (from the *java.lang* package) is called to parse the input string into a real value. For the rules `SignedInt` and `HEX`, a single integer converter class is used. By having its constructor take the base (or radix) in which the number is expressed, this class can be instantiated to convert both hexadecimal and decimal integers. It calls the method `Integer.valueOf(String s, int radix)` (*java.lang* package) to perform the conversion to value.

The proposal provider in the Common UI plug-in was also customized to provide basic completion proposals for each of these rules.

### 6.1.2 Documentation comments

A new terminal rule, inspired in Java's documentation comments, was created to allow the user to document the model. These multi-line comments, delimited with `/**` and `*/`, are parsed and can be included in the target code. The rule is called `DOC_COMMENT`, and it can be seen in Listing 6.2. The operator `->` denotes 'until'.

```
terminal DOC_COMMENT: '/**' -> '*/';
terminal ML_COMMENT: '/*' (!'*') -> '*/';
```

Listing 6.2: Definition of documentation comments

To prevent conflicts with the rule `ML_COMMENT`, which defines multi-line comments in the Terminals grammar, that rule must be overridden.

A value converter was implemented for `DOC_COMMENT`, which strips the comment string of its delimiters when converting to value and puts them back when converting back to string. Additionally, to help the user distinguish these and other comments, a custom color is defined by extending the `DefaultHighlightingConfiguration` class. This configuration was then connected to the `DOC_COMMENT` rule by overriding the `calculateId` method of the `DefaultAntlrTokenToAttributeIdMapper` class.

### 6.1.3 Qualified identifiers

An additional data type rule is defined to handle Java-like qualified identifiers. This can be useful for navigating namespaces/packages/scopes or file trees. No value converter has been developed for this particular type, since it is simpler to have an identifier as a single string and split the qualifiers whenever necessary.

## 6.2 Grammar definition

A grammar definition is used by Xtext to automatically generate the parser and infer both the concrete and abstract syntax of the language. With that in mind, a grammar is constructed based on:

- The reference semantic model presented in Section 5.4.2, 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 [43]. 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.

This section starts by discussing the rule that defines the AOCS telecommand parameters, along with the lower-level rules that it requires. Then those concepts higher in the hierarchy of the model are described. The complete AOCS grammar can be found in Appendix A.3.

### 6.2.1 Parameters

The attributes to be included in the `Parameter` class were shown in Figure 5.9, where it is evident that this is the most elaborate concept in the DSL. Listing 6.3 shows how the rule is actually implemented. The best way to understand it is to walk through the rule and discuss its elements one by one.

```
Parameter:
   comment=DOC_COMMENT?
   'parameter' name=ID 'is'
   type=Type (array?='array' '(' arraySize=INT ')')?
   (
      (constrained?='in' (
         rangeConstrained?='range' (range=AnonymousRange | rangeRef=[Range])
         | enumConstrained?='enum' (enumeration=AnonymousEnumeration | enumerationRef=[Enumeration])
      ))?
      & ('with' 'units' unit=MeasurementUnit)?
   )
;
```

Listing 6.3: Definition of a command parameter

Before the actual definition of the parameter in the model, the user has the option of writing a documentation comment which will be included in the model and therefore reachable to the code generator. Then, the user must write “`parameter myParameter is`”, followed by a mandatory declaration of the type of the parameter. If it is to be a multi-valued parameter (an array), then that must be specified with the ‘array’ keyword, immediately followed by its size in between brackets. The boolean assignment to the feature `array` is there merely to simplify the task of traversing the model.

The operator `&` separates the elements of an unordered group, so the parameter value constraint and measurement unit can be defined in any order. The assignment of a measurement unit is pretty straightforward. The constraint, however, is a little more complex. The parameter can be constrained to a range (`Range` rule) or to a set of values (`Enumeration` rule). The user can reference a range or a set defined elsewhere, as long as it is within the scope of the parameter. Alternatively, one can declare the constraint directly in the definition of the parameter, using the anonymous versions of the rules.

Some examples of valid parameter definitions are:

```
/**
 * 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
```

### Parameter constraints

As stated, ranges and enumerations can be defined separately and then referenced, or defined anonymously within a parameter definition. Listing 6.4 shows the grammar rules involved.

```
Range:
   'range' name=ID 'is' min=NumberLiteral 'to' max=NumberLiteral
;

Enumeration:
   'enum' name=ID 'is' '(' (enumerators+=Enumerator (',' enumerators+=Enumerator)*)? ')'
;

AnonymousRange returns Range:
   min=NumberLiteral 'to' max=NumberLiteral
;

AnonymousEnumeration returns Enumeration:
   {Enumeration}
   '(' (enumerators+=Enumerator (',' enumerators+=Enumerator)*)? ')'
;

Enumerator:
   name=ID (explicit?='=' value=SignedInt)?
;
```

Listing 6.4: Ranges and enumerations

First of all, it can be seen that the standalone versions of the rules have a `name` feature. This is required to be able to reference those definitions later. Note also that the anonymous version of each rule specifies a return type to be that of the original rule. This means that no `AnonymousRange` or `AnonymousEnumeration` EClasses will be created by Xtext. Instead, these rules merely present a different way to instantiate the same class: without specifying the name.

The limits of a range can be declared using a number literal, whose defining rule is shown in Listing 6.5. It is an abstract rule that allows the user to input either an integer or a real number.

```
NumberLiteral: IntegerLiteral | RealLiteral;

IntegerLiteral: value=SignedInt;

RealLiteral: value=Real;
```

Listing 6.5: Number literals

An enumeration is composed of a list of members, or enumerators, whose name is declared. The user can explicitly assign an integer value to each enumerator. Otherwise, a sequential numbering is assumed, starting with zero. Enumerations are syntactically allowed to be empty, so that a more elaborate error message can be provided by the validator.

As an example, the above-described types of parameter value constraints can be defined as

```
enum ENUM is (ZERO, TWO=2, THREE)
range RANGE is 0 to 5
```

### Types and measurement units

Parameter types and measurement units are defined in a similar fashion. The user specifies them using a keyword from a set of alternatives. The chosen value is the assigned to a feature called `code`. As an example, the `Type` rule definition is shown in Listing 6.6

```
Type:
    {BooleanType} code='bool'
    | {IntegerType} code=('uint8'|'int8'|'uint16'|'int16'|'uint32'|'int32')
    | {FloatType} code=('float'|'double')
;
```

Listing 6.6: Parameter types

Note that the type codes are grouped by categories, which are distinguished by so-called simple actions. By default, the object returned by a production rule is created lazily upon the first feature assignment. However, simple actions can be used to explicitly demand the creation of an EObject of a certain type, in which case the type of the object is specified between curly braces. Parameter types, for instance, are divided into `BooleanType`, `IntegerType` and `FloatType`. The type codes in the AOCS grammar directly translate to C++ data types.

### 6.2.2 Higher-level elements

The semantic model shown in Section 5.4.2 prescribes that each higher-level element of an AOCS model contains only a collection of the elements immediately below it in the hierarchy. However, one can take advantage of the cross-referencing mechanism of Xtext to give the user some extra freedom, as shown in Listing 6.7.

```
Application:
    comment=DOC_COMMENT?
    'application' name=ID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | commandDefs+=Command
            | componentDefs+=Component
            | 'component' componentRefs+=[Component|QualifiedID]
        )*
    'end' 'application'
;

Component:
    comment=DOC_COMMENT?
    'component' name=QualifiedID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | commandDefs+=Command
            | 'command' commandRefs+=[Command]
        )*
    'end' 'component'
;

Command:
    comment=DOC_COMMENT?
    'command' name=ID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | 'parameter' parameterRefs+=[Parameter]
        )*
    'end' 'command'
;
```

Listing 6.7: Hierarchy of AOCS concepts

By allowing elements to be defined in an outer scope of where they are used, 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. One important difference between these elements and a parameter is that, since their definitions are expected to span multiple lines, they have an end delimiter. The same pattern is applied to all levels of the hierarchy.

Note that the name of a component is given by a qualified identifier instead of a simple identifier. This is an exception, and is meant to be used solely for organizing the generated files. Within the AOCS software, components are treated only by their last name, which must therefore be unique in the whole system.

Listing 6.8 shows the first rule in the grammar, which constitutes the entry point for creating an AOCS model.

```
AocsModel:
    (applicationDefs+=Application
        | componentDefs+=Component
        | commandDefs+=Command
        | parameterDefs+=Parameter
        | enumerationDefs+=Enumeration
        | rangeDefs+=Range
    )*
;
```

Listing 6.8: The entry rule for the AOCS model

At this level, 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. This design pattern can be applied as follows:

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

component cpt is
    command cmd
end component

application app is
    component cpt
end application
```

## 6.3 Model validation

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 generated editor. It continuously checks the code for validity, providing marks and custom messages of different kinds: errors, warnings and information. This section shows how the validation mechanism was implemented for the AOCS DSL, describing the custom validity checks created for each class in the metamodel.

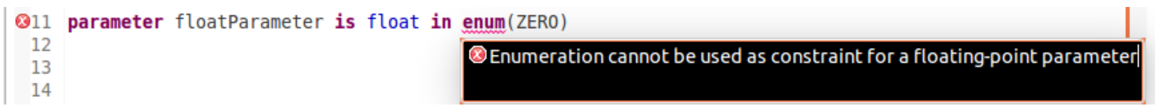
By default, Xtext generates only a class stub for the manual implementation of validation methods. However, there are validators already embedded in Xtext which can be activated in the file *Generate-MyDsl.mwe2*, where the workflow for the generation of Xtext artifacts is defined. One of those validators is programmed to check for the uniqueness of object names within each scope of the model. When it is activated, Xtext adds the `NamesAreUniqueValidator` (*org.eclipse.xtext.validation* package) to the abstract DSL validator class, using a composed check annotation (`@ComposedChecks`). This indicates that the methods of that class are to be used as validation methods.

### 6.3.1 Parameters

The `ParameterValidator` class contains methods which check the validity of the values assigned to each attribute of a parameter.

Since the size of an array parameter is given by the terminal rule `INT`, a check has been implemented which issues an error if a zero-sized array is declared, and a warning if the user declares a parameter to be an array of size one.

Other than that, a few check methods 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 6.1 shows how one of these error messages looks like in the AOCS editor. The error underline marker can be placed in any of the elements of the rule in question.



```
parameter floatParameter is float in enum(ZERO)
```

Enumeration cannot be used as constraint for a floating-point parameter

Figure 6.1: Error message for an enumeration-constrained *float* parameter

Since range limits are allowed to be either integer of real, their types are also checked against that of the parameter in question. A warning is issued when these types are inconsistent.

#### Parameter constraints

Specific validator classes have been implemented for `Range` and `Enumeration`. In a range definition, it is asserted that the minimum is in fact lower than the maximum, otherwise an error is issued. The types of each of these attributes are also checked for consistency.

As stated in the grammar description, enumerations are syntactically allowed to be empty. It is up to the `EnumerationValidator` to check for an erroneous input of this kind and mark it as an error. Another issue is the value of enumerators. The validation method that addresses this issue attributes default values to enumerators, when these are not explicitly declared. While doing so, the enumeration is checked for the existence of repeated values, which results in an error marker.

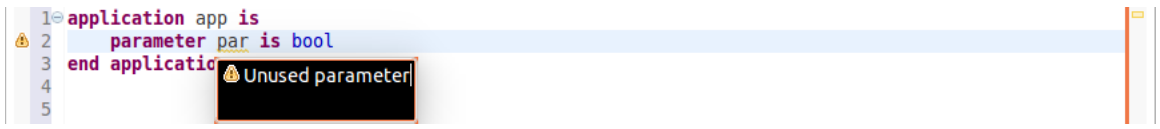
### 6.3.2 Higher-level elements

Also included as composed checks in the main validator class `AOCSValidator` are the specific validators for commands, components and applications.

The default `NamesAreUniqueValidator` will prevent the instantiation of objects with the same name within a scope. However, due to the referencing pattern described in Section 6.2.2, it will still allow, for example, for a command to be defined and referenced within the same component definition. Therefore, a validation method must be implemented at each level to correct this.

The fact that telecommands must be unique for each application, which is not their immediate container, demands for a specific check as well. Similarly, components must be checked for uniqueness at the model level, *i.e.*, in the `AOCSValidator`. Also in Section 6.2.2, it is stated that only the last name of the qualified identifier attributed to a component is used in the target software. This leads to the necessity for assuring the system-wide uniqueness of the last name of each component.

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 6.2 shows how the warning looks like for a defined but unused parameter. The parameter `par` is defined, but not referenced in a command within the `app` application, hence the warning.



```
application app is
    parameter par is bool
end applicatio
```

Unused parameter

Figure 6.2: Warning for an unused parameter

## 6.4 Code generation

By comparing the target files which need to be generated and the semantic model used (both addressed in Chapter 5), it becomes evident that there is a big gap between them in terms of abstraction and complexity. Without an intermediate M2M transformation, the code generator needs to single-handedly bridge this gap. However, note that the AOCS DSL was in fact developed iteratively, following a direct approach. This quest for simplicity ends up backfiring in a way, since the approach taken does not scale well as the complexity of the system increases. As a result, it was observed that the code generator evolved to be very complex. Nevertheless, the generator was logically split into several well organized files, which significantly mitigates the effects of its complexity.

This section describes the implementation of the code generator, including the output configuration, the types of helper classes and generator methods developed, the mechanisms for defining names for the target code, and the overall workflow of the generator.

### 6.4.1 Output configuration

The output configuration of the code generator was, as already mentioned, made to follow the GGP. This is achieved by creating a custom class to implement the `IOutputConfiguration` interface, named `AOCSOutputConfigurationProvider`. As with any customization in Xtext, it was then bound to the dependency injection mechanism. According to the design described in Section 5.4.3, two project-level output folders are created:

**src-gen** Purely generated files are written to this folder. All the resources in it are cleaned and rewritten 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 follow the *generate once* policy (see Section 2.2.3). They are not overwritten or deleted, and the user may edit them at will.

### 6.4.2 Helper classes

Two helper classes were created to ease specific tasks of the generator classes: the `FileInfo` class and the `MultiWordIdentifier` class.

The `FileInfo` class is intended to automate the compilation of the headers of the target files. Its attributes hold relevant data like the names of the source model file and the target file, a message instructing on whether to edit the file or not, a descriptive title, and any additional information. Its only non-constructor method, `compileHeader`, automatically compiles the file header, depending on the above-mentioned attributes. In this way, a `FileInfo` object can be passed to the *compile* methods (see Section 6.4.5), reducing the header compilation to a single line.

The second helper class contains a set of static methods to handle multi-word identifiers. It is an essential tool for transforming identifiers declared in the model into variable names, class names, *enum* IDs, *etc.* It contains two *split* methods: one to split a qualified identifier, and another one to split all words in an identifier. The word limits are identified by commonly used notations like white spaces, dots, capital letters (*CamelCase* notation) or underscores (*snake_case* notation). Additionally, it contains several methods to transform a multi-word string. This class is outlined in Figure 6.3 to show the variety of transformations it supports. Its methods are pretty much self-explanatory.

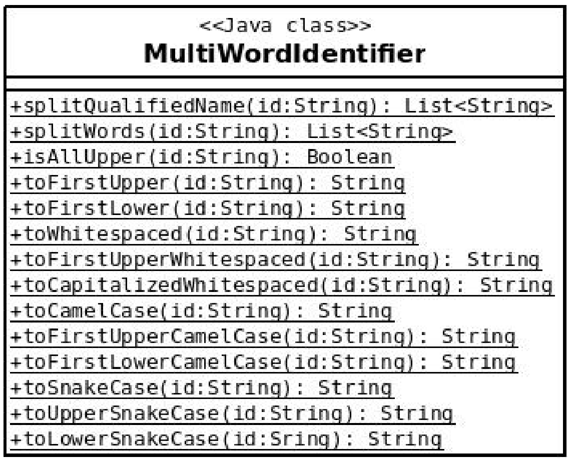

<<Java class>>
MultiWordIdentifier
+splitQualifiedName(id:String): List<String>
+splitWords(id:String): List<String>
+isAllUpper(id:String): Boolean
+toFirstUpper(id:String): String
+toFirstLower(id:String): String
+toWhitespaced(id:String): String
+toFirstUpperWhitespaced(id:String): String
+toCapitalizedWhitespaced(id:String): String
+toCamelCase(id:String): String
+toFirstUpperCamelCase(id:String): String
+toFirstLowerCamelCase(id:String): String
+toSnakeCase(id:String): String
+toUpperSnakeCase(id:String): String
+toLowerSnakeCase(id:Sring): String

Figure 6.3: Helper class for multi-word identifiers

### 6.4.3 Names

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[1]. 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 file or class template. Static names, which are independent of the input AOCS model, are defined with constant static strings. As an example, Listing 6.9 shows the definition of the name of the file containing the command interpreter template class, which happens to reuse other such strings.

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

Listing 6.9: Definition of a static name

Other names which must be dynamically created are given by static methods. Listing 6.10 shows one of these methods. This one in particular infers the name of the specific command handler class from the name of the command.

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

Listing 6.10: Method to derive a dynamic name

All of these variables and methods are public, so that they can be accessed by the other generator classes.

### 6.4.4 Generate methods

In the AOCS code generator, the creation of the target files is commanded by a group of methods prefixed with *generate* (hence the denomination '*generate* methods'). These methods receive as parameters the input model (a *Resource*), an object which provides access to the file system (implementing the `IFileSystemAccess` interface), and any needed additional objects. Listing 6.11 shows the *generate* method concerning the class that holds the instances of all the command handlers of a component (`<MyComponent>Commands`), as an example.

```
def package static void generateHandlerInstances(Resource resource, IFileSystemAccess fsa, Component
       component, Application application) {

        // Definition file
        generateHandlerInstancesDef(resource, fsa, component)

        // Implementation file
        generateHandlerInstancesImpl(resource, fsa, component, application)
}
```

Listing 6.11: *Generate* method for the component commands class

For each C++ class to be generated, two files are typically compiled: a header (or definition) file and an implementation file[2]. The `generateHandlerInstances` method abstracts away this detail. It is called by the main generator class `AOCSGenerator`, and then orders the generation of each individual file. It then calls, for example, the method `generateHandlerInstancesImpl`, shown in Listing 6.12, to create the implementation file. This file-specific *generate* method starts by creating an instance of

[1]Debug flags are used for conditional compilation, making use of the C preprocessor. If a flag is specified in the compilation command, additional code which prints debug messages is included in the target executable file.

[2]Exceptions are abstract classes and concrete classes with simple default implementations, which only require a definition file.

`FileInfo`. Then it must call the `generateFile` method of the file access object, providing the target relative file path and the file contents. The file contents are obtained from a so-called *compile* method (See Section 6.4.5), which receives the previously created `FileInfo` object. For the sake of brevity, the equivalent method for generating the definition file is not shown here.

```
def private static void generateHandlerInstancesImpl(Resource resource, IFileSystemAccess fsa, Component
    component, Application application) {
  val fileInfo = new FileInfo(
     MESSAGE_gen, // message
     component.commandInstancesImplFileName, // target file
     resource.URI.toPlatformString(false), // source file
     "Implementation of class to hold all " + component.className.toReadableName + " commands" // title
  )
  fsa.generateFile(
     component.commandInstancesImplFilePath, // file path
     compileHandlerInstancesImpl(fileInfo, component, application) // contents
  )
}
```

Listing 6.12: *Generate* method for the implementation file of the component commands class

### 6.4.5 Compile methods

As already stated in Section 6.4.4, methods prefixed with *compile* 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 Xtend.

Continuing with the example of the `<MyComponent>Commands` class, the method `compileHandlerInstancesImpl` is called to generate the contents of the implementation file. This method works at the file level, but other *compile* methods exist: class-level methods and helper methods. File-level *compile* methods are in charge of:

- Using the received `FileInfo` object to compile the file header.
- Compiling *include guards*[3] (in definition files).
- Compiling *include* directives for necessary header files.
- Compiling the opening and closing of namespaces.
- Compiling the declaration of used namespaces.
- Calling a class-level *compile* method to create the class definition/implementation.

The called class-level method (`compileHandlerInstancesClassImpl` in this case) will then return the actual contents of the class. Both file- and class-level *compile* methods make intensive use of small helper *compile* methods.

### 6.4.6 Library files

Xtend templates are not the only mechanism through which files can be generated in the AOCS DSL framework. A folder named *resources/library* exists in the main Eclipse plug-in project of the AOCS DSL.

[3] An *include guard* is a construct used to avoid double inclusion of header files when dealing with *include* preprocessor directives (as is the case with C/C++).

The `generateLibrary` method, shown in Listing 6.13, copies each file found in that folder (and subfolders) 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.

```
def void generateLibrary(Resource resource, IFileSystemAccess fsa) {

   // Get library folder
   val libPath = getLibPath()

   // Get all files in library folder
   val libFiles = getFileList(libPath)

   // Generate copies of the library files
   for (file : libFiles) {
      fsa.generateFile(
         getLibFileTargetPath(file), // file path
         getFileContent(file) // contents
      )
   }
}
```

Listing 6.13: MEthod to generate library files

### 6.4.7 Generator workflow

All the different kinds of classes, methods and variables involved in the code generation process have been introduced in the previous sections. Here the high-level workflow of the code generator is presented, to give a better understanding of what target files are created and under which conditions.

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 starts by creating the static files, followed by those which are conditioned by the input model, and finishes by copying the library files. It goes as follows:

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. To filter out these definitions, the generation of all the custom files concerning applications, components and commands is done hierarchically. Furthermore, this structure assigns the responsibility of generating logically contained elements to their container. The described workflow results in a minimum of eleven generated files, adding one file for each application, two for each component and four for each command.

## 6.5 Other features

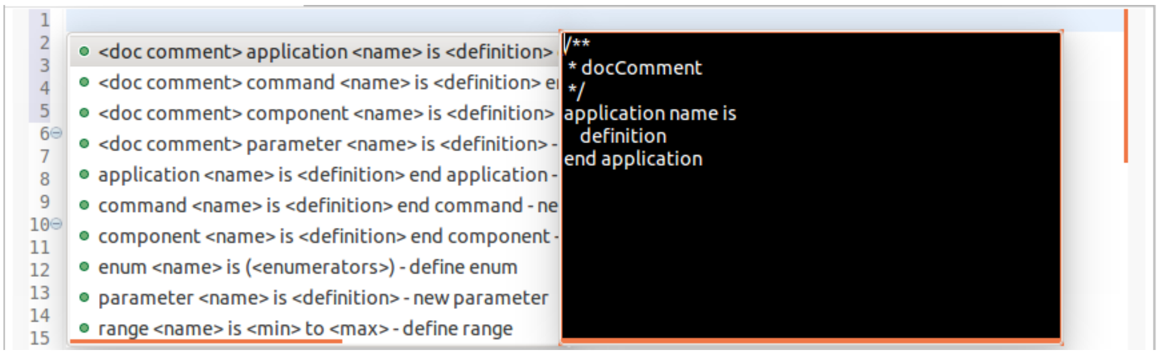
The essential components that need to be implemented for an Xtext-based DSL to be functional are the grammar, the model validator, the code generator and the scope provider. The first three were described in the previous sections. The scope provider, used for resolution of cross-references, was not mentioned because the default offered by Xtext is used. It allows forward referencing (referencing a variable before its declaration) and references to variables defined in an outer scope [43]. As this is exactly the desired behavior for the AOCS DSL, no customization was made.

Several other features can be implemented/customized with Xtext, mainly intended to improve the user experience. To make usage of the DSL easier in a way to increase acceptance, custom content assist was implemented, along with editor syntax highlighting and an automatic code formatter. This section describes these customizations.

### 6.5.1 Content assist

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.

A custom provider of template proposals (`AOCSTemplateProposalProvider`) was implemented. 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. For instance, if the user asks for completion proposals[4] in the very first line of an AOCS model file, he will get template proposals for each of the concepts of the language, since all are valid in that context. Figure 6.4 shows the mechanism in action.



```
● <doc comment> application <name> is <definition>
● <doc comment> command <name> is <definition> e
● <doc comment> component <name> is <definition>
● <doc comment> parameter <name> is <definition> -
● application <name> is <definition> end application -
● command <name> is <definition> end command - ne
● component <name> is <definition> end component -
● enum <name> is (<enumerators>) - define enum
● parameter <name> is <definition> - new parameter
● range <name> is <min> to <max> - define range
```

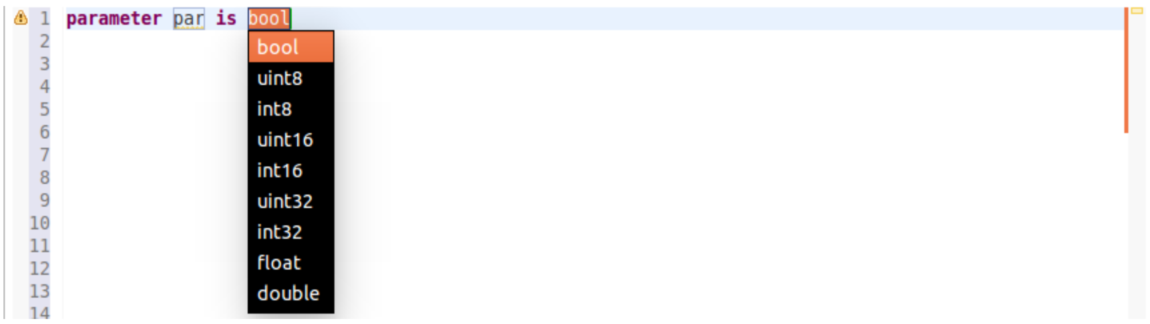
```
/**
 * docComment
 */
application name is
  definition
end application
```

Figure 6.4: Template proposals at model scope

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

[4]In Eclipse, completion proposals can be requested by simply pressing *ctrl+space*.

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. This is achieved by extending the class `AbstractTemplateVariableResolver`. The result can be seen in Figure 6.5.

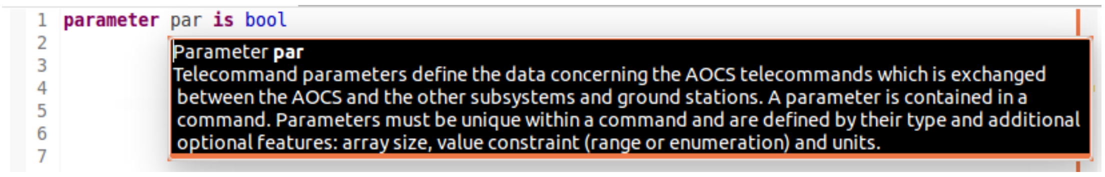


```
parameter par is bool
```

Figure 6.5: Automatic drop-down list to choose the parameter type

To avoid confusion, keywords which can be replaced by more complete template proposals are filtered out by customizing the `AOCSProposalProvider`.

Finally, editor hover assist was implemented to instruct the user on the meaning of each concept of the DSL. Figure 6.6 shows the information that the user sees when hovering a parameter definition.



```
parameter par is bool
```

Parameter **par**
Telecommand parameters define the data concerning the AOCS telecommands which is exchanged between the AOCS and the other subsystems and ground stations. A parameter is contained in a command. Parameters must be unique within a command and are defined by their type and additional optional features: array size, value constraint (range or enumeration) and units.

Figure 6.6: Hover assist for a parameter

### 6.5.2 Syntax coloring

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

As already mentioned in Section 6.1, the class in the Common UI plug-in which connects the custom highlighting configuration for the documentation comments is `CommonAntlrTokenToAttributeIdMapper`. This class is bound to Guice in the Common language. However, due to a bug in Xtext, this binding is not inherited by the AOCS language. This is an issue within Xtext and solving it is consequently out of the scope of this thesis work. The binding was simply manually added to the `AOCSUiModule` class.

### 6.5.3 Formatting

Text formatting is necessary for two tasks. One of them is to automatically format an AOCS textual model file. The other is applying quick fixes. As stated in Section 4.3, a quick fix can act upon the text or the Ecore model. In the latter case, changes to the model must be reflected in the text file, thus calling for a text formatter.

No quick fixes have been implemented in this framework. Nevertheless, a custom text formatter is relatively simple to implement and allows the user to easily and quickly format the textual model[5]. This customization was implemented in the class `AOCSFormatter`, thus overriding Xtext's default `OneWhitespaceFormatter` [43] which simply inserts a white space between each token.

[5]In Eclipse, simply pressing *ctrl+shift+f* automatically formats the code.

# Chapter 7

# Demonstration

The best way to show the capabilities of the developed tool is to perform a demonstration. This chapter does so in a tutorial-like form, guiding the reader through the steps of a simple example using the Eclipse IDE. Section 7.1 shows how to set up an AOCS project in Eclipse. Next, an example AOCS model is presented in Section 7.2, along with an overview of the automatically generated files. Section 7.3 then provides a step-by-step description of simple customizations to the generated files. Finally, the resulting software is tested in Section 7.4 to show that it works as expected.

## 7.1 Project setup

Since no specific project wizard got to be developed for the AOCS DSL framework, this section describes how to manually set up an AOCS project in Eclipse.

The first step is to create a new C++ project. For this example, an empty executable project was created and given the name *ExampleAOCSProject*. The project wizard also requires the user to select a toolchain[1] to use for compilation of the C++ code. The *Linux GCC* toolchain was selected in this case, but several appropriate options may exist.

Then a model file must be created. A simple empty file was created in the project folder with the extension *.aocs*. The empty model file was named *ExampleAOCSModel.aocs*. When a file with a DSL extension is created, Eclipse automatically asks the user if the Xtext nature should be added to the project[2]. After clicking "Yes", the code generator of the DSL is immediately called and the *src* and *src-gen* folders are created. Typically, the model file would be put into the *src* folder, but lets keep it separated from the target code for this demonstration.

At this point, the Xtext-based DSL is already working. However, a few additional steps are required to allow the compilation of the C++ code. First the compiler must be instructed on where to look for included files. To do that, lets open the project properties and find the "include paths", under the settings for the C++ compiler. Then, both the *src* and the *src-gen* folders must be added. In Eclipse, this can be specified with the following paths:

[1]A software development toolchain typically consists of a compiler, an assembler, a linker, a set of libraries and a debugger.
[2]See Section 4.1 for information on project natures.

```
${workspace_loc:/${ProjName}/src}
${workspace_loc:/${ProjName}/src-gen}
```

*Note*: If *Visual C++* was the toolchain chosen when creating the C/C++ project, an extra step is required here. The *Windows SDK Library* must be manually added in the "additional libraries" field of the C++ Linker settings, with the path:

```
${WindowsSdkLib}
```

The last step of the project setup is to provide an entry point for the program, so that it can be compiled and tested. In C++, the entry point is the so-called *main* function. A new C++ source file, named *ExampleMain.cc*, was created in the project folder, containing the code in Listing 7.1.

```
int main() {
    return 0;
}
```

Listing 7.1: Simple *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 7.1.

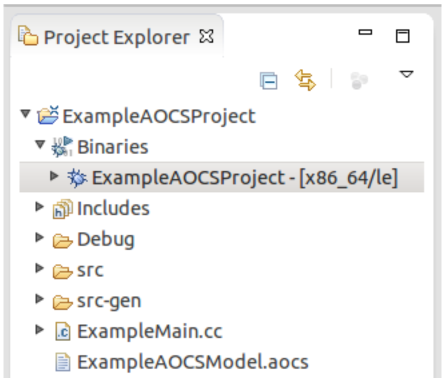
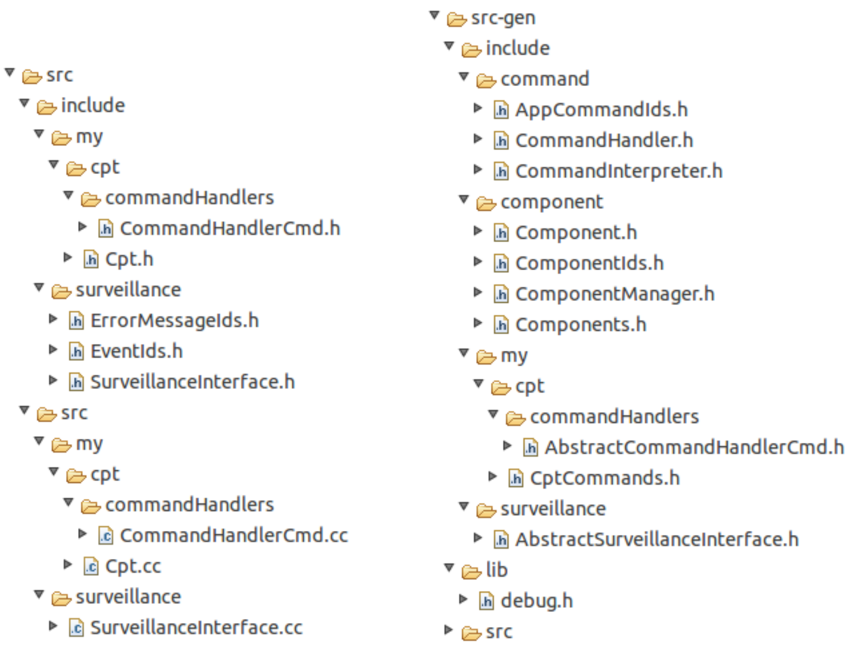


Figure 7.1: Project folder after compilation

## 7.2 An example model

A simple model was developed to show the possibilities of the AOCS DSL. The code, which can be seen in Appendix B.1, was put in the *ExampleAOCSModel.aocs* file. 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.

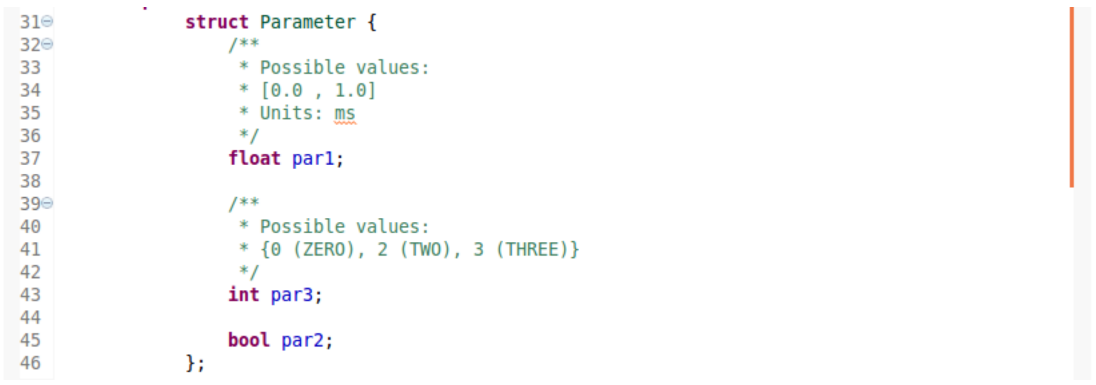
After saving the example model, the AOCS code generator produces a large number of C++ files, displayed in Figure 7.2. It can be noted that the files subject to customization are a minority. Note also the folder structure resulting from the `cpt` component name qualifier. The implementation files in the *src-gen* folder are not shown to avoid presenting an excessively long list.



```
src
  include
    my
      cpt
        commandHandlers
          CommandHandlerCmd.h
        Cpt.h
    surveillance
      ErrorMessageIds.h
      EventIds.h
      SurveillanceInterface.h
  src
    my
      cpt
        commandHandlers
          CommandHandlerCmd.cc
        Cpt.cc
    surveillance
      SurveillanceInterface.cc
```

```
src-gen
  include
    command
      AppCommandIds.h
      CommandHandler.h
      CommandInterpreter.h
    component
      Component.h
      ComponentIds.h
      ComponentManager.h
      Components.h
    my
      cpt
        commandHandlers
          AbstractCommandHandlerCmd.h
        CptCommands.h
    surveillance
      AbstractSurveillanceInterface.h
  lib
    debug.h
  src
```

Figure 7.2: Customizable (left) and purely generated (right) files

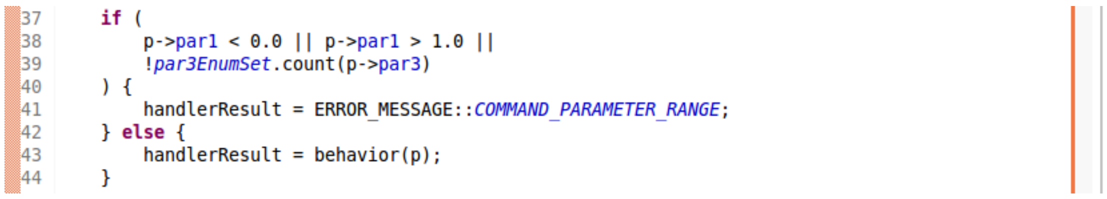


```
        struct Parameter {
            /**
             * Possible values:
             * [0.0 , 1.0]
             * Units: ms
             */
            float par1;

            /**
             * Possible values:
             * {0 (ZERO), 2 (TWO), 3 (THREE)}
             */
            int par3;

            bool par2;
        };
```

Figure 7.3: Parameter definition in *AbstractCommandHandlerCmd.h*



```
if (
    p->par1 < 0.0 || p->par1 > 1.0 ||
    !par3EnumSet.count(p->par3)
) {
    handlerResult = ERROR_MESSAGE::COMMAND_PARAMETER_RANGE;
} else {
    handlerResult = behavior(p);
}
```

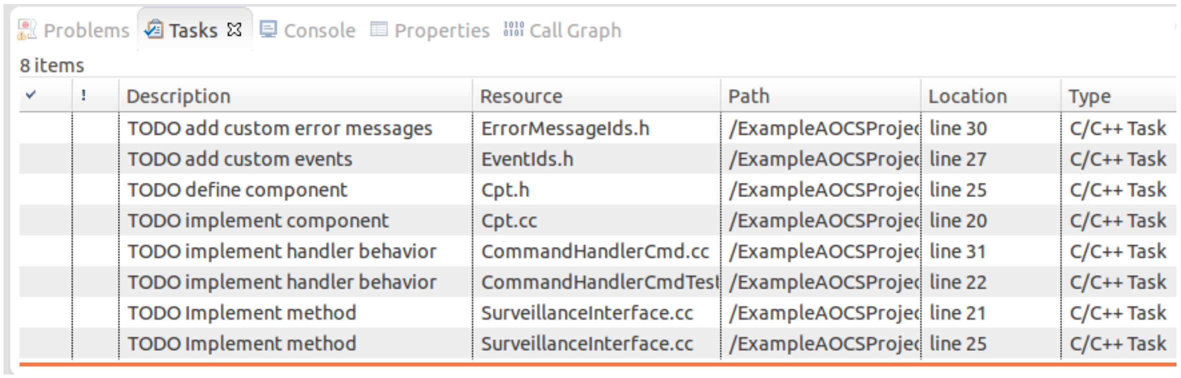
Figure 7.4: Command parameter value checking in *AbstractCommandHandlerCmd.cc*

The command handler for the `cmdTest` command is the one that is customized and tested in the next sections. However, one can look into the abstract handler class for the `cmd` command to see how the parameters defined in the example model are reflected in the target code. Figures 7.3 and 7.4 show the declaration of the `cmd` parameters and how their value constraints are applied, respectively. Note that a *set* is created as a member of the class for checking enumeration-constrained parameters.

## 7.3 Target code customization

In the previous sections an AOCS project was set up, and an example model file was created, from which the AOCS code generator derived a collection of target files for the system software. Some of the files were outputted to the *src* folder, so that they can be edited and customized. This section provides an example customization of those files.

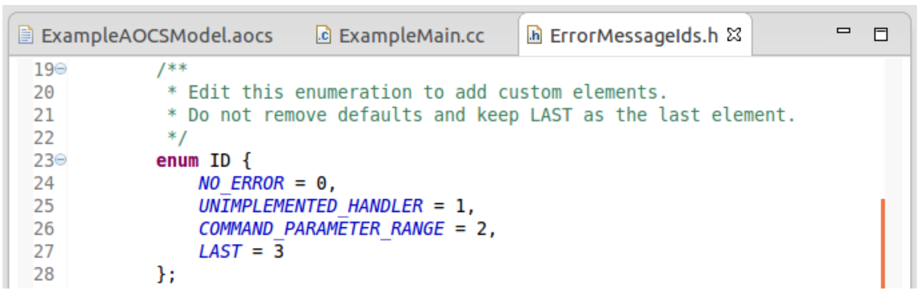
All generated classes, methods and enumerations needing (or allowing) customization contain a *TODO* task tag. These tags, which are written in the form of single-line comments, are recognized by Eclipse, and can be seen in the *Tasks* view. Figure 7.5 shows a screenshot of this view after the steps taken in the previous sections. It shows which task must be performed in which file, and the exact line of code where the tag is inserted.



| ✓ | ! | Description | Resource | Path | Location | Type |
|---|---|---|---|---|---|---|
| | | TODO add custom error messages | ErrorMessageIds.h | /ExampleAOCSProje | line 30 | C/C++ Task |
| | | TODO add custom events | EventIds.h | /ExampleAOCSProje | line 27 | C/C++ Task |
| | | TODO define component | Cpt.h | /ExampleAOCSProje | line 25 | C/C++ Task |
| | | TODO implement component | Cpt.cc | /ExampleAOCSProje | line 20 | C/C++ Task |
| | | TODO implement handler behavior | CommandHandlerCmd.cc | /ExampleAOCSProje | line 31 | C/C++ Task |
| | | TODO implement handler behavior | CommandHandlerCmdTest | /ExampleAOCSProje | line 22 | C/C++ Task |
| | | TODO Implement method | SurveillanceInterface.cc | /ExampleAOCSProje | line 21 | C/C++ Task |
| | | TODO Implement method | SurveillanceInterface.cc | /ExampleAOCSProje | line 25 | C/C++ Task |

Figure 7.5: Customization points in the *Tasks* view

Lets start by adding a custom error message to the correspondent enumeration. All the user has to do is double-click the task corresponding to the file *ErrorMessageIds.h*. Eclipse will open the file in the editor and highlight the tag. At this point, it can be seen that the file contains the definition of default IDs and instructions on how to edit it (see Figure 7.6). Following the provided instructions, a new error message labeled `EXAMPLE_ERROR` is introduced with the code '3', meanwhile attributing the code '4' to the special member `LAST`.



```
        /**
         * Edit this enumeration to add custom elements.
         * Do not remove defaults and keep LAST as the last element.
         */
        enum ID {
            NO_ERROR = 0,
            UNIMPLEMENTED_HANDLER = 1,
            COMMAND_PARAMETER_RANGE = 2,
            LAST = 3
        };
```

Figure 7.6: Default error message IDs

Now lets open the two files concerning the `cpt` component class, in order to define and implement a simple custom method. Again, double-clicking the corresponding entries in the *Tasks* view is all it takes. A boolean method named `exampleComponentMethod` is defined in the header file *Cpt.h* (Listing 7.2). Its implementation is put in the file *Cpt.cc*. Listing 7.3 shows that the method does nothing more than returning the boolean value *false*.

```
class Cpt : public Component {
public:
    bool exampleComponentMethod();
}; /* class Cpt */
```

Listing 7.2: Component definition

```
bool Cpt::exampleComponentMethod() {
    return false;
}
```

Listing 7.3: Component method implementation

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 file can be accessed directly through the corresponding task as well. The handler's `Parameter` structure has *protected* visibility[3], so a `Parameter` cannot be created, for example, in the `main` function. A handler without parameters, like the `CommandHandlerCmdTest` is easier to test, since no parameter validity check is conducted, so any *void* pointer can be provided as a parameter.

The handler implementation file contains commented *include* directives for the header files concerning the component manager and the component IDs. In order to use the component `cpt`, the user must uncomment these two lines and add an *include* for the file *Cpt.h* (see Listing 7.4).

```
#include "include/my/cpt/commandHandlers/CommandHandlerCmdTest.h"
#include "include/component/ComponentManager.h"
#include "include/component/ComponentIds.h"
#include "include/my/cpt/Cpt.h"
```

Listing 7.4: Headers included in *CommandHandlerCmdTest.cc*

Finally, the command `behavior` is implemented according to Listing 7.5. 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. From the method implementation shown earlier, it is obvious that the condition will not be verified, so the command handler is expected to return the custom error message `EXAMPLE_ERROR`, with associated value '3'.

```
ERROR_MESSAGE::ID CommandHandlerCmdTest::behavior(const Parameter* p) {
    // Declare result variable
    ERROR_MESSAGE::ID result;

    // Get component
    Cpt* cptPtr = static_cast<Cpt*>(ComponentManager::getLocation(COMPONENT::CPT));

    // Calculate result
    if (cptPtr->exampleComponentMethod() == true) // Not going to happen
        result = ERROR_MESSAGE::NO_ERROR;
    else
        result = ERROR_MESSAGE::EXAMPLE_ERROR;

    return result;
}
```

Listing 7.5: Custom code for the test command handler

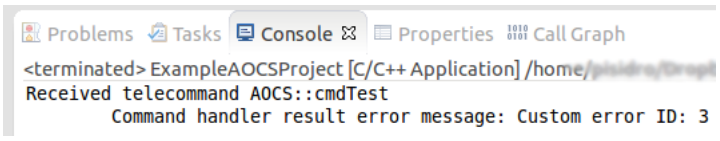
## 7.4 Testing

The last step of this demonstration is to test the command handler customized in Section 7.3, to check that it behaves as expected, *i.e.*, that it returns the error code '3'.

[3]Methods and members with *protected* visibility can only be accessed by objects of the class and subclasses.

Command handlers can be instructed to print debug messages by adding a debug flag to the compilation command. In Eclipse, the user can easily access the header file *CommandHandlerCmdTest.h* by left-clicking the corresponding *include* directive while pressing the *ctrl* key. The class definition is preceded by a comment indicating which debug flag should be used. Alternatively, simply hovering the class name in the `behavior` method implementation also shows this comment. The C++ compiler settings must be accessed to add the flag "`-D AOCS_TEST_COMMAND_HANDLER`". 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 `main` function created earlier. The implemented routine can be seen in Appendix B.2. It essentially orders the execution of `cmdTest` through the command interpreter, to simulate a telecommand received from a ground station. The project is subsequently rebuilt and run in Eclipse (as a *Local C/C++ Application*). Figure 7.7 shows the resulting message printed in the console. The message confirms the expected behavior, thus concluding this demonstration.



```
<terminated> ExampleAOCSProject [C/C++ Application] /home/
Received telecommand AOCS::cmdTest
        Command handler result error message: Custom error ID: 3
```

Figure 7.7: Resulting printed message

# Chapter 8

# Conclusions and recommendations

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.

## 8.1 Summary of the results

A literature review was conducted on related projects, *i.e.*, any projects whose results could be applied to solving or mitigating the identified problem, to show that this thesis work complements previous research and developments. In this process, the proposed solution was compared against existing possible alternatives, showing to be a promising approach. The comparison also indicates that the AOCS is significantly more complex than any other system to which textual DSM has been applied so far, so it cannot be expected to reach the level of simplicity of the studied example applications. Finally, it was seen that the use of DSL-based solutions fits in well with the ongoing efforts to standardize space systems.

The scope for implementation of a proof of concept was chosen to be centered in the telecommand handling functionality. With that in mind, the software of an orbiting satellite was analyzed, starting with the command handlers and exploring adjacent systems elements as needed. From this analysis, it was concluded that the proof of concept would need to be able to recreate not only the command handlers, but also the classes involved in managing them, as well as AOCS component classes and others to manage the components, and a few different files related to the surveillance component.

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 telecommand *Parameter* is the most complex concept of the AOCS semantic model, while other higher-level concepts are trivial. *Parameters* are contained in *Commands*, which are contained in *Components*, which are in turn contained in *Applications*, the top-level concept in the metamodel.

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. Although these elements require almost full manual implementation, they are generated to make the target code compilable, so that the developed solution does not present any drawback when compared to manual development of the system.

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.

Taking advantage of the cross-referencing capabilities of Xtext, the designed semantic model was extended on the implementation stage to give the user greater flexibility to separate a model definition by levels of abstraction. Validity check methods were implemented to continuously validate the model, providing feedback on its correctness. A file containing basic support for printing debug messages was included in the framework, to prove a feature of the code generator which allows it to simply copy existing files into the target folder. The user experience was further improved by implementing editor features like content assist, syntax highlighting and code formatting, making the workbench quite easy to use.

The workflow of the code generator results in a minimum of eleven generated files, to which are added: one for each application, two for each component and four for each command. 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 AOCS software developer.

To show the result of this thesis work, a tutorial-like demonstration of the developed DSW was performed. To begin, it was shown how to set up an AOCS project in Eclipse. Then, an example AOCS model was presented to illustrate the features of the language. 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.

## 8.2 Future work

This section addresses some unresolved problems and recommendations for future work, by the order in which they appeared during the development of the AOCS DSL.

The first unresolved problem to appear was mentioned in Section 6.5.2, and concerns the inheritance of syntax coloring customizations. The class in the Common UI plug-in which connects the custom highlighting configuration for the documentation comments is `CommonAntlrTokenToAttributeIdMapper`. It was bound to the dependency injection mechanism in the `CommonUiModule`. However, due to an issue within Xtext itself, it was observed that this binding is not inherited by the AOCS language.

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, so it's feasibility greatly depends on how interested possible stakeholders would be in its development. 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, partly due to the simplicity of existing DSL applications. Although its files are still perfectly readable due to their organization, looking back it seems like an intermediate M2M transformation would have made the implementation of this framework more modular. The Eclipse-based model transformation language ATL (Atlas Transformation Language) could possibly be used to bridge the gap between the Ecore model which results of the parsing process and the generated code. Following a similar pattern to the one used in EMF itself, 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 Section 7.1 of the product demonstration. This would significantly improve the ease of use of the tool.

Another product of SC (which was not mentioned in Section 5.1) is *Virtual Satellite* [53]. It is an IDE intended to support the design process of spacecraft systems over the full development life cycle. Its core element is an underlying data model using an OO approach, also defined with the EMF. The software also acts as a framework for research activities in the areas of MDE, Design Theory and Formal Verification & Validation (V&V). Given the promising results of the AOCS DSL, its development could be continued with the end goal of being integrated in the *Virtual Satellite* software.

# Bibliography

[1] TU Delft. Spacecraft bus subsystems. URL `http://www.lr.tudelft.nl/en/organisation/departments/space-engineering/space-systems-engineering/expertise-areas/spacecraft-engineering/design-and-analysis/configuration-design/subsystems/subsystems/`. Accessed 2014-09-27.

[2] W. J. Larson and J. R. Wertz. *Space Mission Analysis and Design*, volume 8 of *Space Technology Library*. Microcosm Press and Kluwer Academic Publishers.

[3] A. Jung and J. L. Terraillon. Faster, Later, Softer: COrDeT - an on-board software reference architecture, December 2010. URL `http://flightsoftware.jhuapl.edu/files/2010/FSW10_Jung.pdf`. Presentation at the *2010 Workshop on Spacecraft Flight Software*, in Pasadena, California, USA. Accessed 2014-09-23.

[4] A. Blum, V. Cechticky, A. Pasetti, and W. Schaufelberger. A Java-Based Framework for Real-Time Control Systems. In *Emerging Technologies and Factory Automation, 2003. IEEE Conference*, volume 2, pages 447–453, Lisbon, Portugal, September 2003.

[5] A. Pasetti and W. Pree. A Component Framework for Satellite On-board Software. In *18th Digital Avionics Systems Conference*, volume 2, pages 7.C.1–1–7.C.1–10, St. Louis, Missouri, USA, 1999.

[6] T. Erkkinen. Simulink Capabilities for Embedded Code Generation. URL `http://www.mathworks.com/company/events/miadc_03/MIADC_CodGen_May30.pdf`. Presentation at *The MathWorks International Aerospace and Defense Conference 2003*, in Natick, Massachussets, USA. Accessed 2014-09-21.

[7] J. Gärtner and W. Klinge. Advanced Methodologies for Aerospace, Automotive and Transportation software development, February 2004. URL `http://www.dlr.de/fs/portaldata/16/resources/dokumente/vk/vortrag_klinge_050203.pdf`. Presentation at *Braunschweiger Verkehrskolloquium*, in Braunschweig, Germany. Accessed 2014-09-21.

[8] A. Jung and J. L. Terraillon. SAVOIR - Software aspects of the reference architecture, October 2012. URL `http://congrexprojects.com/docs/12c25_2310/sa1025_jung.pdf?sfvrsn=2`. Presentation at *The 6th ESA Workshop on Avionics, Data, Control and Software Systems*, in Noordwijk, The Netherlands. Accessed 2014-09-14.

[9] A. Pasetti and W. Pree. A Reusable Architecture for Satellite Control Software, 2000. Department of Computer Science - University of Constance, in Constance, Germany.

[10] A. Pasetti. *Software Frameworks and Embedded Control Systems*, volume 2231 of *Lecture Notes in Computer Science*. Springer, 2002.

[11] M. Völter. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. Self-published, 2013.

[12] N. M. J. Basha, S. A. Moiz, and M. Rizwanullah. Model Based Software Development: Issues & Challenges. *International Journal of Computer Science & Informatics*, 2(1,2):226–230, 2012. InterScience.

[13] R. Viennau. Tech Views. *Software Tech News*, 12(4):3, January 2010. The Data & Analysis Center for Software, United States Department of Defense.

[14] P. H. Feiler and J. Hansson. Toward Model-Based Embedded System Validation through Virtual Integration. *Software Tech News*, 12(4):26–32, January 2012. The Data & Analysis Center for Software, United States Department of Defense.

[15] C. Petrone. Domain Specific Language for Magnetic Measurements at CERN, October 2009. Master's thesis at CERN, for the University of Sannio, in Benevento, Italy.

[16] S. Gretlein. Modeling of embedded designs - Part 1: Why model?, October 2012. URL `http://www.embedded.com/design/prototyping-and-development/4399743/Modeling-of-embedded-designs--Why-model--`. Accessed 2014-09-14.

[17] S. Gretlein. Modeling embedded designs - Part 2: Modeling method examples, November 2012. URL `http://www.embedded.com/design/prototyping-and-development/4400903/Modeling-embedded-designs--Modeling-method-examples-`. Accessed 2014-09-14.

[18] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[19] H. Kashif, M. Mostafa, H. Shokry, and S. Hammad. Model-Based Embedded Software Development Flow. In *4th International Design and Test Workshop*, pages 1–4, Riyadh, Saudi Arabia, November 2009.

[20] E. Tyugu and P. Grigorenko. Components in Model-Based Software Development. In *Computer Science and Information Technologies 2013*, pages 1–8, Yerevan, Armenia, September 2013.

[21] J. Küster. Model-Driven Software Engineering – Code Generation, 2011. URL `http://researcher.ibm.com/researcher/files/zurich-jku/mdse-08.pdf`. Lecture at IBM Research, Zurich, Switzerland. Accessed 2014-09-14.

[22] S. Efftinge, P. Friese, and J. Köhnlein. Best Practices for Model-Driven Software Development, June 2008. URL `http://www.infoq.com/articles/model-driven-dev-best-practices`. Accessed 2014-09-14.

[23] H. Behrens. Generation Gap Pattern, April 2009. URL http://heikobehrens.net/2009/04/23/generation-gap-pattern/. Accessed 2014-09-09.

[24] A. Natali. Introduction to EMF, Ecore and Xtext. URL http://edu222.deis.unibo.it/ANIS1213/CorsoIS1213BOLM/target/site/pdf/Models/IntroEmfEcoreXtext.pdf. University of Bologna. Accessed 2014-09-09.

[25] M. P. Ward. Language Oriented Programming. *Software Concepts and Tools*, 15:147–161, 1995. Association for Computing Machinery.

[26] V. Cechticky, G. Montalto, A. Pasetti, and N. Salerno. The AOCS Framework. In *International ESA Conference on Spacecraft Guidance, Navigation and Control Systems*, Frascati, Italy, October 2002.

[27] D. Darvas. ICE TEA: Domain-specific languages - What, how and when?, February 2014. URL https://indico.cern.ch/event/304218/contribution/0/material/slides/0.pdf. Presentation at CERN, in Zürich, Switzerland. Accessed 2014-09-11.

[28] J.-P. Tolvanen. Domain-Specific Modeling for Full Code Generation. *Software Tech News*, 12(4):4–7, January 2010. The Data & Analysis Center for Software, United States Department of Defense.

[29] R. Pohjonen. Boosting Embedded Systems Development with Domain-Specific Modeling. *RTC Magazine*, pages 57–61, April 2003. RTC Group.

[30] K. Dörfler and M. Vöelter. Programming Refrigerators with Eclipse Xtext, November 2011. URL http://www.voelter.de/data/presentations/RefrigeratorsAndDSLs.pdf. Presentation at *EclipseCon Europe 2011*, in Ludwigsburg, Germany. Accessed 2014-09-11.

[31] M. Veldthuis. Quby - A domain-specific language for non-programmers, August 2012. Master's thesis at the University of Groningen, in Groningen, The Netherlands.

[32] V. Vernon. Developing a Complex External DSL, April 2009. URL http://www.infoq.com/articles/External-DSL-Vaughn-Vernon. Accessed 2014-09-20.

[33] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In *9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 7–13, Orlando, Florida, USA, October 2009.

[34] M. Fowler. Domain-Specific Languages: An Introductory Example, September 2010. URL http://www.informit.com/articles/article.aspx?p=1592379. Accessed 2014-09-29.

[35] A. Pasetti. AOCS Framework Project, June 2002. URL http://www.pnp-software.com/AocsFramework/. Accessed 2014-09-21.

[36] V. Cechticky, P. Chevalley, A. Pasetti, and W. Schaufelberger. A Generative Approach to Framework Instantiation. In *2nd International Conference on Generative Programming and Component Engineering*, pages 267–286, Erfurt, Germany, September 2003.

[37] J.-L. Terraillon. SAVOIR: Reusing specifications to improve the way we deliver avionics. In *Embedded Real Time Software and Systems 2012*, Toulouse, France, February 2012. SAVOIR Advisory Group - European Space Agency.

[38] L. J. Hansen, P. Graven, D. Fogle, and J. Lyke. The Feasibility of Applying Plug-and-Play Concepts to Spacecraft Guidance, Navigation, and Control Systems to Meet the Challenges of Future Responsive Missions. In *7th International ESA Conference on Guidance, Navigation & Control Systems*, Tralee, County Kerry, Ireland, June 2008.

[39] J. Lyke, D. Fronterhouse, S. Cannon, and D. Lanza. Space Plug-and-Play Avionics. In *3rd Responsive Space Conference*, Los Angeles, California, USA, April 2005. American Institute of Aeronautics and Astronautics.

[40] J. Singer. SpaceDev Satellite Chosen to Ride SpaceX's Third Falcon 1 Rocket, May 2008. URL `http://www.space.com/5507-spacedev-satellite-chosen-ride-spacex-falcon-1-rocket.html`. Accessed 2014-09-22.

[41] M. Völter, D. Ratiu, B. Schätz, and B. Kolb. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, pages 121–140, Tucson, Arizona, USA, October 2012. Association for Computing Machinery.

[42] W. Beaton. Eclipse Platform Technical Overview (v3.1), April 2006. URL `https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`. Accessed 2014-10-01.

[43] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, August 2013.

[44] D. Lüdtke, S. Mece, M. Deshmukh, M. Bock, A. Schreiber, and A. Gerndt. A Framework to Model Metadata for Knowledge Management Tools. In *4th International Conference on Knowledge Management for Space Missions*, volume 4, Toulouse Space Show '12 – Toulouse, France, June 2012.

[45] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, and S. Zarnekow. Xtext user guide, 2008 - 2010. URL `http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf`. Accessed 2014-10-02.

[46] S. Föckersperger, K. Lattner, C. Kaiser, S. Eckert, S. Ritzmann, R. Axmann, and M. Turk. The On-Orbit Verification Mission TET-1 – Project Status of the Small Satellite Mission & Outlook for the One Year Mission Operation Phase. In *4S Symposium 2010 (Small Satellites Systems and Services)*, Funchal, Madeira, Portugal, May-June 2010.

[47] C. Bergin. Russian Soyuz-FG successfully launches five satellites, July 2012. URL `http://www.nasaspaceflight.com/2012/07/russian-soyuz-fg-launches-five-satellites/`. Accessed 2014-10-20.

[48] S. Föckersperger, K. Lattner, C. Kaiser, S. Eckert, W. Bärwald, S. Ritzmann, P. Mühlbauer, M. Turk, and P. Willlemsen. The Modular German Microsatellite TET-1 for Technology On-Orbit Verification. In *59th International Astronautical Congress*, Glasgow, Scotland, September-October 2008.

[49] S. Eckert, S. Ritzmann, S. Römer, and W. Bärwald. The TET-1 Satellite Bus – A High Reliability Bus for Earth Observation, Scientific and Technology Verification Missions in LEO. In *4S Symposium 2010 (Small Satellites Systems and Services)*, Funchal, Madeira, Portugal, May-June 2010.

[50] K. Brieß, W. Bärwald, E. Gill, H. Kayal, O. Montenbruck, S. Montenegro, W. Halle, W. Skrbek, H. Studemund, T. Terzibaschian, and H. Venus. Technology demonstration by the BIRD-mission. *Acta Astronautica*, 56(1,2):57 – 63, 2005. International Academy of Astronautics.

[51] K. Brieß, H. Jahn, E. Lorenz, D. Örtel, W. Skrbek, and B. Zhukov. Fire recognition potential of the bi-spectral Infrared Detection (BIRD) satellite. *International Journal of Remote Sensing*, 24(4): 865–872, 2003. Taylor & Francis.

[52] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Profesional Computing Series. Addison-Wesley, 2005.

[53] H. Schumann, A. Berres, O. Maibaum, and A. Röhnsch. DLR's Virtual Satellite approach. In *10th International Workshop on Simulation on European Space Programmes*, Noordwijk, The Netherlands, October 2008.

# Appendix A

# Grammar files

## A.1 Terminals

```
/*******************************************************************************
 * Copyright (c) 2008 itemis AG and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************************/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID        : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING    :
            '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
            "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
        ;
terminal ML_COMMENT  : '/*' -> '*/';
terminal SL_COMMENT  : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS        : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER:  .;
```

Listing A.1: Default *Terminals* grammar definition

## A.2 Common

```
/*
 * Common terminals and data types for all AOCS grammars
 */

grammar de.dlr.lambda.dsl.common.Common with org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT,
    SL_COMMENT)

generate common "http://www.dlr.de/lambda/dsl/common/Common"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

/*
 * Production rule
 *
 * Needed to generate EPackage and for testing
 */

CommonModel: {CommonModel}
    (
        'AnyInt' intValue=AnyInt
        & 'SignedInt' signedInt=SignedInt
        & 'Real' real=Real
        & 'HEX' hex=HEX
        & 'QualifiedID' qid=QualifiedID
    )
;

/*
 * Data type rules
 */

QualifiedID returns ecore::EString: ID ('.' ID)*;

AnyInt returns ecore::EInt: SignedInt | HEX;

/*
 * Data types with value converters
 */

Real returns ecore::EDouble: SignedInt '.' INT (('E'|'e') ('+'|'-')? INT)?;

SignedInt returns ecore::EInt: ('+'|'-')? INT;

/*
 * Custom/overridden terminal rules
 */

terminal HEX returns ecore::EInt: '0' ('x'|'X') ('0'..'9'|'a'..'f'|'A'..'F')+;

/**
 * For documentation comments to work, ML_COMMENT (from Terminals)
 * must be overridden
 */
terminal DOC_COMMENT: '/**' -> '*/';
terminal ML_COMMENT: '/*' (!'*') -> '*/';
```

Listing A.2: *Common* grammar definition

## A.3 AOCS

```
grammar de.dlr.lambda.dsl.aocs.AOCS with de.dlr.lambda.dsl.common.Common hidden(WS, ML_COMMENT,
      SL_COMMENT)

generate aOCS "http://www.dlr.de/lambda/dsl/aocs/AOCS"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

AocsModel:
    (applicationDefs+=Application
        | componentDefs+=Component
        | commandDefs+=Command
        | parameterDefs+=Parameter
        | enumerationDefs+=Enumeration
        | rangeDefs+=Range
    )*
;

Application:
    comment=DOC_COMMENT?
    'application' name=ID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | commandDefs+=Command
            | componentDefs+=Component
            | 'component' componentRefs+=[Component|QualifiedID]
        )*
    'end' 'application'
;

Component:
    comment=DOC_COMMENT?
    'component' name=QualifiedID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | commandDefs+=Command
            | 'command' commandRefs+=[Command]
        )*
    'end' 'component'
;

Command:
    comment=DOC_COMMENT?
    'command' name=ID 'is'
        (enumerationDefs+=Enumeration
            | rangeDefs+=Range
            | parameterDefs+=Parameter
            | 'parameter' parameterRefs+=[Parameter]
        )*
    'end' 'command'
;

Parameter:
    comment=DOC_COMMENT?
    'parameter' name=ID 'is'
    type=Type (array?='array' '(' arraySize=INT ')')?
    (
        (constrained?='in' (
            rangeConstrained?='range' (range=AnonymousRange | rangeRef=[Range])
            | enumConstrained?='enum' (enumeration=AnonymousEnumeration | enumerationRef=[Enumeration])
        ))?
        & ('with' 'units' unit=MeasurementUnit)?
    )
;

Range:
    'range' name=ID 'is' min=NumberLiteral 'to' max=NumberLiteral
;

Enumeration:
    'enum' name=ID 'is' '(' enumerators+=Enumerator (',' enumerators+=Enumerator)* ')'
;

AnonymousRange returns Range:
    min=NumberLiteral 'to' max=NumberLiteral
;

AnonymousEnumeration returns Enumeration:
    '(' enumerators+=Enumerator (',' enumerators+=Enumerator)* ')'
;

Enumerator:
    name=ID (explicit?='=' value=SignedInt)?
;

/**
 * Used for syntax coloring and content assist
 */
Alternatives:
    Type | MeasurementUnit
;

Type:
    {BooleanType} code='bool'
    | {IntegerType} code=('uint8'|'int8'|'uint16'|'int16'|'uint32'|'int32')
    | {FloatType} code=('float'|'double')
;

MeasurementUnit:
    {TemperatureUnit} code=('C'|'F') |
    {CurrentUnit} code=('A'|'mA') |
    {ElectricChargeUnit} code='Ah' | // Ampere-hour
    {PowerRatioUnit} code='dBm' | // Decibel-milliwatts
    {AngleUnit} code=('deg'|'rad') |
    {AngularSpeedUnit} code=('deg/s'|'rpm'|'rad/s') |
    {DistanceUnit} code=('km'|'m') | // meter
    {TorqueUnit} code=('mNm'|'Nm') | // millinewton-meter
    {TimeUnit} code=('s'|'ms') |
    {VoltageUnit} code=('V'|'mV') |
    {MagneticFluxDensityUnit} code=('nT'|'T') // Tesla
;
```

```
NumberLiteral: IntegerLiteral | RealLiteral;

IntegerLiteral: value=SignedInt;

RealLiteral: value=Real;
```

Listing A.3: *AOCS* grammar definition

# Appendix B

# Demonstration files

## B.1 Example AOCS model

```
/*
 * Example AOCS model
 */

/**
 * Documentation comments preceding applications, components,
 * commands or parameters will be included in the target code
 */
application app is
   /*
    * Components can be given qualified names for the purpose
    * of organizing the generated files, but they will be treated
    * by their last name only, so it must be unique.
    */
   component my.cpt is
      // Command definition
      command cmd is
         // Parameter definition
         parameter par1 is float in range 0.0 to 1.0 with units ms

         /*
          * Components, commands, parameters, ranges and enums can also
          * be referenced, as long as their definition is in scope
          */
         parameter par2
         parameter par3 is int32 in enum ENUM
      end command

      // Command with no parameters for testing
      command cmdTest is
      end command
   end component
end application

// Definition of parameter to be referenced within a command
parameter par2 is bool
// Definition of enum to be referenced within a parameter
enum ENUM is (ZERO, TWO=2, THREE)
```

Listing B.1: *ExampleAOCSModel.aocs*

## B.2 Test main function

```
/*
 * ExampleMain.cc
 *
 *  Created on: Oct 12, 2014
 *      Author: pisidro
 */

#include "include/component/ComponentManager.h"
#include "include/component/ComponentIds.h"
#include "include/component/Components.h"
#include "include/command/CommandInterpreter.h"
#include "include/command/AppCommandIds.h"
#include "include/my/cpt/commandHandlers/CommandHandlerCmd.h"
#include "include/my/cpt/CptCommands.h"

using namespace AOCS;

int main() {

   // Register components
   Components componentInstances;
   componentInstances.registerComponents();

   // Register commands of component Cpt
   CptCommands cptCommandInstances;
   cptCommandInstances.registerCommands();

   // Get command interpreter
   CommandInterpreter<APP_COMMAND::ID, APP_COMMAND::LAST, APP_COMMAND::OFFSET>* tcInterpreter
      = static_cast<CommandInterpreter<APP_COMMAND::ID, APP_COMMAND::LAST, APP_COMMAND::OFFSET>*>
         (ComponentManager::getLocation(COMPONENT::COMMAND_INTERPRETER_APP));

   // Telecommand code
   int tcCode = APP_COMMAND::OFFSET + APP_COMMAND::CMD_TEST;

   // Mock pointer to parameter
   void* voidParameterPtr;

   // Expected to print "Custom error ID: 3"
   tcInterpreter->executeCommand(tcCode, voidParameterPtr);

   return 0;
}
```

Listing B.2: *ExampleMain.cc*