

A Model-Driven Approach for the Development of an IDE for Spacecraft On-Board Software

Luigi Pomante

Sante Candia

Emilio Incerto

Università degli Studi dell'Aquila
Center of Excellence DEWS - ITALY
luigi.pomante@univaq.it

Thales Alenia Space Italy
Software Solutions Competence Center
sante.candia@thalesaleniaspace.com

Gran Sasso Science Institute
INFN - ITALY
emilio.incerto@gssi.infn.it

Abstract—This paper presents the application of a Model-Driven Engineering (MDE) approach to the aerospace domain. Specifically, it shows the Model-Driven Development (MDD) of an Integrated Development Environment (IDE) for a Domain-Specific Language (DSL) targeted to the achievement of the so called “Spacecraft on-board software flexibility”. In fact, the goal of the presented work has been to deploy a full-featured IDE to be used for the development of the “On-board Command Procedures” (OBCPs). The OBCPs coding is done by using the “OBCP Definition Language” (ODL), specified by Thales Alenia Space Italy (TASI) on the basis of the requirements stated in the “Space Engineering: Spacecraft On-board Control Procedures” ECSS standard (ECSS-E-ST-70-01, 16 April 2010). This standard does not impose specific language syntax but provides the guidelines for its specification. By following such guidelines and by exploiting some MDE technologies and tools, such as Eclipse Modeling Framework (EMF) and Xtext, it has been possible to realize an Eclipse-based IDE able to provide to the ODL developer the entire features essential in a modern environment for software development. The considered features include the “traditional” ones as syntax-highlighting, code-completion, version-control, on-line error-checking, and also “advanced” ones like syntactic validation, semantic validation, and integrated code compilation. Moreover, by means of the adopted MDE approach, a very large part of the IDE code has been automatically generated starting from the Extended Backus-Naur Form (EBNF) specification of the ODL grammar so allowing for the IDE developers to be more focused on validation issues and on the quality of product than on the coding activity. All this has been obtained by following the paradigm “coding equals modeling”, for which each program represents a behavioral model compliant to the meta-model specified by the grammar of the language itself. The obtained result is a professional product that satisfies all the expected requirements, but this would be just a starting point since the ultimate goal of this work is to contribute to fostering the adoption of MDE approaches in the spacecraft software domain.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. ECSS-E-ST-70-01C STANDARD & OBALEX ...	2
3. ODL-IDE MODEL-DRIVEN DEVELOPMENT ...	3
4. CONCLUSIONS.....	15
REFERENCES.....	16
BIOGRAPHY	17
ACKNOWLEDGEMENTS.....	17
GLOSSARY	17

1. INTRODUCTION

Each unmanned spacecraft launched into space needs to be remote-controlled by ground operators. Thus, it is necessary to establish a communication-link between the spacecraft and such operators. However, in an environment like space, where the only energy comes from the sun, a communication link is not always guaranteed. In fact, for missions significantly remote from Earth, two important operational constraints are the round-trip travel time of the radio signal to the spacecraft, and the coverage available from heavily booked deep space stations. In this scenario, the use of the *On-Board Control Procedures* (OBCPs) offers one means to alleviate such constraints: an OBCP can be seen as an on-board operator capable of performing dedicated tasks in real-time. For this reason, the OBCPs are increasingly becoming a de-facto standard in operations of spacecraft. An OBCP is a small program that is executed on-board, which can easily be remotely loaded, executed, and also replaced, without modifying the remainder of the on-board software. Moreover, OBCPs are “isolated”, in the sense that possible faults propagation and illegal memory accesses cannot occur. The massive use of OBCPs, along with the newest technologies, has induced the *European Space Agency* (ESA) to define new standards for the OBCPs implementation for spacecraft. Nowadays, OBCPs are brand new programs which can execute (even by mutual cooperation, thanks to the *OBCPs Global Data Area*, i.e. *OGDA*, used to share data among OBCPs) very complex procedures on board, resulting in a decrease of the number of interventions from Earth and improving the spacecraft autonomy during non-visibility or long round-trips periods. Given the importance of such technology, *Thales Alenia Space Italy* (TASI) has realized an environment where it is possible to produce, test and execute OBCPs: the *On Board Algorithm Executor* (*obAlex*). Moreover, TASI has defined its on-board software (named the “On-board Engine”) and its related language: the *OGDA & OBCP Definition Language* (ODL). The ODL allows the coding of both the *On-board Command Procedures* (OBCPs) and of the *OBCPs Global Data Area* (OGDA).

In this context, this paper presents the application of a *Model-Driven Engineering* (MDE) approach to realize a full-featured *Integrated Development Environment* (IDE) to be used with the *OGDA & OBCP Definition Language* (ODL). In fact, ODL is a *Domain Specific Language* (DSL)

aimed at implementing the “Spacecraft on-board software flexibility” concept. Its definition has been driven by the *Space Engineering: Spacecraft On-board Control Procedures ECSS Standard* (ECSS-E-ST-70-01C) [16] adopted, among others, by TASI for the development of its own products. This standard does not impose specific language syntax but provides the guidelines for its specification. Moreover, the standard defines the characteristics of the OBCPs development environment. By following such guidelines and by exploiting some MDE technologies and tools, as *Eclipse EMF* [6] and *Xtext* [7], it has been possible to realize an Eclipse-based IDE able to provide to the OBCPs developer the entire features essential in a modern software development environment. The considered features include both the basic ones, as *syntax-highlighting*, *code-completion*, *version-control*, *on-line error-checking* and also advanced ones like *syntactical validation*, *semantic validation*, and *integrated code compilation*. Moreover, by means of the adopted MDE approach, a very large part of the IDE code has been automatically generated starting from the *Extended Backus-Naur Form* (EBNF) specification of the ODL grammar so allowing the IDE developers to be more focused on validation issues and on the product quality than on coding activity. All this has been obtained by following the paradigm “coding equals modeling”, for which each program represents a behavioral model compliant to the meta-model specified by the grammar of the language itself. The obtained result is a professional product that satisfies all the expected requirements. However, this would be just a starting point since the ultimate goal of this work is to contribute to fostering the adoption of MDE approaches in the spacecraft software domain.

The paper is structured as follow. Section 2 introduces the main features of the ECSS-E-ST-70-01C standard and of the *obAlex* environment. Section 3 describes in detail the performed ODL-IDE Model-Driven Development. Finally, Section 4 draws out some conclusions and presents possible future developments.

2. ECSS-E-ST-70-01C STANDARD & OBALEX

This section provides more details about the environment where the IDE has been developed and the requirements imposed to the IDE itself. It allows assigning a clear meaning to the OBCPs, their execution-context and how they are described by means of the ODL. The section also summarizes the ODL capabilities.

Overview

Packet Utilization Standard (PUS) – ECSS and CCSDS standards recommendations address the end-to-end transport of telemetry and telecommands data between user applications on the ground and application processes on-board the satellite, and the intermediate transfer of these data through the different elements of the ground and space segments. *Packet Utilization Standard (PUS, ECSS-S-ST-70-41C)* [15] complements these standards by defining the

application-level interface between ground and space, in order to satisfy the requirements of electrical integration and testing and flight operations. The operations concepts are identified so that they imply distinct sets of capabilities to be implemented on-board the satellite. These sets of capabilities constitute “on-board services”. *PUS 18* is the service that allows the *on-board procedures management*.

On-Board Control Procedures (OBCP) - OBCPs have been already used for many years for ESA’s operative missions, as it can be read in the introduction of ECSS-E-70-01C [16]: “On-board control procedures (OBCP) have been implemented on an ad-hoc basis on several European missions over the last 25 years, so the validity and utility of the concept has been widely demonstrated”. The purpose of having on-board control procedures is to allow the ground operators to be able to prepare and uplink complex operations sequences (more complex than simple sequences of mission time-line telecommands) to be executed on-board during the mission operational phase. This is possible because the OBCPs run in a well isolated subsystem, so the creation of a new procedure does not require modification, uplink and re-validation of the whole on-board software. The OBCP subsystem allows these control procedures to be developed, qualified on ground, and executed on the spacecraft. During the years the OBCPs were improved both to fulfill PUS 18 requirements and to offer new functionalities to ground operators. Indeed, the new OBCP standard implements many capabilities as, for example, flow controls instructions, expressions, and many more as noted in ECSS-E-70-01C [16]. Thus, OBCPs provide a means to control the spacecraft through small script-like programs written in a specific language called ODL. The OBCP system consists of an OBCP preparation environment located on the ground and an OBCP execution environment that is located on-board (a ground instance of the OBCP execution environment is also available for the conduction of the pre-validation of the OBCPs). The OBCP preparation environment includes the OBCP code production and editors to support the scripting of OBCPs. The OBCPs are developed and validated in this environment, and are maintained within a repository called the “engineering data repository”, from which the execution environment obtains the OBCP images to be executed. The OBCP execution environment at ground includes dedicated services for uplinking OBCPs, monitoring and controlling their execution. The on-board environment consists of one or more OBCP engines that manage the execution of OBCPs.

On-Board Algorithms Executor (obAlex) - *obAlex* is an integrated environment for OBCPs specification and execution. This environment fulfills the new ECSS-E-70-41C [15] and ECSS-E-ST-70-01C [16] standards. The *obAlex* environment includes ground tools and on-board tasks and resources. *Ground obAlex* is integrated with the *Thales Alenia Space Avionics Services Worldwide* (ASW) production environment. *On-board obAlex* is integrated with the ASW PUS services, low-level services and scheduling mechanisms.

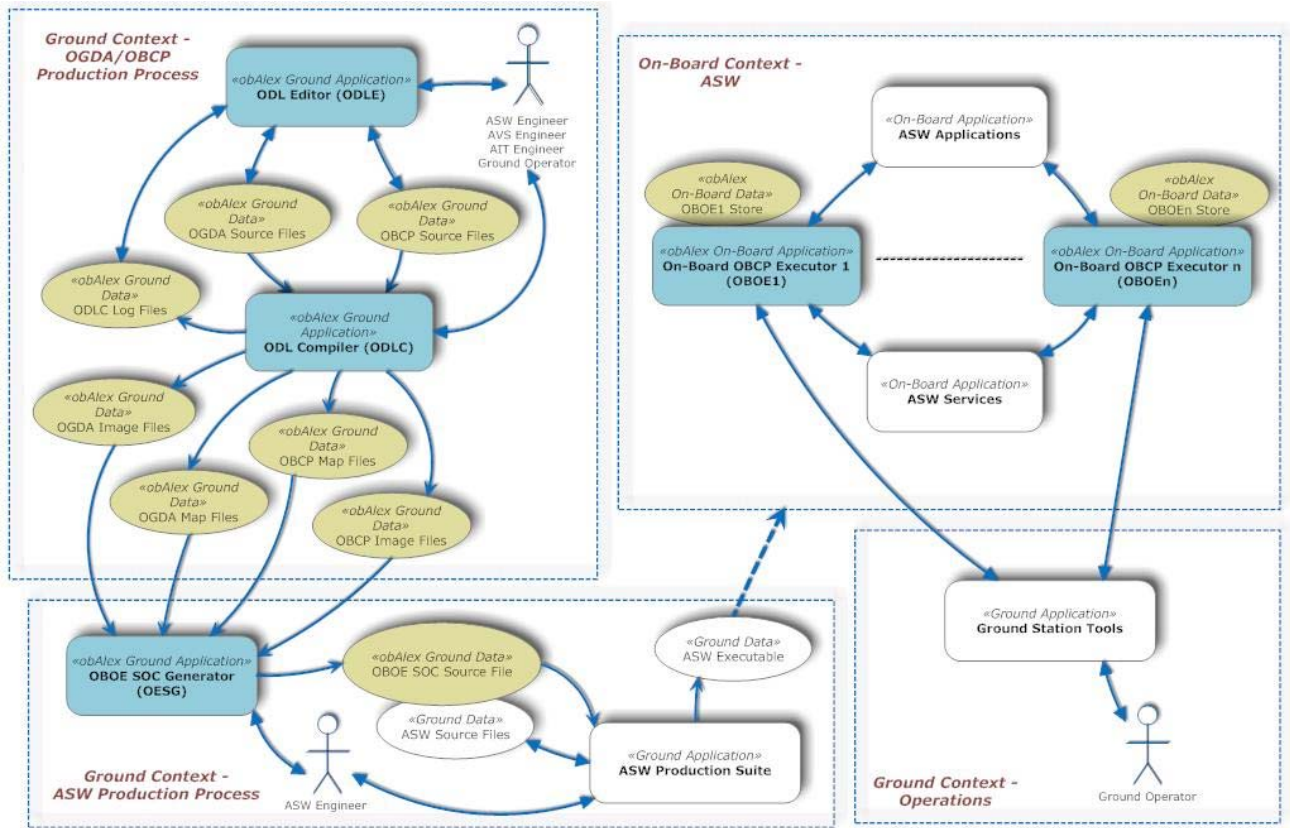


Figure 1 - obAlex overview

The on-board part of the environment is in charge of controlling the execution of the OBCPs loaded in its execution area and started by ground (by telecommand) or by another OBCP. It has a store where the OBCPs are saved and it is initialized from a *Static Database* that defines the basic operations of the spacecraft mission. The store contains an *OBCP Global Data Area* (OGDA) and a set of *On-Board Command Procedures* (OBCP). The *On-Board Command Procedures Engine* (OBOE) is the software that implements these functionalities. More than one OBOE can be executed on the spacecraft with associated OBCPs. In this environment, the OBCPs implement a set of actions to be executed on-board under the OBCP engine control. Each OBCP can access to global data declared in an OGDA. The OBCPs and OGDA are specified using the *OBCP and OGDA Definition Language* (ODL) and are compiled by an *ODL Compiler* [12]. The ground part of the environment is composed of all the software needed to produce OBCPs that will be subsequently sent to the spacecraft engine store. Thus, the ground part of the environment includes:

- *ODL Compiler* (ODLC) that compiles OBCPs and OGDA and produces a log, the images and a map file. The latter is a file that contains the association between the data identifiers used in ODL source and relative data index in compiled image. It contains also a useful description of all words of compiled OBCP.
- *ODL Editor* (ODLE) used to specify the OGDA and

OBCPs source file with code highlighting. The ODLE helps ground operators to write OBCP source correctly with syntax error checking and auto generation of project file.

- *OBCP Engine Static Database Generator* (OESG) that is used to generate the *Ada* source files containing the definition of all the OBCPs images which are part of the on-board static database.

Figure 1 shows how the Ground and On-Board contexts are interconnected in *obAlex* environment.

The OBCP has a structure that is firstly defined in ECSS-E-70-01C [16]. The OGDA structure is defined in *obAlex* environment and it is a special OBCP that has only a declaration section where global data are declared. The ODL language enables the end-users to define the script of the OBCP, making reference to system elements, activities, reporting data and events that are fully-defined within the engineering data repository.

3. ODL-IDE MODEL-DRIVEN DEVELOPMENT

Model-Driven Engineering (MDE) is a development method that emphasizes the creation of software from models characterized by a high degree of abstraction with the aim to increase productivity by reducing programming errors and facilitating the software refactoring as a result of future developments or changes in the requirements.

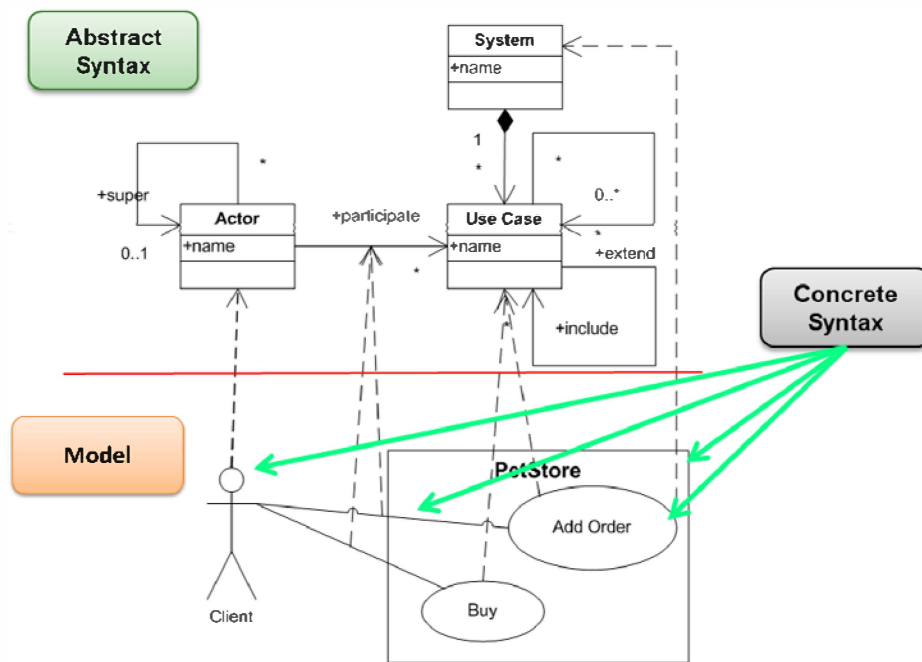


Figure 2 - Formal elements of a model

Essentially, the developer can describe the desired system through the definition of a set of models and generate code from these ones. The degree of self-generation is variable and ranges from simple skeleton classes to the entire application.

According to this definition, every developer is a model-driven software developer. Whenever he writes a program in any third generation language as Java, PHP, C#, implicitly he realizes a behavioral model of the system to be finally implemented by means of some low level machine-executable language. Following this stream, it becomes then possible to focus on the customization of the generated code and its tests. The Model-Driven approach “promises to be the first true generational leap in software development since the introduction of compilers”, it is based on the concepts of “model”, “meta-model” and in general “meta-modeling architecture”.

What is a model?

“A model is a coherent set of formal elements describing something (for example, a system, a bank, a phone, or a train) built for some purpose that is amenable to a particular form of analysis” [1]. The crucial point of this definition is the concept of “formalism” that is, and must be, at the base of each model. Only in this way, the MDE approach would be able to offer the possibility of automatic validation and transformation of a model with a high degree of abstraction into a running system (i.e. an implementation). More formally, in a Model-Driven context [2], each model is written in a particular language, and it is possible to differentiate them by four main aspects (Figure 2) :

- **Abstract Syntax:** it defines the syntactic structure (i.e. the elements and their relationships) that a model must

possess in order to be considered compliant with respect to a given language. For example, when you define a class diagram in UML, in order to be considered syntactically valid, it cannot contain elements not foreseen from the abstract-syntax of UML itself.

- **Concrete Syntax:** it defines the concrete rendering (graphical or textual) of the elements specified in the abstract syntax and therefore, it determines the way in which a model is created and how it appears to its user. For example, in the UML world, concrete syntax represents the set of 2D geometric shapes used to build and display models.
- **Semantic:** it defines the semantic of the concepts contained in the models, what a language is representing, and its meaning. An example is the execution of the implementation of a construct described in a java (dynamic semantics) model (program).
- **Level of abstraction:** it is the distance between the concepts used in the abstract syntax and those in the domain of modeling. The smaller this distance, more abstract the language, which, in this way, is as close as possible to the one used by the experts of the domain under consideration.

The abstract syntax of a model represents the formalism followed during the process of abstraction. Its definition is crucial for the creation of useful models for several purposes (analysis, communication, code generation). To define a new abstract syntax means to define a new modeling language that is the formal structure of future abstractions.

Meta-Model and Meta-Modeling Architecture

From a pragmatic point of view, the main activity that commits the developer during the process of *Model-Driven Development* (MDD) is to define new DSL languages suitable to provide the right abstractions (modeling) on the application domain under consideration. The definition of these languages passes through the creation of new *Meta-Models*, where each of these:

- describes the abstract syntax of the language being defined;
- defines the structure of the static validation of future models;
- influences the definition of transformation rules that will be based on the meta-model of the original model and the arrival point (in the case of model-to-model transformations);
- influences the definition of the template generation code that will be based on the structure of the meta-model.

It is worth noting as the adoption of a particular meta-model doesn't constraint to use a special concrete syntax. Several concrete syntaxes could be defined later and they can be associated to the same meta-model.

The creation of new meta-models, also known as *meta-modeling*, is, as mentioned, a practice central to any model-driven business. Very often it is possible to refer to it as a "conceptual modeling" because, in each effective meta-model, they must be contained entities and relationships that make up the ontology of the domain on which it is desired to perform the modeling. It is possible to think of each meta-model as a model of models defining the structure of all possible models conform to it. It is like the grammar of a programming language that defines the structure of all the syntactically valid programs.

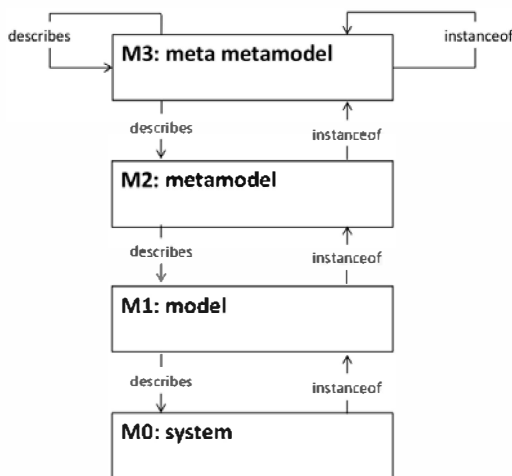


Figure 3 - Meta-Modeling Architecture

Figure 3 shows the *Meta-Modeling Architecture*. It represents the formal conceptual tool used to create new DSL and models:

- at level *M0* of this architecture there is the system to be modeled, i.e. the object of the modeling activity (in this paper a running software);
- at level *M1* there are all the models (of the final system) made in a particular *Modeling Language* (ML), i.e. the output of the modeling activity;
- at level *M2* there is the *Meta-model ML* (MML). It describes the structure of all possible models obtainable by means of a ML, i.e. the rules to be followed by using ML;
- at level *M3* there is the language used to write the *Meta-model MML*. It is called *MetaMeta-Model* and it is the model of a meta-model. It is defined in terms of itself.

Each element of a layer is involved in a relationship of type *instanceOf* with the layer above it. So, each level is composed of a set of templates written in function of the abstract-syntax specified by the model contained in the upper layer, namely, by its meta-model. Viewed from M0 to M3 relations are of the type *describe*, i.e. a formal expression of the concept that the layers observed from the top down may be one model of the other. By setting a precise metameta-model determines the language that will be used to write the meta-model, i.e. the new DSL.

For example, choosing the *Meta Object Facility* (MOF) [3] as metameta-model, it is an implicit reference to the meta-modeling architecture, defined by the *Object Management Group* (OMG), called *Model Driven Architecture* (MDA) and shown in Figure 4.

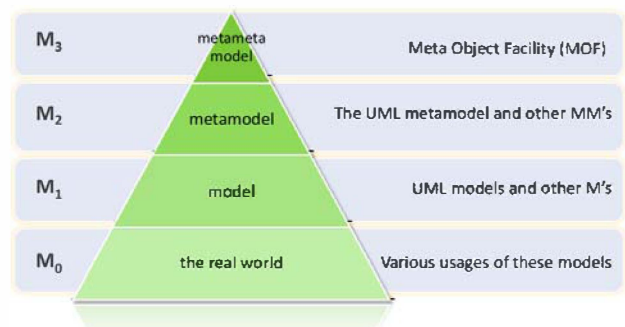


Figure 4 - OMG MDA: Model Driven Architecture

ODL-IDE & MDE workflow

By following a MDD methodology, the typical workflow for the development of an application can be decomposed in two main sections:

- **Meta-modeling:** from the study of the problem domain to define the meta-model (MM). This is constructed so that it is suitable for the creation of models, with the right level of abstraction, and so that they can contain

concepts useful to describe the application domain and so relevant for the realization of the final application. It is then possible to generate the editor (*concrete syntax*) and validators (*static semantic*), necessary for the manipulation and the validation of the models conform to the MM. Last but not least, it is the possibility to create transformations, i.e. *parser*, making it possible to transform models conforming to a MM to other models conforming to other meta-models, or to run them directly. At this stage, in other words, it is possible to generate all the tools needed and/or useful to develop the final application (Figure 5).

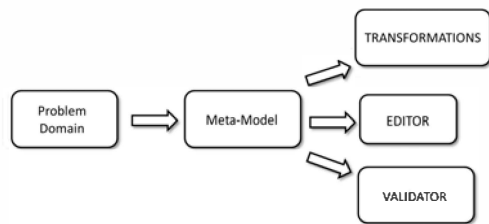


Figure 5 - Meta-modeling

- **Modeling:** by using the editor created in the process of meta-modeling it is possible to define models compliant with MM. Then, by applying transformations and/or parser, it is possible to transform and/or execute these models (Figure 6).

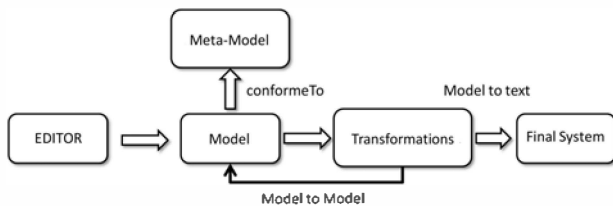


Figure 6 - Modeling

In the development of an ODL-IDE, exploiting the MDE approach implies:

- **Domain of the problem:** the creation of an IDE that manage a particular language.
- **Meta-Model:** the definition of the syntax (abstract and concrete) of a language that allows modeling an IDE for a DSL. For example, this should include the concepts of *grammar*, *syntax rule*, *keyword* and all other entities that allow modeling the final application.
- **Transformations, Editor, Validators:** the creation of application tools, defined starting from the structure of the meta-model, through which instantiate, manipulate and validate the models that will be defined during the modeling.

The definition of meta-modeling tools is the most critical and demanding task of the whole MDD process. The choice of *Eclipse* as a development environment and *EMF* [6] as a modeling framework, has enabled to exploit the power of *Xtext* [7], which already defines a set of tools for meta-

modeling so allowing to focus more to the phase of modeling during the creation of a new DSL.

Xtext

Xtext [7] is an *Open Source Framework* for the creation of *Domain Specific Languages* integrated into *Eclipse*. It enables fast implementation of all aspects related to a complete software infrastructure able to support a DSL. By using *Xtext* it is possible to generate parsers, interpreters, tools for code generation along with editor capable of syntax-highlighting, auto-completion, automatic-build, syntactic and semantic validation. *Xtext* is a DSL to implement new DSL! It provides a set of DSLs and APIs for defining new languages starting from their grammar. In a Model-Driven view, *Xtext* sets the stage for Meta-modeling for creating *Eclipse-based IDE-DSL*. With reference to Figure 7:

- **Grammar Language:** it represents the DSL that *Xtext* provides for the definition of the grammar of the *Xtext Languages* (i.e. the set of languages developed by using *Xtext*).
- **Xtext Generator:** it represents the software component that, starting from the model of grammar written by using the *Grammar Language*, is able to generate the entire software infrastructure to support the new language.
- **Editor:** it is the tool by which *Xtext* allows the definition of the models (concrete-syntax text) written with his *Grammar Language*.
- **Validators:** they are the software tools used by *Xtext* to validate the models written in his *Grammar Language* in order to be correctly parsed by its *Generator*.

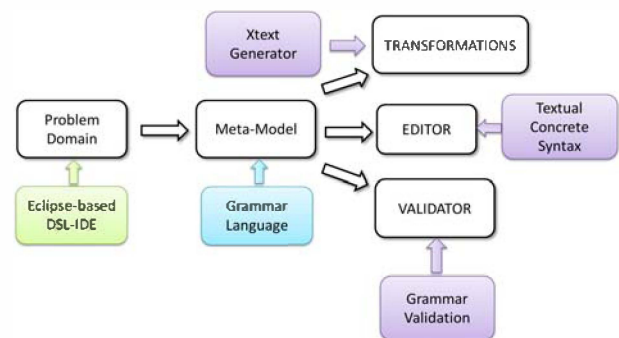


Figure 7 - Xtext in MDE

The creation of a new DSL with *Xtext* passes through the definition of a model that describes the grammar of the language, then using the *Xtext* generator, it is possible to automatically obtain the Java code needed to implement the IDE. The generated code is organized in the form of *Eclipse plug-in* [4][5], which allows the integration of the languages developed with *Xtext* and the *Eclipse Platform* itself.

Anatomy of an Xtext language

The development of a DSL with Xtext consists of 4 main elements (Figure 8):

- **Grammar Model:** it is the model that describes the syntax of the grammar of the DSL to be implemented. The information contained in it have a dual nature. Xtext, starting from a *Grammar Model*, generates two additional models. The *Abstract Syntax* of the language, written as *ecore* model [6], and the specification of the grammar in ANTLR format [9] that allows the generation of the textual Concrete Syntax through the *Parser Generator Framework*. The first one allows the integration of Xtext language run-time with the Eclipse Modeling Framework EMF [6] while the second one allows the creation of a low-level parser that implements the textual concrete syntax specified by the rules in the grammar.
- **Runtime Plug-in:** it is the plug-in that contains the run-time logic of the languages defined with Xtext. It, in addition to containing the generated code, allows easy extensions and customizations. In fact, DSL (and thus the IDE) generated by Xtext represents a recursive application of the Model Driven approach. For each model written with this DSL, it will be associated an in-memory representation made by a class structure (instance of its abstract-syntax). So, manipulations, transformations and validations of this model will be based on this representation and not on its textual version. All this is made possible through the integration of Xtext language in the EMF framework.
- **UI Plug-in:** it contains the code necessary to connect the language runtime logic with the interface of Eclipse. It also provides the ability to easily customize the code obtained from the Xtext Generator.
- **Plug-in Test:** it contains the code necessary for the integration of an Xtext language with the test infrastructure provided by Eclipse (i.e. JUnit [13]). It provides a set of APIs for the definition of test cases to verify the functionality while using a DSL.

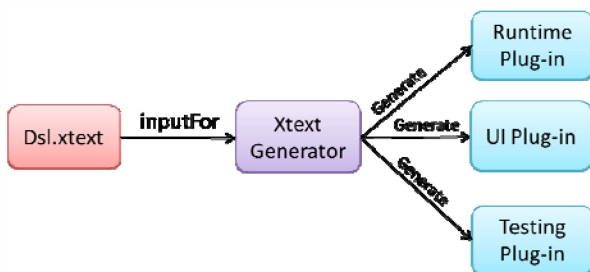


Figure 8 - Xtext Generation Process

The ODL implementation consist of 3 different Xtext languages and each one consists of the elementary components described below.

ODL-IDE Architecture

As shown before, at the base of ODL there are OGDAs and OBCPs. The task of an OGDA is to declare data structures visible to all the *Global OBCPs* associated with its own on-board engine. Each OBCP, in addition, has a section for local declarations, which are visible only within it. So, the need to deal with separate files to edit the sources of the two main constructs, has led to think about ODL as the composition of 3 different languages:

- **Odl.odata:** it represents the ODL section responsible for managing data types, i.e. its *TypeSystem* [14]. It contains all the syntactic constructs that define the declaration of data structures and the statements that use them.
- **Odl.ogda:** it represents the section of ODL related to the definition of the OGDA structure. It contains the syntactic constructs that allow the OGDA declarations and their connection with the rest of the language. *Odl.ogda* includes *Odl.odata* using a subset of the defined constructs.
- **Odl.obcp:** it represents the section of ODL related to the definition of the OBCP structure. It contains the syntactic constructs that allow the OBCP declarations and their connection with the rest of the language. *Odl.obcp* includes *Odl.odata* using a subset of the defined constructs.

Each identified sublanguage is mapped to an Xtext language according to the architecture shown in Figure 9. In this way, it is possible to separately manage the development of the 3 grammars through Xtext. For each one, it has been followed the workflow described in the previous paragraphs and the dependencies of the different sublanguages will result in dependencies of the plug-ins generated from Xtext at the end of the build process. With this choice, it is possible to avoid code duplication and to confine the impact of any changes only in the section where they occur and not on the entire grammar.

The definition of each *OdlSubLanguage* starts by the creation of the relevant *Grammar Model*. This will be composed of a list of rules that, depending on their nature, affect the creation of the abstract syntax and/or the concrete syntax of the language. The size and the complexity of the whole ODL grammar doesn't allow a full analysis in this paper. However, it reports the main aspects to provide an insight on the approach followed to implement such a grammar through Xtext.

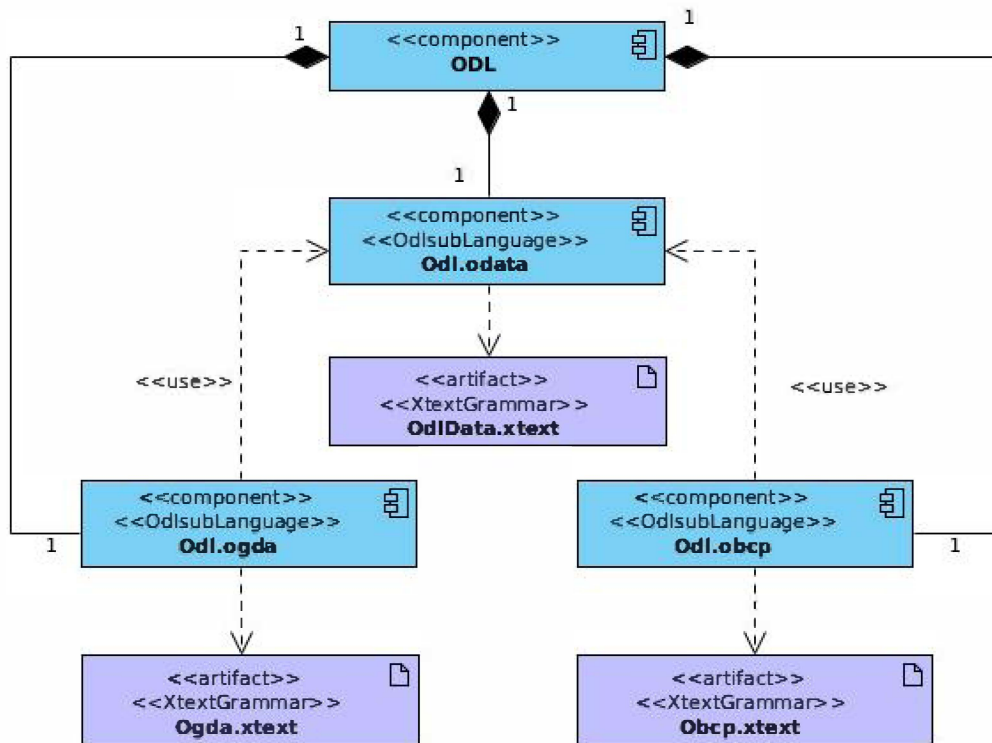


Figure 9 - Odl Grammar Architecture

Odl.odata Grammar Model

The objective of ODL sub-language *odata*, is to declare all the syntactic structures that represent the ODL type-system. This starts from the declaration of a root “Parser Rule” that will act as a logical container for all these structures.

```
/*
 * #odl-grammar odlData model root
 */
odlModelType:
    {odlModelType}
    (typeDeclarations+=typeDeclaration |
    typeExpressions+=typeExpressions |
    typeDataStatements+=TypeStatement)*;
```

The rule *odlModelType* defines the structure of a type model of ODL. This is characterized by the presence of:

- *typeDeclaration*: it is the syntactic rule supertype of all the possible declarations in ODL;
- *typeExpressions*: it is the syntactic rule supertype of all the ODL expressions. Each expression is associated with a specific return type.
- *typeStatement*: it is the syntactic rule supertype of all the statements that involve ODL data structures. Each statement is associated with a specific return type.

odlModelType is not intended to define the structure of any particular source, but only as a container for all the syntactic rules that define its type system. It is not permitted to

declare or use data structures outside of the appropriate sections as defined in OBCP or OGDA. Below, they are shown the syntax rules that define the rule-calls contained in *odlModelType*.

```
/*
 * Supertype rule for ODL declarations
 */
typeDeclaration:
    sev_variable_declaration |
    oct_constant_declaration |
    opt_constant_declaration |
    obrt_declaration |
    flg_declaration | ... | wbf_declaration;
/*
 * Supertype rule for expressions
 */
typeExpressions:
    obrt_expressions | flg_expressions |
    ui8_expressions | ... |
    oes16_logical_expression;
/*
 * Supertype rule for statements
 */
TypeStatement:
    sev_statement | obrt_statement |
    flg_statement | ... | wbf_statement;
```

Given this syntactical hierarchy, providing a new type of ODL means to add declarations, expressions, and statements as additional alternatives to the rules above. Below, as a clarifying example, it is shown the implementation of these rules for the data type *flg*.


```

/*
 * #odl-grammar-00350
 * The flg-constant-identifier metaidentifier
 * represents the alphanumeric strings which
 * are used to identify the FLG constants.
 */
terminal FLG_CONSTANT_IDENTIFIER:
    "flgc_"GENERIC_IDENTIFIER;

/*
 * #odl-grammar-00360
 * The flg-variable-identifier
 * metaidentifier represents
 * the alphanumeric strings which are used to
 * identify the FLG variables.
 */
terminal FLG_VARIABLE_IDENTIFIER:
    "flgv_"GENERIC_IDENTIFIER;

/*
 * #odl-grammar-00370
 * The flg-parameter-identifier
 * metaidentifier represents
 * the alphanumeric strings which are used to
 * identify the FLG on-board parameters.
 */
terminal FLG_PARAMETER_IDENTIFIER:
    "flgp_"GENERIC_IDENTIFIER;

flg_literal_ref:
    value=GENERIC_NUMERIC_LITERAL;

```

```

/*
 * #odl-grammar-00380
 * The flg-constant-declaration
 * metaidentifier
 * defines the construct to be used
 * to declare a FLG constant.
 */
flg_constant_declaration:
    "declare"
    identifier=FLG_CONSTANT_IDENTIFIER
    "set"
    value=GENERIC_NUMERIC_LITERAL
    ";";

;

/*
 * #odl-extension regola che permette di
 * dichiarare cross-reference con costanti
 * flg precedentemente dichiarate
 */
flg_constant_ref:
    ref=[flg_constant_declaration|
        FLG_CONSTANT_IDENTIFIER]
;

/*
 * #odl-grammar-00390
 * The flg-variable-declaration meta-
 * identifier defines the construct to be
 * used to declare a FLG variable.
 */
flg_variable_declaration:
    "declare"
    identifier=FLG_VARIABLE_IDENTIFIER
    "set"
    (literalValue=GENERIC_NUMERIC_LITERAL
    flgConstantRef=[flg_constant_declaration|
        FLG_CONSTANT_IDENTIFIER]
    )
;

```

The terminal rules shown above represent, depending on the type of statement, the syntax that a generic identifier *flg* must possess to be considered valid. It is worth noting as the identifiers of an ODL data type take the form “<datatype> GENERIC IDENTIFIER” so embedding the kind of statement in its identifier. The following ones are the syntax rules that define the structure of the abstract and the concrete syntax for the different types of statements *flg*.

In the Grammar Model of Odl.odata there are similar rules for the definition of expressions and statements specific to each type of ODL data. The adoption of this implementation strategy has enabled the creation of a type-system highly structured and flexible with respect to any changes. By running the Xtext generator on the *OdlData.xtext* grammar, it will be generated the abstract syntax and the textual concrete syntax for this language. Figure 10 shows the structure of the meta-model derived from the grammar *Odl.odata* while an example of a concrete syntax for the declaration of a variable *flg* is the following one:

```
declare flgv_prova set 0;
```

Odl.ogda Grammar Model

The Grammar of the OGDA language model establishes the abstract and concrete syntax relative to each OGDA and uses constructs defined within the *odata* language. The first step to create the OGDA grammar language is to declare the grammar OGDA *odata* extension to reuse the rules defined herein.

```

// ogda grammar as extension of
// odlData
grammar org.xtext.thales.Ogda with
    org.xtext.thales.OdlData

//syntax to model ogda
ogdaModel:
    ogda+=ogda_definition+
;

// rules that defines OGDA structure
ogda_definition :
    "ogda"
    identifier=OGDA_IDENTIFIER
    "is"
    ogdaContent=ogda_content
    "end_ogda" ";";

;
/*
 * #odl-grammar-03410
 * The ogda-identifier meta-identifier
 * represents
 * the alphanumeric strings which are used to
 * identify the OGDAs.
 */
terminal OGDA_IDENTIFIER:
    "ogda_"GENERIC_IDENTIFIER;

```

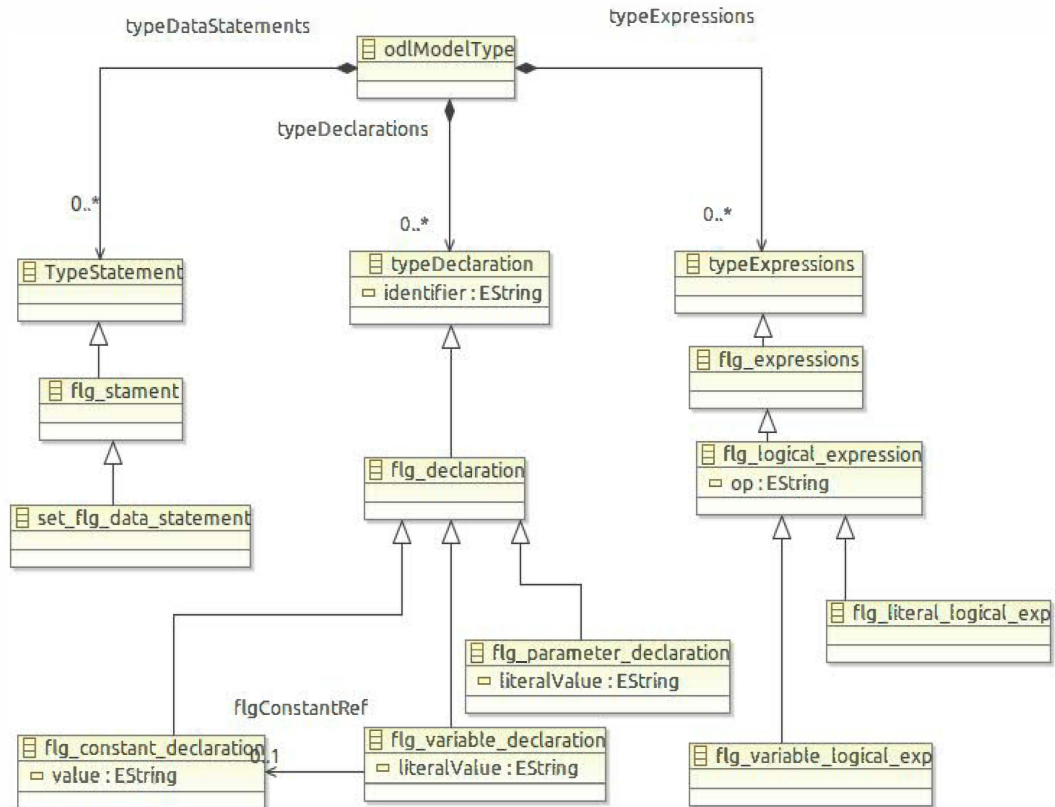


Figure 10 - Odl.odata Meta-model

In a model can be declared from 1 to n OGDA. Each one has a structure defined by the Parser rule *ogda_definition* and it is characterized by an identifier of the terminal with a structure defined by the rule *OGDA_IDENTIFIER*. Each *ogda* is provided with a containment relationship *ogdaContent* of type *ogda_content*. The set of keywords (words in red) and terminal symbols define the usable textual concrete syntax for declaring an OGDA.

```

// ogda structure
ogda_content :
    releseaData=ogda_release_data
    globalAreaSpareSize=
        ogda_global_data_spare_area_size
    delarationSection=
        ogda_declarations_section
;

```

The rule shown above defines the different sections that make up an OGDA. As said for the other rules, each assignment of this type results in the creation, in the corresponding meta-model, a containment relationship with name equal to the left-side of the assignment of the same type and equal to the meta-class defined by the rule used as a right-side of the assignment. The most important section of each OGDA is one of the statements, that is, the *declarationsSection*. This contains the set of terms used by the global OBCP associated with the same OBOE of the OGDA that declares them.

```

/*
 * #odl-grammar-03450
 * The ogda-declarations-section meta-
 * identifier is delimited with the
 * "declarations" and "end_declarations"
 * reserved words and contains either:
 *
 * - The "no_declaration" reserved
 * word which allows to leave the OGDA
 * empty or
 * - A not empty list of declarations.
 */
ogda_declarations_section :
    "declarations"
    ( ogda_declarations+=ogda_declaration+ |
      ogda_emptydeclarations=NO_DECLARATION )
    "end_declarations"
    ";"
;

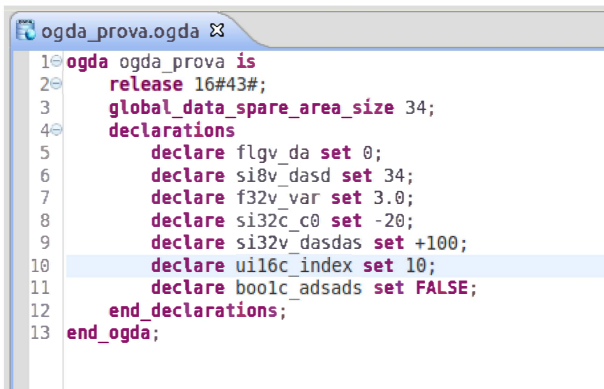
/*
 * #odl-grammar-03440
 * The ogda-declaration meta-identifier
 * represents the set of declarations which
 * is allowed in an OGDA.
 */
ogda_declaration :
    declaration=ogda_types_declaration
;
NO_DECLARATION:
    declaration="no_declaration" ";"
;

```

This section of the grammar is the point of connection between the OGDA and the type system of ODL. The rule *ogda_types_declaration* is defined in *Odl.odata* and collects all data-types in a declarable OGDA.

```
// defines the list of data types list
//in odl declarable in ogda
ogda_types_declaration returns typeDeclaration:
(sev_variable_declaration |
 oct_constant_declaration |
 opt_constant_declaration |
 obrt_constant_declaration |
 obrt_variable_declaration |
 obrt_parameter_declaration |
 flg_constant_declaration |
 flg_variable_declaration |
 flg_parameter_declaration |...|
 bool6_constant_declaration|
 bool6_variable_declaration|
 bool6_parameter_declaration|
 oes_variable_declaration)
;
```

By invoking the Xtext generator it is possible to derive the software infrastructure to support the language. Figure 12 shows the structure of the abstract syntax of an OGDA while Figure 11 shows an example of an OGDA defined through its textual concrete syntax.



```
1 ogda ogda_prova is
2   release 16#43#;
3   global_data_spare_area_size 34;
4   declarations
5     declare flgv_da set 0;
6     declare si8v_dasd set 34;
7     declare f32v_var set 3.0;
8     declare si32c_c0 set -20;
9     declare si32v_dasdas set +100;
10    declare uil6c_index set 10;
11    declare boolc_adsads set FALSE;
12  end_declarations;
13 end_ogda;
```

Figure 11 – Example of OGDA concrete syntax

Odl.obcp Grammar Model

The aim of the *Odl.obcp* grammar is to define the concrete and the abstract syntax to support the definition of an OBCP. As the *Odl.ogda* grammar also the model for the OBCP Grammar begins with the declaration of the new language and the explicit dependencies of the latter by the grammar of types.

```
// including the odlData language
grammar org.xtext.thales.Obc with
  org.xtext.thales.OdlData
```

Each OBCP model is syntactically composed of a set of OBCP.

```
/* obcp root
*/
obcpModel:
  obcp+=obcp_definition+;
/*
 * #odl-grammar-02690
 * The obcp-definition start symbol specifies
 * the structure of an OBCP definition.
 */
obcp_definition :
  "obcp" identifier=OBCP_IDENTIFIER "is"
  content=obcp_content
  "end_obcp" ";"
;
```

Each OBCP is characterized by different sections, each one specific for a particular purpose. The set of sections and their order is defined by the rule *obcp_content* very similar to the rule *ogda_content* in the definition of OGDA. The following are the structure of the declarations section and the main body of an OBCP. These are the main sections and define the set of local declarations and the actual logic of an on-board procedure.

```
//declaration section of OBCP
obcp_declarations_section:
  "declarations"
  ( obcpDeclaration+=obcp_declaration + |
    obcpDeclaration+=obcp_nodeclaration+ )
  "end_declarations" ";"
;
obcp_nodeclaration:
  declaration="no_declaration"
  ","
;
/*
 * #odl-grammar-02570
 * The obcp-declaration meta-identifier
 * represents the set of declarations
 * which is allowed in an
 * OBCP declarations section.
 */
obcp_declaration :
  //connection beetwen odata and obcp grammars
  declaration=obcp_types_declaration
;
```

```
// OBCP main body
obcp_main_body_section:
  "body"
  (obcpSteps+=obcp_step+|
   obcpSteps+=obcp_no_step)
  "end_body" ";"
;
obcp_no_step:
  step="no_step" ";"
;
/*
 * #odl-grammar-02650
 */
obcp_step :
  "begin_step"
  identifier=obcp_step_identifier
  (stepStatement+=(obcp_statement|obcp
    _action)+|
   stepStatement+=obcp_no_statement)
  "end_step" ";"
;
```

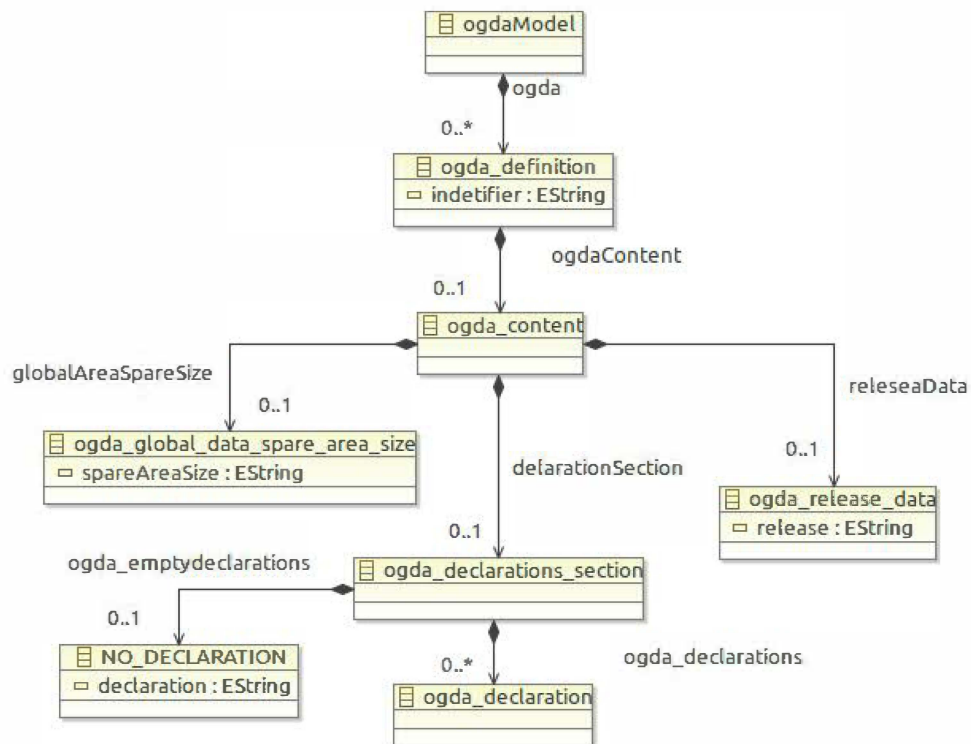


Figure 12 - Odl.ogda Meta-model

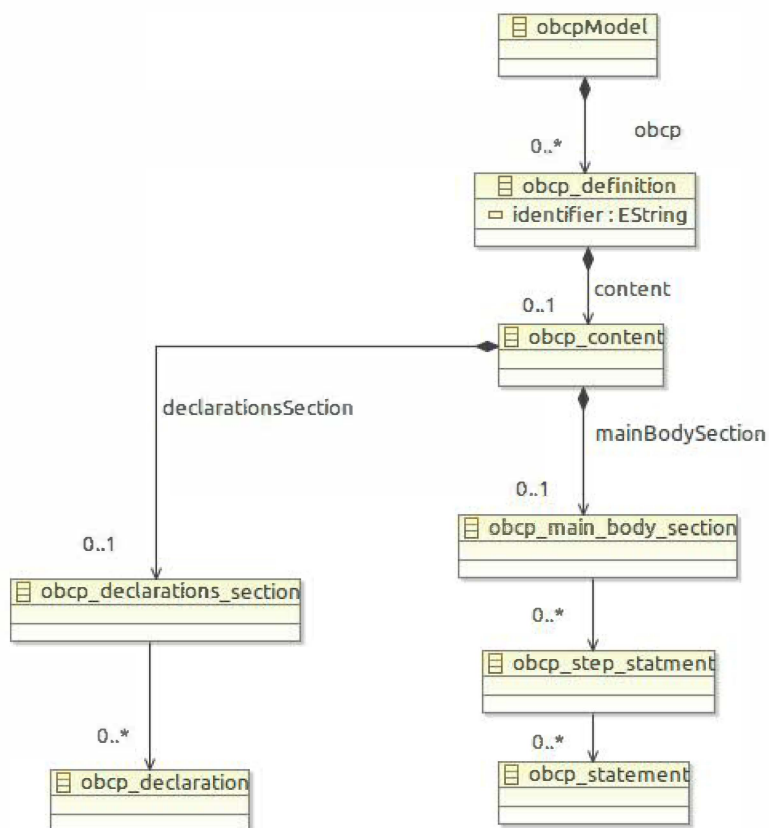
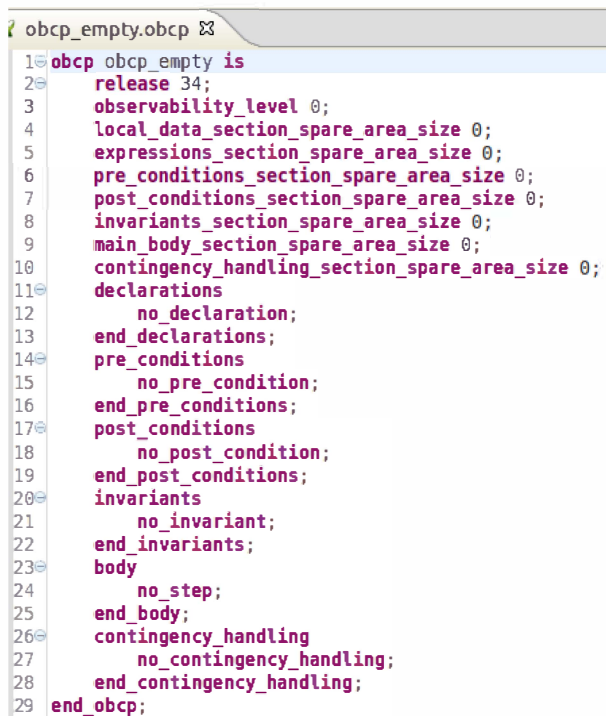


Figure 13 - Odl.obcp Meta-model

The declarations section is identified by the keyword *declarations* and it defines a list of data-type declarations. The syntax of each statement is the one specified by the grammar *Odl.odata*. The types declarable in an OBCP are defined by the rule *obcp_declaration_types* defined precisely in the grammar of the language type-system.

The set of statements is partitioned in steps of computation. These have the objective to collect all the instructions that participate in the achievement of a precise goal. Figure 13 shows an excerpt of the abstract-syntax while Figure 14 shows an example of concrete-syntax of an OBCP, as defined by the Grammar model.



```

1 obcp obcp_empty is
2   release 34;
3   observability_level 0;
4   local_data_section_spare_area_size 0;
5   expressions_section_spare_area_size 0;
6   pre_conditions_section_spare_area_size 0;
7   post_conditions_section_spare_area_size 0;
8   invariants_section_spare_area_size 0;
9   main_body_section_spare_area_size 0;
10  contingency_handling_section_spare_area_size 0;
11  declarations
12    no_declaration;
13  end_declarations;
14  pre_conditions
15    no_pre_condition;
16  end_pre_conditions;
17  post_conditions
18    no_post_condition;
19  end_post_conditions;
20  invariants
21    no_invariant;
22  end_invariants;
23  body
24    no_step;
25  end_body;
26  contingency_handling
27    no_contingency_handling;
28  end_contingency_handling;
29 end_obcp;

```

Figure 14 – Example of OBCP concrete syntax

Runtime plug-in e UI plug-in

Through the execution of Xtext generator on the grammar model defined above, it is possible to generate the *UI* and runtime plug-ins for each language. The dependencies of the different grammars are translated in this way in the dependencies of the various plug-ins. Through these, it is possible to provide to ODL the following tools:

- **Editor:** an editor dedicated to write OBCPs and one for the creation of OGDAs. Through these it is possible to define separately the two main constructs of ODL. The editors generated are perfectly integrated with the Eclipse environment and they are associated with the file **.obcp* and **.ogda* respectively. Each editor allows writing models through the textual concrete syntax defined by their grammar model.
- **Syntax-highlighting:** when creating a model the text is highlighted according to the rules defined by the

grammar.

- **Syntactic validation:** when creating a model (OBCP or OGDAs) the tool generates error messages when violating the syntax rules. The error messages are displayed in-line with respect to the error in the same way that *Eclipse JDT* handles errors for Java programs.
- **Validation semantics:** when creating a template (OBCP or OGDAs) the tool generates error messages when violating the rules of semantic validation. The main feature of this type of validation is that it occurs after the syntactic one and it is not based on the structure of a model but on its content. The error messages will be displayed in-line with respect to the error in the same way as for the syntax errors.
- **Generator:** for every model (OBCP or OGDAs) it could be invoked the generation logic that is able to generate a transformed version of them (e.g. a compiler [12]).
- **Scoping:** it represents the management of the visibility of declarations within the constructs that make up ODL.
- **Semantic model:** the task of the runtime of each language is to instantiate, from the textual representation, a model version of the object. This is achieved through the instantiation of meta-classes and relationships as specified from the abstract-syntax of the language.
- **Linking:** based on the visibility of the elements of a language, cross-reference between the elements of a model or between elements of different models are resolved.
- **Content assist:** when creating a model they are provided suggestions that help the user of the IDE in creating a correct model. The content assist also provides a list of items visible for the resolution of a cross reference.
- **Labeling:** it associates a custom name to each meta-class contained in the abstract-syntax of a language. By default, Xtext uses the name of the meta-class or, if present, the value of the attribute "name" of the meta-class itself.
- **Outline view:** this view has the objective of presenting an overview of the model being edited. This is represented as a tree structure and it is quite similar, for example, to the outline provided by Eclipse for Java programs.

In general, the philosophy of MDE technology is to generate from a model a default implementation for these features together with the extension mechanisms that allow easy customization of the auto-generated code. Xtext implements this strategy using two design patterns:

- **Generation gap:** it manages the separation of the auto-generated code by the custom one so that future

regenerations do not overwrite the custom logic. In Xtext, this is guaranteed by the fact that the classes are self-generated in the source folder "src-gen" of each plug-in while those specific to a language, in the source folder "src" of the same plug-in.

- **Dependency injection:** it manages the decoupling between the components of a software system. Xtext, such as Eclipse, widely use "Google Guice" [10] which is a framework for the management of *dependency injections*. The idea is that a class is not directly responsible for the creation of instances of other classes, but delegates this responsibility to an external framework in order to maximize its level of isolation from the rest of the components. Through this mechanism, it is also possible to provide different implementations of the same interface without modifying the code of the classes that they make use of.

In practice, Xtext generates a mapping from the grammar with the default implementation of the various features. In this way it is possible to provide custom implementations where they are needed, such as for the mechanism of *semantic validation*, *scoping*, *labeling* and *generator*. In this way it is possible to realize a complete IDE by writing only the code for the application to be implemented and therefore reducing the possibility of coding errors and the overhead of development.

Testing

For each language, as well as plug-in runtime and UI, Xtext generates a plug-in dedicated to the unit testing. This is integrated with the infrastructure provided by Eclipse *JUnit* [13]. The idea is to define tests for each language feature independently from the rest of the language itself. During the implementation of ODL IDE they have been provided numerous test cases aimed at the verification of the various implemented features. In order to quickly achieve significant test cases it has been used the testing framework *Xpect* [11]. This represents an Xtext language designed to streamline the creation of procedures for testing, encouraging code reuse and improving readability. The basic idea is to define the procedures for testing models by using such a language. Each *Xpect* model is composed of 3 main blocks:

- **Header:** it contains the configurations needed to correctly launch the test.
- **Oracle:** it describes the result that the testing procedure must produce in order to consider the test passed.
- **SUT:** *System Under Test* is the model, or part of it, that have to be tested.

Each *Xpect* model is written in two *Xtext* languages. The first is the one of the language to be tested, the second one is *Xpect*. The *Header* and the *Oracle* of each test is written with *Xpect* syntax while the SUT is described in the language of the SUT itself.

```
//These sections are considered as comments
// by the ODGA editor

!@@ XPECT_SETUP
//class implementing the testing procedure

org.xtext.xspect.ogda.tests.parser
.ParserTest
//configurazione al di fuori di Eclipse
ResourceSet {
    ThisFile {}
}
//configurazione all'interno di Eclipse
Workspace {
    JavaProject {
        SrcFolder {
            ThisFile {}
        }
    }
}
END_SETUP @@!
```

The header of a *Xpect* model specifies what are the classes implementing the testing methods. In this section are specified, in addition, what are the files involved in the testing. These are divided into two sections "ResourceSet" and "Workspace" which differ according to the test mode. It is important to note that these sections will be ignored by the parser as they are OGDG-wrapped multi-line comments of this language. Thanks to this strategy, *Xpect* greatly increases the readability of a testcase and, at the same time, it presents the oracle, the test and the SUT.

```
!@@
//Test parsing of an empty ogda
// it indicates to Xpect which test to perform
XPECT ast ---
ogdaModel{
    ogda=[
        ogda_definition{
            identifier='ogda_empty'
            ogdaContent=ogda_content{
                releaseData=ogda_release_data{
                    release='16#43#'
                }
            }
            globalAreaSpareSize=
            ogda_global_data_spare_area_size{
                spareAreaSize='34'
            }
        }
        declarationSection=
        ogda_declarations_section{
            ogda_emptydeclarations=
            NO_DECLARATION{
                declaration='no_declaration'
            }
        }
    ]
}
--- @@!
```

The idea of the testing procedure is to parse an OGDG and to verify that its in-memory representation coincides with the desired one. In the section of each oracle model, *Xpect* indicates to the runtime of the latter which method has to be invoked to launch the test procedure. Then, it is possible to specify what is the result expected at the end of the test. The

SUT section shows the model being tested, as reported below.

```
ogda ogda_empty is
  release 16#43#;
  global_data_spare_area_size 34;
  declarations
    no_declaration;
  end declarations;
end_ogda;
```

At this point, running the model as a *JUnit* testcase, it will be applied the following strategy execution:

- *Xpect* runtime invokes the method "ast" class ParserTest passing the string as a parameter along with the oracle model Ogda (SUT).
- Within this method it is run the OGDA parser on the model, generating a serialized representation of the result of this parsing and comparing the two strings.

It generates an error when the expected result is different from the obtained one, otherwise the test is passed. To verify, through *Xpect*, that the parsing of another OGDA model properly involves only the definition of a new model and its oracle, the code of procedure of testing "ast" is preserved in its entirety so maximizing the reuse of code between the different test cases aimed at testing the same functionality of a language. Similar strategies have been adopted for the testing of OBCP and other aspects of ODL IDE.

Eclipse RCP

A *Rich Client Platform* [8] is a development tool that makes it easy to integrate independent software components into a well-defined architecture. It provides all basic services for the construction of complex applications. At this level of development of ODL IDE, in order to create a final product that could be a stand alone solution executable outside of Eclipse, it has been exploited the potential offered by the Eclipse vision of *RCP*. To do this, it is necessary to define three additional components:

- **Application:** like any standard Java program, the Eclipse-based applications have to explicitly define their behavior and an *Eclipse Application* is very similar to the *main()* method. Once the platform has been started it executes a specific application. In Eclipse, this is specified by using a mechanism called *Extension point*. Once invoked, an *Application* has total control over the runtime platform including its shutdown.
- **Product:** this is a configuration file that is located on the upper level of an *Application*. Each product specifies an application to run, the branding and the set of plug-ins needed by the *Application*. By defining a

new product it will be possible to quickly export an *Eclipse RCP* application that, after the start-up, will run the specified Application which will be characterized by the branding and the set of plug-in composing it.

- **Feature:** an *Eclipse Product* is structured as a collection of plug-ins where each one contains the code that implements some functionality. From a logical point of view, the plug-ins are grouped into a product feature. Each feature represents a functionality that could be downloaded and installed separately from the rest of the product itself.

So, finally, the development of ODL IDE has been possible through the definition of an application, a product and a specific feature:

- **Odl.rcp:** Eclipse plug-in that contains the declaration of ODL IDE Application and its logic.
- **Odl.rcp.product:** configuration files contained in the plug-in Odl.rcp representing ODL Eclipse IDE product.
- **Odl.rcp.feature:** feature that contains the plug-in necessary for the execution of the application of ODL IDE.

Through the definition of these additional plug-ins, it has been realized the ODL IDE as a cross platform software solution based on Eclipse but deployable and executable independently from Eclipse itself (Figure 15).

4. CONCLUSIONS

The aim of this work has been to implement an *Integrated Development Environment* (IDE) for *OBCP* & *OGDA Definition Language* (ODL). This is a *Domain Specific Language* developed by *Thales Alenia Space Italy* (TASI) and aimed to writing spacecraft on-board software. ODL is the central element of the overall architecture of the *obAlex* system, TASI implementation of the ESA standard ECSS-E-ST-70-01C. Objective of this system is to introduce within the software, installed on-board a spacecraft, both the presence of structured code through the constructs of *On-Board Command Procedures* (OBCP) and *OBCP Global Data Area* (OGDA) that of a number of execution engines responsible for their implementation. Important peculiarity of this work has been to apply the *Model Driven Engineering* (MDE) methodology. This aims to be the first true generational leap in the production of the software after the introduction of compilers. It emphasizes the construction of the final application by creating models and their subsequent transformations. The implementation of ODL IDE has been obtained through the use of *Xtext* technology and Eclipse RCP. *Xtext* is a DSL for creating DSL, it provides a set of languages and tools to support the creation of new *Domain Specific Language* integrated with the Eclipse environment. In full respect of the MDE philosophy, the application has been obtained by defining models of the latter and then transforming it into the final IDE.

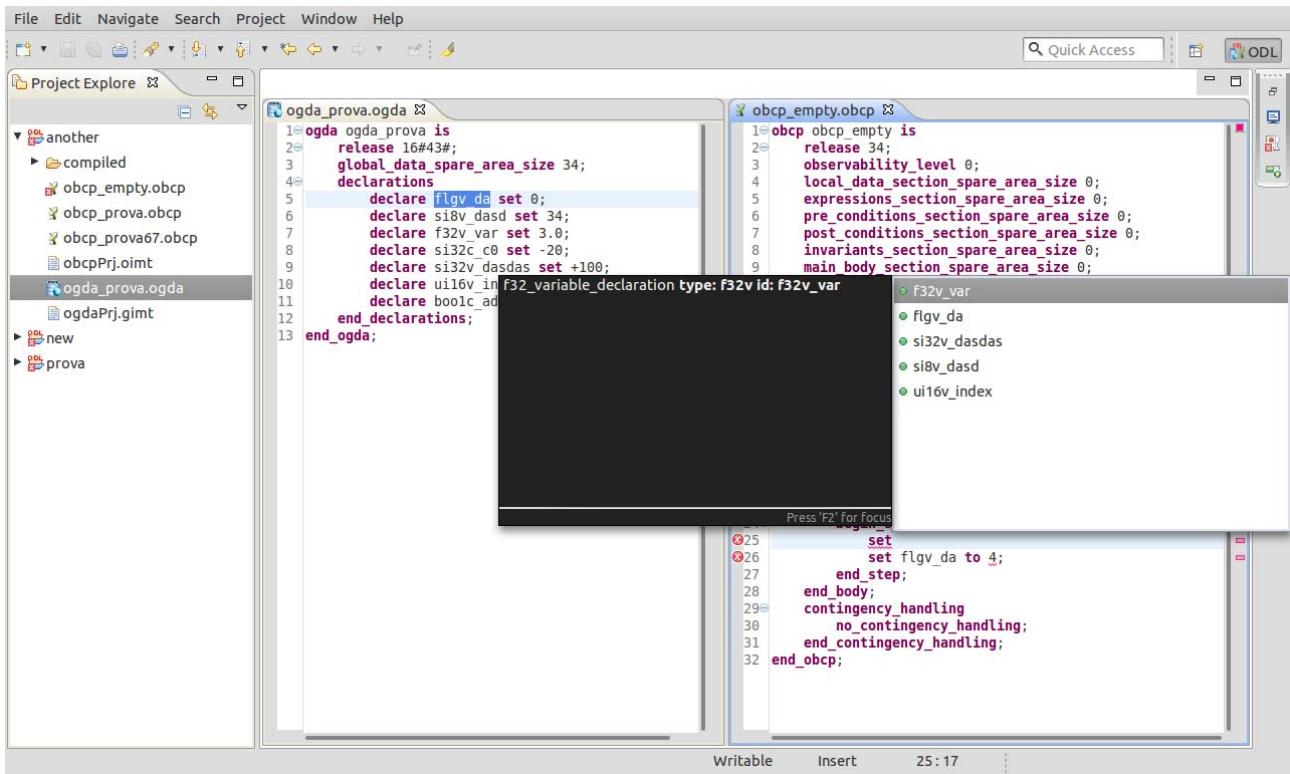


Figure 15 – Screenshot of the developed ODL IDE

More precisely, ODL IDE has been achieved through *Xtext* defining a model of grammar ODL that has been subsequently transformed by the same *Xtext*, in a set of appropriate Eclipse plug-ins. These are installed into the Eclipse IDE to implement the different features. In addition to being developed according to this methodology, it has been used for the production of software written in the language that implements the ODL. Through this methodology, most of the effort during the development of the final system has been addressed to the issues of customization of the generated code and its testing. They have been defined more than 50 test cases designed to verify proper operations of the implemented features.

In conclusion, it has been created a development tool, which presents all the characteristics of a commercial IDE such as Eclipse, but applied to a *Domain Specific Language*, i.e. ODL. The final IDE implements advanced features such as *syntax-highlighting*, *code-completion*, *error-checking*, *auto-filling*, providing the same user experience obtainable by means of a third-level language such as Java. Actually, the images generated by ODL IDE are already fully manageable inside the *obAlex* environment: first on the ground system, for validation purposes, and then they can be uploaded to the on-board OBCP store and executed on the spacecraft. The *obAlex* environment will be made available across all the TASI missions and several roles will be involved in its usage: ASW designers, Avionics subsystem engineers, Ground operators, etc. Moreover, the implementation obtained at the end of this work presents different

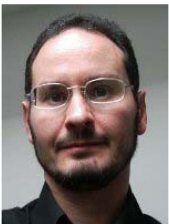
possibilities for future developments. In fact, it is possible, in addition, to define other customizations of the generated code to increase even more the usability and expressiveness of the IDE. Thanks to the development of the latter, such as an Eclipse RCP application, it is also possible to take advantage of the modularity and extensibility of the platform to fully implement the ground system of *obAlex* in a single solution. For this, ODL IDE should not be considered as a point of arrival, but instead as a starting point for the future adoption of MDE techniques in *Spacecraft Software Domain*.

REFERENCES

- [1] Stephen J. Mellor, Anthony N. Clark, Takao Futagami. *Model-Driven Development*. IEEE SOFTWARE, September/October 2003.
- [2] Colin Atkinson, Thomas Kühne. *Model-Driven Development: A Metamodeling Foundation*. IEEE SOFTWARE, September/October 2003.
- [3] Object Management Group OMG *Model Driven Architecture*. Website: <http://www.omg.org/mda/>
- [4] The Eclipse Foundation Eclipse. Website: <https://www.eclipse.org/>
- [5] Eric Clayberg, Dan Rubel *Eclipse Plug-ins (3rd Edition)*. 2008 by Addison-Wesley Professional. Part of the Eclipse Series.

- [6] The Eclipse Foundation. *Eclipse Modeling Framework Project*. Website: <https://www.eclipse.org/modeling/emf/>
- [7] Itemis Xtext. Website: <https://www.eclipse.org/Xtext/index.html>
- [8] Jeff McAffer, Jean-Michel Lemieux, Chris Aniszczyk. *Eclipse Rich Client Platform Second Edition*. Addison-Wesley.
- [9] Terence Parr. *Another Tool for Language Recognition (ANTLR)*. Website: <http://www.antlr.org/>
- [10] Google Inc. *Google Guice*. Website: <https://github.com/google/guice/wiki/Motivation>
- [11] Moritz Eysholdt Xpect. Website: <http://www.xpect-tests.org/>
- [12] Felice Del Forno. *Development of a domain-specific language compiler for spacecraft on-board software*. Master Thesis, Università degli studi dell'Aquila, 2014.
- [13] Kent Beck, Erich Gamma. *JUnit*. Website: <http://junit.org/>
- [14] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press Cambridge, 2002.
- [15] ECSS-E-ST-70-41C in *Telemetry and telecommand packet utilization*. ESA Requirements and Standards Division, Ed. 2014.
- [16] ECSS-E-ST-70-01C in *Spacecraft on-board control procedures*. ESA Requirements and Standards Division, Ed. 2010.

BIOGRAPHY



Luigi Pomante has received the “Laurea” (i.e. BSc+MSc) Degree in Computer Science Engineering from “Politecnico di Milano” (Italy) in 1998 and the Ph.D. Degree in Computer Science Engineering from Politecnico di Milano in 2001. He is actually an Assistant Professor at DEWS (Center of

Excellence - Università degli Studi dell'Aquila, Italy). His activities focus mainly on Electronic Design Automation and Networked Embedded Systems.



Emilio Incerto has received the master degree summa cum laude in computer engineering from “Università degli studi dell'Aquila” in 2014. Now is a Phd student in computer science at Gran Sasso Science Institute (GSSI) L'Aquila. His main interest is in the Software Engineering domain especially in the Model Driven Engineering (MDE) e Model Driven

Development (MDD).



Sante Candia has received the “Laurea” (i.e. BSc+MSc) Degree in Computer Science from “Università degli Studi di Pisa” (Pisa, Italy) in 1986 and the post-degree master from the “School for advanced Studies in Industrial and Applied Mathematics” (SASIAM, Tecnopolis, Valenzano, Italy) in 1990.

From 1991 to 2009 he has worked for Space Software Italia (SSI) and has been involved in the specification, design, implementation, testing of the on-board software of various satellites (Radarsat-2, SICRAL, COSMO-SkyMed, Sentinel-1A). Moreover, he has worked on various payloads on-board software. Since 2009, he works for Thales Alenia Space Italy and is responsible of the Avionics Software department and has been nominated Expert in Avionics Software Architectures. He is also member of the European Space Agency (ESA) working group in charge of updating the ECSS Packet Utilisation Standard.

ACKNOWLEDGEMENTS

This work has been partially supported by the Artemis-JU AIPP 2013 EMC2 (GA 621429) project.

GLOSSARY

ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
ASW	Avionics Services Worldwide
CCSDS	Consultative Committee for Space Data Systems
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
ECSS	European Cooperation for Space Standardization
EMF	Eclipse Modeling Framework
IDE	Integrated Development Environment
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
ML	Modeling Language
MM	Meta Model
MML	Meta-model ML
MOF	Meta Object Facility
OBCP	On Board Control Procedure
OBOE	On-Board Command Procedures Engine
ODL	OBCP&OGDA Definition Language
ODLC	ODL Compiler
ODLE	ODL Editor
OESG	OBCP Engine Static Database Generator
OGDA	OBCPs Global Data Area
PUS	Packet Utilization Standard
RCP	Rich Client Platform
SUT	System Under Test
TASI	Thales Alenia Space Italy
UI	User Interface
UML	Unified Modeling Language