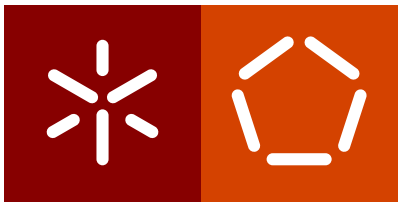


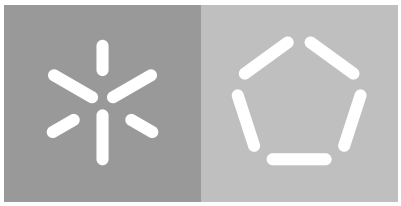
Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Miguel Alexandre Macedo Araújo

Ontology-Driven Metamodeling Towards Hypervisor Design Automation:

Secure Hypervisor Design Environment

February 2018



Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Miguel Alexandre Macedo Araújo

Ontology-Driven Metamodeling Towards Hypervisor Design Automation:

Secure Hypervisor Design Environment

Master dissertation
Master Degree in Embedded Systems

Dissertation supervised by
Professor Doutor Adriano José da Conceição Tavares
Professor Doutor Sandro Emanuel Salgado Pinto

February 2018

ACKNOWLEDGMENTS

To my parents, António Araújo and Filipa Macedo, for all the emotional, financial and educational support during the whole academic period.

To my supervisors, Professor Doctor Adriano Tavares and Professor Doctor Sandro Pinto, who I am grateful for all their time, incentive in pursuing knowledge and very valuable insights on this dissertation.

To Programming Research for providing the license for their static code analysis tool and excellent customer support.

To my colleagues, David Almeida, João Alves, José Lopes, José Martins, Miguel Abreu, Nuno Afonso and Pedro Pereira, for all the help and discussion regarding the work of this dissertation.

To my friends, Leandro Lopes, Ricardo Sousa, Tiago Lopes and Simão Félix, and my brother, Tiago Araújo, for all the much needed decompressing moments.

Thank you very much everyone!

ABSTRACT

Nowadays, the critical embedded software development industry must develop software that adheres to strict safety- and security-related standards. Just as important as the developed software are the development methodologies and tools used in their development. However, certification does require additional efforts on development leading to an increase in both its budget and time-to-market.

The aim of this dissertation is to develop a solution that can ease software developers in achieving compliance with security and safety standards. The solution focuses on the use of automation and modeling techniques via the development of a domain-specific language integrated with semantic technology for the automation and validation of a design flow under a framework named Design Flow Modeling Language.

This dissertation describes the development of the framework through the analysis, design and implementation, focusing on both the language and the ontology perspectives, and its applicability in common development scenarios of embedded systems.

The results show how the developed framework uses semantic rules to enrich the embedded software development process with checks and restrictions to provide safety and security and how automation and external tool integration is made easy with the domain-specific language.

RESUMO

Atualmente, a indústria de desenvolvimento de software embebido crítico deve desenvolver software que adira a rígidos padrões de segurança. Tão importante quanto o software desenvolvido são as metodologias de desenvolvimento e ferramentas utilizadas no seu desenvolvimento. No entanto, a certificação requere esforços adicionais no desenvolvimento que levam ao aumento tanto do custo como do tempo de colocação no mercado.

O objetivo desta dissertação é desenvolver uma solução que facilite aos arquitetos de software alcançar conformidade com os padrões de segurança. A solução foca-se no uso de automação e técnicas de modelação através do desenvolvimento de uma linguagem de domínio específico integrada com tecnologia semântica para a automação e validação do fluxo de *design* sob uma ferramenta chamada Design Flow Modeling Language (linguagem de modelação de fluxo de *design*).

Esta dissertação descreve o desenvolvimento da ferramenta através da análise, conceção e implementação, focando-se em ambas as perspetivas da linguagem e da ontologia, e a sua aplicabilidade em cenários comuns de desenvolvimento de sistemas embebidos.

Os resultados mostram como a ferramenta usa regras semânticas para enriquecer o processo de desenvolvimento com verificações e restrições de forma a providenciar segurança e como a automação e integração de ferramentas externas se torna fácil com a linguagem de domínio específico.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

ANTLR ANother Tool for Language Recognition.

ASIL Automotive Safety Integrity Level.

C

CC Common Criteria.

CCMODE Common Criteria compliant, Modular, Open IT security Development Environment.

CFG Context-free Grammar.

CI Continuous Integration.

D

DFML Design Flow Modeling Language.

DFO Design Flow Ontology.

DL Description Logics.

DSL Domain-Specific Language.

E

EAL Evaluation Assurance Level.

EBNF Extended Backus-Naur Form.

EMF Eclipse Modeling Framework.

G

GPL General Purpose Language.

I

IDE Integrated Development Environment.

IRI Internationalized Resource Identifier.

IT Information Technology.

ITSDO IT Security Development Ontology.

O

OWL Web Ontology Language.

R

RTO Runtime Ontology.

S

SA Static Analysis.

SDLC Software Development Life Cycle.

SIL Safety Integrity Level.

SSDLC Secure Software Development Life Cycle.

SWRL Semantic Web Rule Language.

T

TCL Tool Confidence Level.

TOE Target of Evaluation.

U

UI User Interface.

UML Unified Modeling Language.

W

W3C World Wide Web Consortium.

Y

YACC Yet Another Compiler Compiler.

1

INTRODUCTION

In this chapter is presented the scope of this dissertation, its respective context as well as the objectives and the motivation behind it. In the end, is also presented the organization of this dissertation.

1.1 CONTEXT

Interconnectivity is the word that best describes the current paradigm of technology of our world. People interact ubiquitously with various amounts of machines and devices that are, in turn, interacting with other machines and devices. These devices range from typical consumer electronic items, to vehicles and planes, from manufacturing control and safety systems to critical medical and healthcare systems. The most recent forecast from Gartner (2017) estimated 8.4 billion connected devices to be in use in 2017 and projected 20.4 billion devices by 2020, which begs the question of how security can be assured for all these connected devices.

Looking at critical embedded software development industries like automotive and aeronautics, which require high assurance levels of safety and security, and recalling that the low failure rate of the safety-critical software is achieved due to the mandatory requirement for certification based on standards(Gutgarts and Temin, 2010), it seems that the answer to the security problem might start by the mandatory compliance with safety- and security-related standards. The use of such standards can be seen as a sign of the growing maturity of the embedded software development.

The *Common Criteria (CC)* is an international standard that resulted from the combination of multiple countries *Information Technology (IT)* security evaluation criteria, therefore making it recognized world-wide. Like many standards currently adopted by the safety- and security-related industry, CC concerns not only with the product, referred to in CC as *Target of Evaluation (TOE)*, but also puts emphasis on the whole development process the TOE goes through(ISO, 2009).

In order to stay competitive in an ever-demanding market and assure the safety and security of its products and consumers, manufacturers are required to follow certification

standards, like CC. In a typical embedded systems development process, designers and engineers already have to manage the elevated complexity of the problem and make difficult decisions and trade-offs concerning the different design metrics, which makes the extra effort, time and cost required for certification an obstacle to the adoption and compliance with a standard.

Currently, several approaches exploring different technologies to reduce the certification costs and efforts are continuously being developed and seem to mainly focus on the use of automation and/or modeling techniques to solve this problem. ANSYS (2016) provides a model-based systems engineering solution for the development of safety-related systems that lowers the time and cost of embedded software development and certification. A similar approach is proposed by Slotosch et al. (2012), where they present another model-based approach for tool chain analysis, that reduces tool qualification costs by detecting critical tools and exposing specific qualification requirements for these.

This dissertation is inserted in the embedded systems area and is part of a collective project titled *Ontology-Driven Metamodeling Towards Hypervisor Design Automation*. This project focuses on the development of different subsystems of a hypervisor architecture while promoting their reuse and reconfiguration via a semantically-enriched *Domain-Specific Language (DSL)* infrastructure. Within this project, this dissertation's work focuses on the development of a framework to help the development process of critical embedded software, such as a hypervisor, adhere to general safety and security co-engineering guidelines, by making use of two modeling technologies, DSLs and ontologies.

Regarding DSLs, these are programming or description languages that target a specific problem domain(Bettini, 2016). Their usage in the description or modeling of a specific domain its quite appealing since they promote a simpler and faster development, while providing higher gains in expressiveness, ease of use and productivity when compared to a *General Purpose Language (GPL)*, like Java or C(Mernik et al., 2005).

Although DSLs are great for modeling domains, Walter (2009) explains how enriching a DSL with ontology allows for the specification of additional constraints and semantics, providing better mechanisms for validation and consistency checking.

1.2 Motivation

The opportunity to study and work with ontologies and semantic technology is what piqued the author's interest, since the beginning of this dissertation. These two subjects were completely new to the author and posed an engaging and motivating challenge, requiring research and study of a new topic which always results in great learning experiences. For reasons already presented in the previous section, semantic technology seems to be a very promising technology for dealing with the current problems and a trending

solution for the future as well. Another area of interest is security, especially in software development, which is a very concerning topic today.

1.3 OBJECTIVES

This dissertation's objectives can be seen broadly in two major perspectives. In the first, regarding the semantic technology area, it is intended to develop an ontology for the design flow domain, which will be used not only for modeling but also for providing means of assessing the safety and security properties of the modeled design flow via semantic rules.

In the second perspective, regarding the DSL, it is intended to develop the language itself which, like its ontology counterpart, will be used as a simple description language for modeling design flow definitions.

The ultimate goal of this dissertation is the framework that will provide support for the language in the form of an editor and will be enhanced with reasoning capabilities that comes from its integration with semantic technology. The framework will assist its user while modeling a design flow, using the DSL, and perform its validation against safety- and security-related guidelines, described in the ontology as semantic rules.

1.4 DOCUMENT ORGANIZATION

After a brief contextualization, motivation and objectives, presented in this chapter, Chapter 2 introduces the theoretical foundations and concepts addressed in the development of this work. In the same chapter are given some remarks regarding some previous conducted work related to this dissertation's scope.

In Chapter 3, is presented the overview of the system architecture for the proposed solution and remarks regarding its design, with an explanation of every system component. Then, are explained the details concerning the creation of the design flow ontology with emphasis on its design process and concepts. Lastly, are discussed all the matters regarding the language conception.

In Chapter 4, are presented the important details concerning the *Design Flow Modeling Language (DFML)* framework implementation, in the domain of the DSL and the ontology.

In Chapter 5, are presented some test scenarios and discussed the results of the work developed during this dissertation.

This document finishes with Chapter 6, where are presented the conclusions drawn from the work developed and some considerations for future work in the improvement of the framework.

2

STATE OF THE ART

This chapter presents the fundamental concepts that were addressed during the work developed in this dissertation. Also, are described some of the tools and technologies used to tackle the objectives discussed in the previous chapter. Lastly, is showed some related work in the scope of this dissertation.

2.1 SECURE DESIGN FLOW

In the past, security-related activities were performed only during testing, often resulting in a high number of defects being discovered too late and increasing the cost of their resolution. A more effective approach has been recently introduced under the concept of *Secure Software Development Life Cycle (SSDLC)* which ensures that security-related activities are integrated across the development process (Mougoue, 2016). By making security a continuous concern, earlier detection and resolution of flaws will follow, leading to more secure software with lower costs and, ultimately, lower risks for the organization.

2.1.1 *Software Development Life Cycle*

Software Development Life Cycle (SDLC) consists of a series of steps or phases that model the development and maintenance of software artifacts, with the intent of producing a product that is cost-efficient, effective and of high quality. Overall, the main phases defined in a software development model consists of the analysis phases, where user requirements are analyzed, design phase, where the software is designed, implementation phase, where the software is coded, testing phase, where the software is tested not only for defects but for the compliance with requirements, release phase and, lastly, maintenance of the released software. Software development models are a very important topic in software engineering, which have a huge impact on the SDLC, influencing testing and development time and cost.

Two of the most common SDLC methodologies are the more traditional Waterfall methodology and the more recent Agile methodology. Waterfall defines a more structured model where each phase depends on the outcome of the previous phase and runs sequentially.

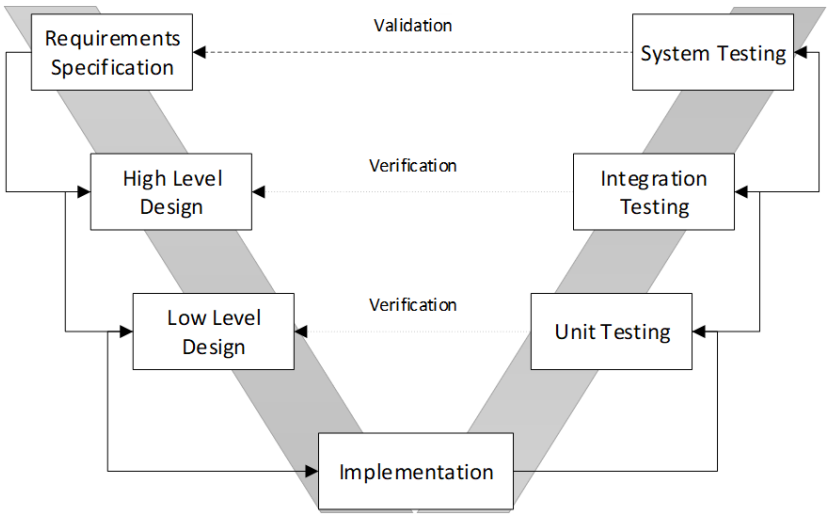


Figure 1.: 'V' model example.

What this model provides in terms of simplicity and straightforwardness it lacks in flexibility as changes in the scope have severe impact in the project's cost, time and quality. The Agile model provides a more interactive approach in which the different phases operate in parallel through cycles, resulting in a working product quickly being available. It promotes the interaction between the customers and developers throughout the project, making easier to react to changes and feedback. Despite providing flexibility, the foundational requirements should remain unchanged to increase the chances of success of the project. That requires some advanced level of maturity and skill in project development to be able to define clear and thorough foundational requirements about the developed product.

Other SDLC models include the V-shaped model, showed in Figure 1, the iterative model, and the spiral model, which are variations of the Waterfall and Agile models, retaining similar advantages and disadvantages.

When comes to embedding security in the SDLC, the Waterfall and Waterfall-based methodologies have the upper hand for two main reasons. First the importance of addressing security early in the analysis phase with the clear definition of security-specific requirements, appropriate risk analysis scheduling and cost, make this stage critical, which is a noted disadvantage in Agile methodologies. Another reason is the fact that secure systems rarely change functionalities, naturally benefiting from a Waterfall rather than an Agile methodology. Despite that, Agile may have a setting when its application regarding security might be advantageous when compared to Waterfall. After the deployment of the product, security must be dealt with as an ongoing process, as new vulnerabilities and new threats appear they are better handled in a more iterative methodology, such as Agile.

2.1.2 *Security and Safety Standards*

A standard is a document that establishes engineering and technical requirements that have been decreed by authority or adopted by consensus, generally covering products, processes, procedures, practices, and methods. Standards can be either *de jure* or *de facto*. A *de jure* standard is an official standard with legal status and is, usually, produced by a national or international organization which has no specific commercial interest or bias. A *de facto* standard is a standard that has achieved a dominant position, by tradition, enforcement, or market dominance.

CC (ISO, 2009) and IEC 61508 (IEC, 2005) are some of the main standards among many currently adopted by the safety- and security-related industry.

The first one, already mentioned in the previous chapter, is according to Brewer (2000), one of the most important information security standards. The CC defines seven *Evaluation Assurance Levels (EALs)* which work as a scale of assurance with EAL 1 being the lowest and EAL 7 the highest.

IEC 61508 is the functional safety standard concerning electrical, electronic and programmable electronic safety-related systems where failure affects people or the environment. The standard explicitly recognizes two types of failures: random hardware failures, which concern specific components to which failure rates can be attributed to, like an electronic component, and systematic failures, which concern failures unique to a given system and its environment, like software, for instance.

IEC 61508 defines the concept of *Safety Integrity Level (SIL)* which is quantitatively defined as probability or frequency of dangerous failures depending on the type of safety function, meaning that higher risk applications require greater robustness to dangerous failures. IEC 61508 defines four SILs with SIL 1 being the least dependable and SIL 4 the most. A SIL is determined based on a combination of quantitative and qualitative factors. Quantitative factors address the frequency of failure and their comparison with a tolerable risk target. Qualitative factors concern with minimization of systematic failures by application of defenses and mechanisms, such as a secure development process and safety life cycle management.

2.1.3 *Secure Coding Standards*

For the development of critical systems, the usage of a coding standard is imperative to improve code safety, security, maintainability, and portability. The use of C/C++ language can be even more troublesome for critical embedded systems as some of its features are not fully specified. Often times, C/C++ features lead to interpretation mistakes where the code behavior differs from what the programmer expects, while other features are implementa-

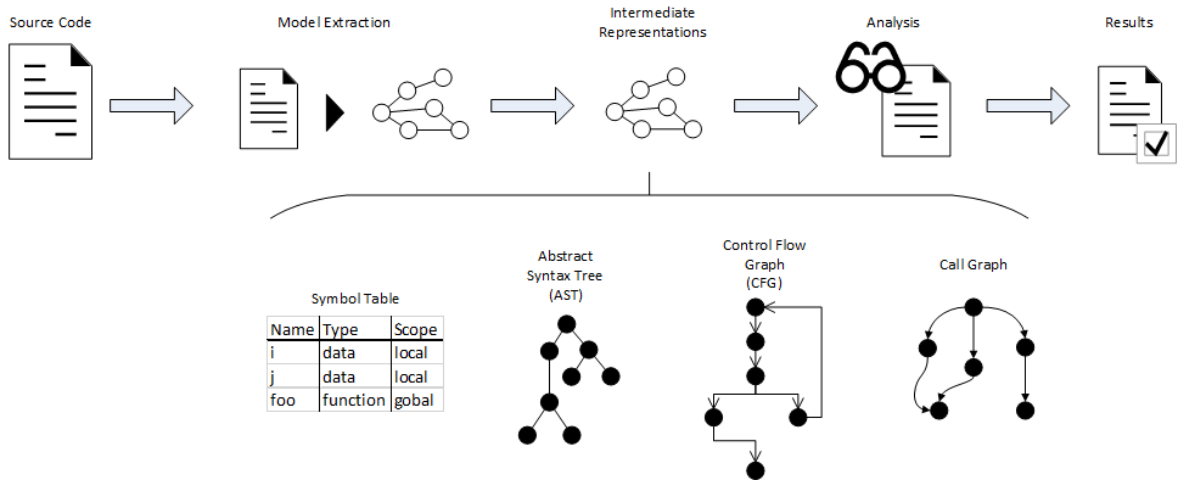


Figure 2.: Static analysis tool architecture.

tion dependent, affecting the portability of the code. Also, some C/C++ statements may be ambiguous since compilers implicitly perform some operations such as type casting.

Due to the various pitfalls of the C++ language, that make it ill-advised for developing critical systems, the main objective of the MISRA C++ guidelines are to define a safer subset of the C++ language suitable for use in safety-related embedded systems (MISRA, 2008). The MISRA guidelines define this safer subset with a series of 228 rules divided across several categories including expressions, namespaces and preprocessing directives, just to name a few. The main purpose of these rules is to restrict the occurrence of known pitfalls and undefined behaviors of the C++ language. Many rules enforce the programmer to be explicit, especially regarding types used in expressions, solving many of the ambiguities of C++, while others address areas like code portability. MISRA also recommends the use of static analysis tools/techniques, whenever possible, not only for validation but also to enforce the compliance with its guidelines.

2.1.4 *Static Analysis*

Static Analysis (SA), or static code analysis, is a technique of program analysis that consists in the examination of source code or object code without executing the program. Automated tools can carry out the static analysis to help programmers and developers find bugs, coding standard violations and security and portability issues, by performing tasks such as control flow and data flow analysis, check for memory leaks, buffer overruns, unreachable code, and other defects. Figure 2 shows a typical architecture of a SA tool. For a deeper understanding regarding algorithms and the current state of the art in the field of SA, the work of Møller and Schwartzbach (2015) is highly suggested. According to PRQA (2016), automation and integration are vital in incorporating static analysis in the develop-

ment process without disturbing development schedules. For instance, the integration of static analysis with source control systems can help significantly with automation as doing so, checking in code in a repository would trigger the analysis so that code defects can be immediately noticed and quickly resolved.

2.1.5 *Continuous Integration*

Continuous Integration (CI) is a software development practice where members of a team integrate their work into a shared repository several times a day (Fowler, 2006). Each integration is verified by an automated build system which includes automated testing tasks to detect integration errors as quickly as possible. The early detection of bugs and code defects introduced with integration allows software developers to quickly resolve them, significantly reducing integration problems.

2.2 Domain-Specific Language (DSL)

Reminding again that DSLs are programming or description languages that target a specific problem domain, in contrast with GPLs which have a generic purpose, is important to emphasize that, while not all programs can be described with a DSL when compared to a GPL, if a DSL covers a problem's domain then this can be solved easier and faster using that DSL instead of any other GPL (Bettini, 2016).

2.2.1 *Xtext and Xtend*

Xtext (Eysholdt and Behrens, 2010) is an Eclipse framework for development of programming languages and DSLs. Besides allowing a simple implementation, Xtext provides support for a fully-featured infrastructure, complete with parser, linker, typechecker and compiler. The framework is also highly customizable, allowing for the creation of a custom eclipse editor tailored to the specific language in development.

Xtend (Efftinge and Zarnekow, 2011) is a dialect of the Java programming language being much more concise, readable and expressive. It provides many additional and improved features like extension methods, lambda expressions, operator overloading and template expressions, just to name a few. Another great feature of Xtend is the fact of it being completely interoperable with Java, as everything written in Xtend compiles into Java source code and interacts seamlessly with any Java library as well.

Xtext is not the only tool available for building languages. Lex (Lesk and Schmidt, 1975) and *Yet Another Compiler Compiler (YACC)* (Johnson, 1975) are another alternative used for building compilers and interpreters. Lex uses a set of regular expressions to generate the

lexical analyzer or lexer, which is responsible for segmenting a sequence of characters from a file into smaller and meaningful units called tokens. These tokens are used by the parser to determine the syntax of a program, which is described using *Context-free Grammar (CFG)* rules. YACC uses a set of these rules to generate a parser.

Another very well known tool for language development is *ANother Tool for Language Recognition (ANTLR)* (Parr and Quong, 1995), which, similarly to YACC, generates a parser from a simple grammar specification and is also widely used to build languages, tools and frameworks, in fact, Xtext makes use of ANTLR to generate the parser used in the developed languages.

Xtext provides a clear advantage over the other three as, just by itself and with a simple grammar specification, is capable of creating all the necessary tools for supporting not just the language but also a complete feature rich *Integrated Development Environment (IDE)* that can be tailored according to the language. This advantage coupled with some already existing experience working with Xtext made it the chosen tool for the development of the DSL in this dissertation.

2.3 KNOWLEDGE ENGINEERING

Gruber (2007) defines ontology, in the scope of computer and information sciences, as a set of representational primitives to model a domain of knowledge or discourse. In the case of *Web Ontology Language (OWL)*, explained in more detailed in the next section, the main primitives are classes, object properties, data properties and individuals.

A class expresses a concept or a set of individuals with something in common. Classes can be declared as either primitive or defined depending if they only describe necessary conditions or necessary and sufficient conditions. A subclass relationship can be used between classes to form hierarchies and they can also be declared as disjoint, if so, an individual that belongs to one class is specifically excluded from belonging to another. An object property establishes relationships between individuals. Like with classes, it is possible to express subclass and disjoint relationships between object properties, as well as restrictions on their cardinality, domain and range. It is also possible to express an inverse relationship between object properties. A data property relates individuals to data values. The same relationships and restrictions enumerated previously for object properties can, likewise, be expressed between data properties. An individual represents an instance of a class. It may belong to several classes simultaneously and is characterized by its object and data properties in addition to the class it belongs to.

When these primitives are used to model a domain of knowledge in an ontology, the information becomes available to be manipulated by a reasoner, which is a program that infers

logical consequences from a list of explicit asserted facts or axioms and, typically, provides automated support for reasoning tasks such as classification, debugging and querying.

2.3.1 *Web Ontology Language*

In 2004, the *World Wide Web Consortium (W3C)* published a document that described OWL as a language to be used for the semantic web. OWL was defined as a language that can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms, in order to form an ontology. The goal of OWL was to provide a richer vocabulary for describing concepts in an ontology, while at the same time making it easier for computers to automatically process such information.

OWL can be separated into three sublanguages or profiles according to different levels of expressiveness. OWLLite is the least expressive and provides hierarchy classification and simple constraints. Then there is OWLDL, named due to its correspondence with *Description Logics (DL)*, which increases in complexity and provides maximum expressiveness while guaranteeing that all conclusions are computable (completeness) and finish in finite time (decidability). For that to be possible, OWLDL has access to all the OWL language constructs but they can only be used under certain conditions that help guarantee completeness and decidability. The third sublanguage is OWLFull and it allows for maximum expressiveness without guaranteeing computational completeness and decidability. For that reason, it is certain that no reasoning software will ever be capable of supporting every feature of this sublanguage.

In 2009, a newer document was published describing OWL 2, an improved version of OWL that included new features like syntactic sugar and allowed for more expressiveness while maintaining a very similar overall structure to OWL 1. This document received a second edition in 2012 being, currently, the most up to date documentation on OWL (W3C OWL Working Group, 2012).

2.3.2 *Semantic Web Rule Language*

Since not all statements can be expressed using OWL constructs, the *Semantic Web Rule Language (SWRL)* was proposed as an extension to the set of OWL axioms, to include Horn-like rules (Horrocks et al., 2004). A rule consists of an antecedent (body) and a consequent (head) and the meaning behind it is if the conditions of the antecedent hold then the conditions of the consequent must also hold. Both the antecedent and consequent can contain zero or more atoms. SWRL rules operate over individuals of the ontology and can be used to directly express or infer new knowledge in the ontology. Figure 3 shows an example of what an SWRL rule looks like and has the following meaning: if exists an individual of

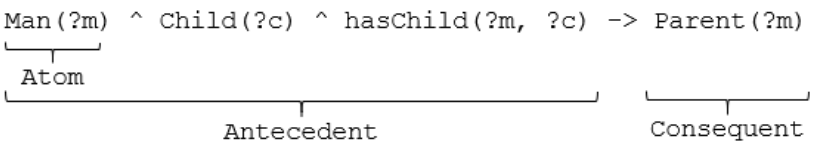


Figure 3.: SWRL rule example.

class Man, characterized by the variable 'm', and exists an individual of class Child, 'c', as well as a relationship hasChild between 'm' and 'c', then, 'm' belongs to class Parent. In other words, this rule translates the knowledge that if a man has a child then he is a parent.

2.3.3 *Protégé*

Protégé, is a Java-based, free open-source ontology development tool created by Stanford University (Noy et al., 2003). It is a very convenient tool for ontology development and evaluation as it allows an ontology developer to quickly prototype and tweak different variants, assess them and experiment with the ontology behavior before committing to its final model. The tool provides some facilities for testing the ontology like consistency checking and the use of DL to query the ontology. It has built-in support for different reasoners and also supports the use of plug-ins, like the SWRL Tab, that allows the formulation of SWRL rules as well as testing their effect on the ontology, and the OWLViz, which provides a visualization of the ontology taxonomy.

2.4 RELATED WORK

In this section are presented and discussed some of the related work in the scope of this dissertation.

2.4.1 *ANSYS SCADE Suite*

The ANSYS company provides market solutions for engineering simulation and computer-aided software development. SCADE Suite is one of its products in the line of embedded systems that uses a model-based approach for critical embedded systems design and development, having been successfully used in many safety-critical aeronautics systems by Airbus, Boeing, Pratt & Whitney, General Electric and many others. The underlying language of SCADE Suite is SCADE, a DSL dedicated to critical systems development which belongs to the family of synchronous languages (Colao et al., 2017).

Overall, the suite provides requirements management, model-based design, verification, certified code generation, and interoperability with other development tools, reducing cer-

tification costs by simplifying application design, automating verification and generating standard-specific documents. Beyond that, the main cause for the reduced certification costs and efforts lies in SCADE Suite KCG Code Generator, which is certified under several safety- and security-related standards. Because its generated code is already compliant with the best practices of embedded code, costly code reviews and low-level verification activities are no longer required (ANSYS, 2016).

2.4.2 *Tool Chain Analysis*

ISO 26262 is a safety standard, based on IEC 61508, specifically addressing the application of embedded systems and software in the automotive industry (ISO, 2011). Besides defining, similarly to IEC 61508, four *Automotive Safety Integrity Levels (ASILs)* levels ranging from ASIL A to ASIL D, the standard requires determination of a *Tool Confidence Level (TCL)* for each software tool that is achieved via a 2-step process.

In (Slotosch et al., 2012) the author explores the possibility of avoiding expensive tool qualification by adding checks and restriction to the development process in order to detect and prevent tool defects. According to ISO 26262 standard, by adding these checks and restrictions the TCL can be lowered so that tool qualification is no longer required. The authors propose a meta model for tool chains as well as a method for its usage in tool chain evaluation and TCL determination. They highlight the main advantages of using a model-based approach: automation, adaptivity, explorability, rework support and enhanced quality assurance due to automated plausibility checks, ultimately resulting in a reduction of tool qualification costs.

2.4.3 *Intelligent Sensors*

In this section of the state of the art, is presented the work developed by Bialas (2011) for the *Common Criteria compliant, Modular, Open IT security Development Environment (CCMODE)* project (CCMODE, 2017), which main goal is to work out a CC-compliant methodology and tools to develop and manage development environments of IT security-enhanced products and systems for the purposes of their future certification.

In earlier work, Bialas (2010) introduced the patterns-based IT security development method and discussed its validation on two different sensors, a medical sensor that remotely monitored patients and a methane detector deployed in a coal mine. In (Bialas, 2011) the author improves the method by using possibilities and advantages offered by the knowledge engineering approach.

He concisely explains the process used for the development of *IT Security Development Ontology (ITSDO)*, an ontology which domain is constituted by the CC compliant IT security

development and the TOE development processes, from the definition of its domain and scope, requirements specification, definition of classes and their hierarchy, definition of class properties and restrictions, creation of instances and, finally, testing and validation.

The author uses the methane detector sensor as an example to drive the validation of the ITSDO ontology application. He uses Protégé to show how the ITSDO is used to elaborate the conceptual security model of the sensor, following the CC-related patterns, including the identification of threats, specification of security objectives, specification of functional security requirements for the security objectives, specification of security functions which implement functional requirements and, finalizing the IT security development process, the specification of security functions and their evaluation evidences.

The author remarks the advantages of the ontological approach by saying that ITSDO, and to some extent ontologies in general, enable common understanding of the terms in the ontology domain, facilitate the reuse and analysis of the domain knowledge and allow the separation of domain knowledge from operation knowledge. Concluding, the author shows that all advantages and possibilities offered by the ontological approach, may be achieved with respect to the IT security development process.

2.5 SUMMARY

The subject study of this dissertation is contained in the secure development field, focusing on the application of DSLs and semantic technology. It is intended to develop an ontology as well as a DSL, targeting the domain of design flow, and a framework that can combine and leverage both artifacts to guide the modeling of design flows in addition to perform validation according to safety- and security-related standards.

The aim of this chapter is to familiarize the reading of this dissertation with the introduction of some technical terms and terminology as well as the presentation of the theoretical background that supports this dissertation. Some of these topics are brought up in the following chapters.

Regarding the related work, the presented solutions show the relevance of this field and serve as proof of the application of DSLs and semantic technology when tackling problems on par with this dissertation's, even though in a more advanced level.

3

SYSTEM ANALYSIS

This chapter presents the fundamental analysis effort behind the conception of the work developed in this dissertation. The system architecture is detailed in addition to the methodologies employed in the development of both the ontology and language.

3.1 SYSTEM ARCHITECTURE

The system being developed was thought and approached, simplistically, as a text editor capable of not only understanding the language being developed for the design flow domain, but also, capable of interfacing with its ontology counterpart. In terms of its broad behavior, the system would parse a file written in the DFML language, translate its data into an ontology equivalent and then execute the described design flow and, if necessary, generate artifacts to support it.

Figure 4 displays an overview of the system, hereafter referred to as the DFML framework, in the form of a block diagram. The *DFML File* is presented as the only input to the framework and it describes a design flow to be executed, using the constructs of the DFML's DSL. The *User Interface* represents the point of interaction between the user and the framework, encompassing the editor and its features. The *Design Flow Ontology* contains the knowledge representation of design flow domain concepts and also serves as the base ontology from which the *Runtime Ontology* is derived. The *Runtime Ontology* is another form of representation of the design flow described in the *DFML File*. This representation allows for the data to be manipulated with the *Reasoner*, which is used to perform some operations like retrieve information from the ontology, perform consistency checks and infer new knowledge. The *Ontology Manager*, like the name suggests, is a subsystem to handle ontologies operations like their creation and modification. The *InferenceEngine* is the core of the DFML framework and its behavior can be described in two major steps. In the first step, it begins by extracting the design flow model from the DFML file. After its validation, it starts parsing the model and mapping the DSL constructs into ontology axioms that are used in the creation of the *Runtime Ontology*, finishing with the consistency check of the produced *Runtime Ontology*. In the second step, it starts the execution of the chain of

activities that make the design flow by using the *Reasoner* to decide which activity is executed until the last one, following the specified chain. The *Build System* block represents a set of outputs of the framework which includes artifacts used to manage the build process of a software project. The *Design Flow Execution* block simply represents the automated execution of tasks carried out by the *InferenceEngine*.

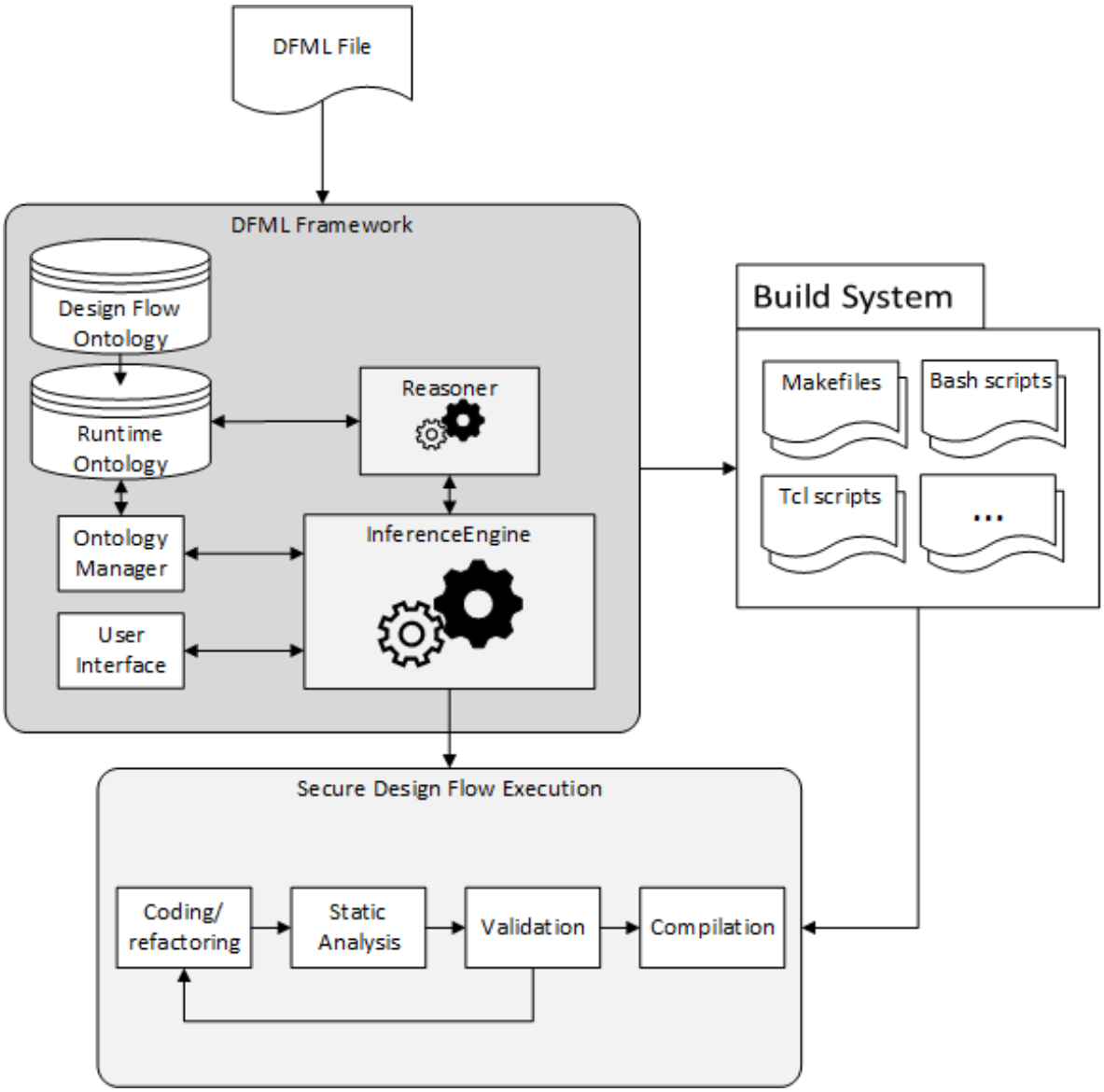


Figure 4.: DFML framework architecture.

Table 1.: The DFO competency questions.

What is the domain?	Design flow description
What can the ontology be used for?	Plan a design flow Assess the safety and security of a design flow Understanding the different phases of development
What questions should the ontology answer?	Is the design flow secure? Is the design flow complete? Does the design flow use static analysis? Does the design flow use continuous integration? Does the design flow have errors? What is the status of the activities? Do activities have errors? What files are used in an activity? What tools are used in an activity?
Who will use and maintain it?	It will be used by software designers/developers and general users It will be maintained by software development domain experts

3.2 DESIGN FLOW ONTOLOGY

For the development of the *Design Flow Ontology (DFO)*, was considered the perspective provided by Noy and McGuinness (2001). According to them, the following are fundamental rules to be considered during ontology design:

- There is no one correct way to model a domain. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.
- Ontology development is necessarily an iterative process.
- Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.

They also describe a step-by-step ontology development process and address some complex issues that arise during ontology design.

3.2.1 *Competency Questions*

The first step in ontology development consists in determining the domain and scope of the ontology and for that the authors suggest the use of competency questions as a mechanism to help defining them. Table 1 contains the competency questions formulated during the analysis of the DFO. The ontology will be used mainly for design flow description and to help its users planning and assessing a design flow with regards to their safety and security properties.

3.2.2 *Classes*

Following the formulation of the competency questions, was done the enumeration of terms to be included in the ontology. Since a design flow involves a chain of activities and each activity generally uses tools and/or files, the terms *activity*, *tool* and *file* were immediately selected. When working with tools or files the terms like *option*, *name*, *location* are often used as properties of both these concepts. Also, since the design flow encompasses a chain of activities, the ontology would need terms to specify the order of such activities and so, the terms *first*, *last*, *next* and *previous* were also included. Regarding the design flow evaluation, since the goal is to guarantee its security, the terms *evaluation*, *goal* and *status* were considered. When doing some categorization, more generic and encompassing terms appeared like *resource* and *process*. Other initially contemplated terms were discarded during the successive iterations of the ontology for either being to generic or overlapping with other terms. That was the case with terms like *Input* and *Output*, for instance.

After the enumeration of the useful terms to be part of the ontology came perhaps the most important part of the ontology development, the definition of classes and their hierarchy. This step helps to differentiate between terms that describe an object with independent existence, thus being used to specify a class, and terms that describe these objects, that are used to specify properties of classes, which should be addressed in a following step, although as the authors put it, both are closely intertwined. The definition of the classes and their hierarchy resulted in the taxonomy displayed in Figure 5. The *owl:Thing* class is a predefined class in any OWL ontology. It represents, as the name suggests, things and every defined class belongs to this class. If a class does not belong to the *owl:Thing* class, that indicates that something is wrong with that class as it cannot be classified as a thing, being instead classified as a subclass of *owl:Nothing*. Next comes the first set of subclasses that start to filter the set of all individuals into something more specific. The *Entity* class represents the set of units or building blocks that are part of any design flow and has two subclasses: *Activity* and *Resource*. *Activity* defines the basic structural unit of a design flow, which can be chained and consume resources, being also refined into five subclasses according to different stages of the design flow. *Resource* establishes the resources used in the execution of an activity and is further specialized with the classes *File* and *Tool*. Although Figure 5 represents the final version of the taxonomy, the *Goal* class was one of the last to be included as it simply exists for the sole purpose of providing some organization and stands for the goal of a process. This concept will become clearly understood when the created individuals or instances are explained. The class *Process*, much like the previous class, serves the purpose of providing some organizational structure to the ontology, being further refined into the *DesignFlowEvaluation* class which represents the evaluation process of the design flow with regards to its security properties. The last generic class is *Prop-*

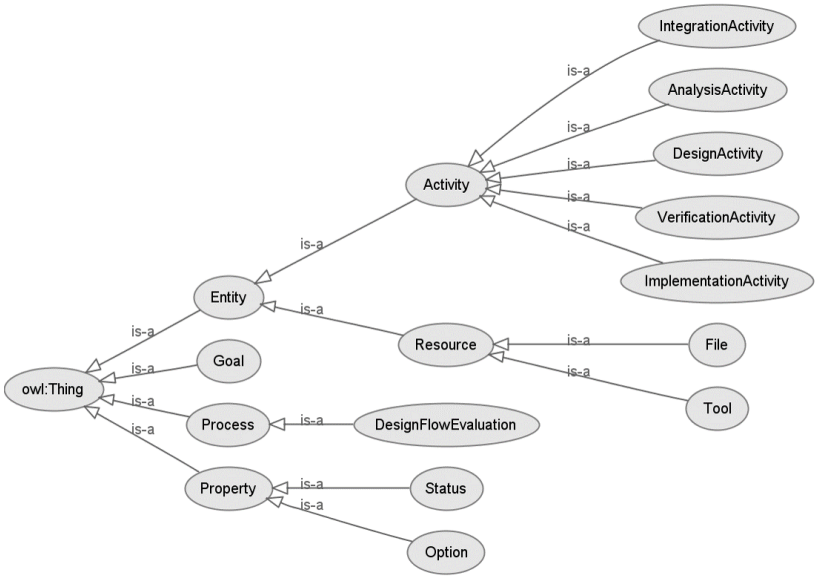


Figure 5.: The DFO taxonomy.

erty and encompasses properties of the *Entity* class. It is specialized in the class *Status*, which establishes the status property of an activity, and *Option*, which defines a property of resources, for instance, the name of a file or the directory of a tool.

3.2.3 *Properties*

As the classes alone are not able to answer the competency questions, the next step in the further refinement of the ontology focuses on the definition of classes' properties.

Table 2 contains the object properties, represented by the '*', and the data properties of the DFO. The third column represents the type of a property which can be asserted, when it is explicitly assigned, or inferred, in this case, from an SWRL rule. The last column indicates the range of the property which, in the case of object properties is always going to be another class, and in the case of data properties is a data type where '(int)' represents an integer number, '(string)' represents simply a string and '(boolean)' represents a true/false or yes/no condition.

3.2.4 *Individuals*

The last step is the creation and parametrization of individuals which represent an instance of the class they belong to. The DFO contains six individuals in total instantiated across three different classes. The *DesignFlowEvaluation* class contains an instance that materializes the required evaluation named *Evaluation*. For the *Goal* class an individual named *Security*

Table 2.: The DFO object and data properties.

Class	Property	Type	Range
Activity	hasNext*	Asserted	Activity
	hasPrevious*	Asserted	Activity
	hasFile*	Asserted	File
	hasTool*	Asserted	Tool
	hasStatus*	Asserted	Status
	isActivityOf*	Asserted	DesignFlowEvaluation
	hasError	Inferred	(string)
	isFirst	Asserted	(boolean)
	isLast	Asserted	(boolean)
Goal	isGoalOf*	Asserted	Process
Option	isOptionOf*	Asserted	Resource
	isRequired	Asserted	(boolean)
	optionName	Asserted	(string)
	optionOrder	Asserted	(int)
	optionValue	Asserted	(string)
Process	hasGoal*	Asserted	Goal
DesignFlowEvaluation	hasActivity*	Asserted	Activity
	hasError	Inferred	(string)
	isComplete	Inferred	(boolean)
	isSecure	Inferred	(boolean)
	usesCI	Inferred	(boolean)
	usesSA	Inferred	(boolean)
Resource	hasOption*	Asserted	Option
	isResourceOf*	Asserted	Activity
File	isFileOf*	Asserted	Activity
	fileExtension	Asserted	(string)
	fileLocation	Asserted	(string)
	fileName	Asserted	(string)
Tool	isToolOf*	Asserted	Activity
	toolDomain	Asserted	(string)
	toolLocation	Asserted	(string)
	toolName	Asserted	(string)
Status	isStatusOf*	Asserted	Activity

was created to represent the goal being achieved by the *Evaluation* individual. The *Status* class contains four individuals that represent the possible state of an *Activity* and these are *NotFinished*, *NotStarted*, *Finished* and *Started*.

In Figure 6 is a *Unified Modeling Language (UML)* class diagram representation of part of the ontology. The classes contain its data properties, with the inferred ones being suffixed

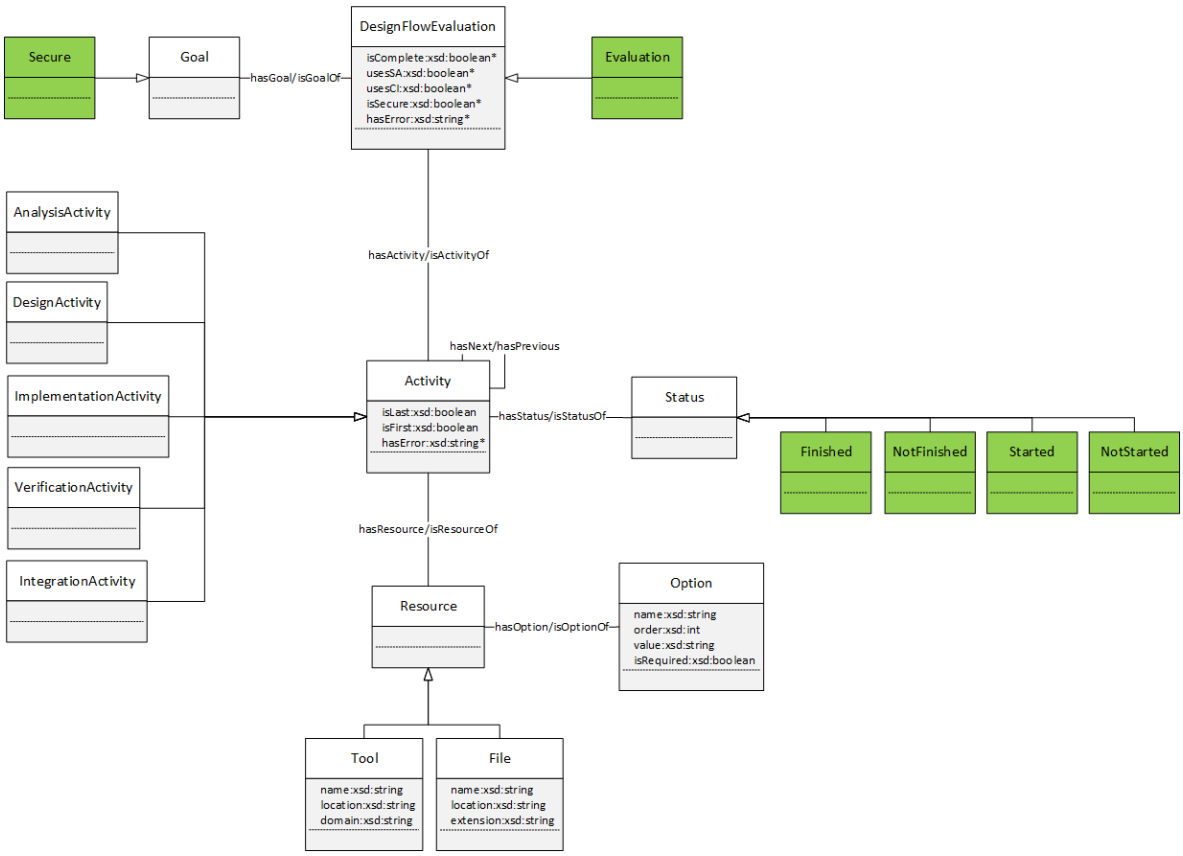


Figure 6.: The DFO class diagram.

with the '*', and object properties and its inverses represented by connections between classes. The six individuals are highlighted in green.

3.2.5 *SWRL Rules*

The ontology by itself is still not able to answer some of the competency questions concerning the design flow validation. For that was necessary to develop SWRL rules that extend the flexibility of the ontology making it capable of providing answers to the remaining questions.

Table 3 contains the seven semantic rules that were developed for the DFO. To answer the question about the security of a design flow was decided that a design flow is considered secure if it is complete, meaning to have at least one activity in every one of its phases, and includes static analysis and continuous integration tools. Rule-1 was therefore formulated to translate that concern and is responsible for the evaluation of the security of the design flow. Rule-1 by itself cannot answer the question and is supported by three other rules. Rule-2 is used to infer whether a design flow is complete, while Rule-3 and Rule-4

are used to infer the use of static analysis in the verification phase and continuous integration in the integration phase, respectively. This type of interaction achieved between rules demonstrates the flexibility provided by SWRL rules and how they are able to provide more complex inferences over the ontology. Rule-5 and Rule-6 are both used to report activities' errors during the execution of the design flow. Rule-5 accounts for unfinished activities and Rule-6 particularizes the case of executing an activity before its precedent activities are finished. Finally, Rule-7 notifies when the desired goal of security for a design flow is not met.

3.2.6 *Reasoner Selection*

Regarding the selection of a reasoner, a comparison of the different available reasoners done by Bock et al. (2008) was taken into account. Pellet was selected for showing good performance at load and classification time, especially for small and simple ontologies like DFO. Furthermore, it offers support for SWRL, on which DFO greatly depends, and an interface via OWLAPI as well as being available as a plug-in for Protégé which facilitates testing.

3.3 DESIGN FLOW MODELING LANGUAGE

Regarding the design of the language, the author's experience from past work with a DSL and Xtext/Xtend Bettini (2016) was taken into account. The aim of the DFML is to allow the textual description of a design flow by using the very own syntax of the domain and additionally to some extent, work as an abstraction layer for ontology management which relieves its user from the need to understand and interact with ontologies directly.

3.3.1 *Language Overview*

To use the DFML language correctly in a file, this latter is required to have the '.dfml' extension. Since the aim is to describe a design flow, a '.dfml' file was initially thought out as having a first section for the design flow description followed by a section for a more detailed description of the activities and ending with a section for describing resources, either files or tools. After several development iterations, a feature was introduced to allow the description of a resource template promoting the reuse in various design flows. With it was necessary a file, with the same '.dfml' extension, but with a different type of content that would provide the needed physical separation between a file that described a design flow and the one that described a resource template.

Table 3.: The DFO SWRL rules.

isComplete(Evaluation, true) ^usesSA(Evaluation, true) ^usesCI(Evaluation, true) **-> isSecure(Evaluation, true)**	Rule-1
AnalysisActivity(?a) ^DesignActivity(?d) ^ImplementationActivity(?i) ^VerificationActivity(?v) ^IntegrationActivity(?int) ^hasActivity(Evaluation, ?a) ^hasActivity(Evaluation, ?d) ^hasActivity(Evaluation, ?i) ^hasActivity(Evaluation, ?v) ^hasActivity(Evaluation, ?int) **-> isComplete(Evaluation, true)**	Rule-2
VerificationActivity(?a) ^hasActivity(Evaluation, ?a) ^hasTool(?a, ?t) ^toolDomain(?t, "StaticAnalysis") **-> usesSA(Evaluation, true)**	Rule-3
IntegrationActivity(?a) ^hasActivity(Evaluation, ?a) ^hasTool(?a, ?t) ^toolDomain(?t, "ContinuousIntegration") **-> usesCI(Evaluation, true)**	Rule-4
hasStatus(?a, NotFinished) **-> hasError(?a, "Activity failed its execution and/or did not finished.")**	Rule-5
hasNext(?a, ?b) ^hasStatus(?b, Finished) ^hasStatus(?a, NotFinished) **-> hasError(?b, "Activity finished before its previous activities.")**	Rule-6
isSecure(Evaluation, false) ^hasGoal(Evaluation, Security) **-> hasError(Evaluation, "The design flow was declared secure but is insecure.")**	Rule-7

Figure 7 shows a representation of DFML in a UML class diagram, with the right side grouping the template description content and the left side grouping the design flow description.

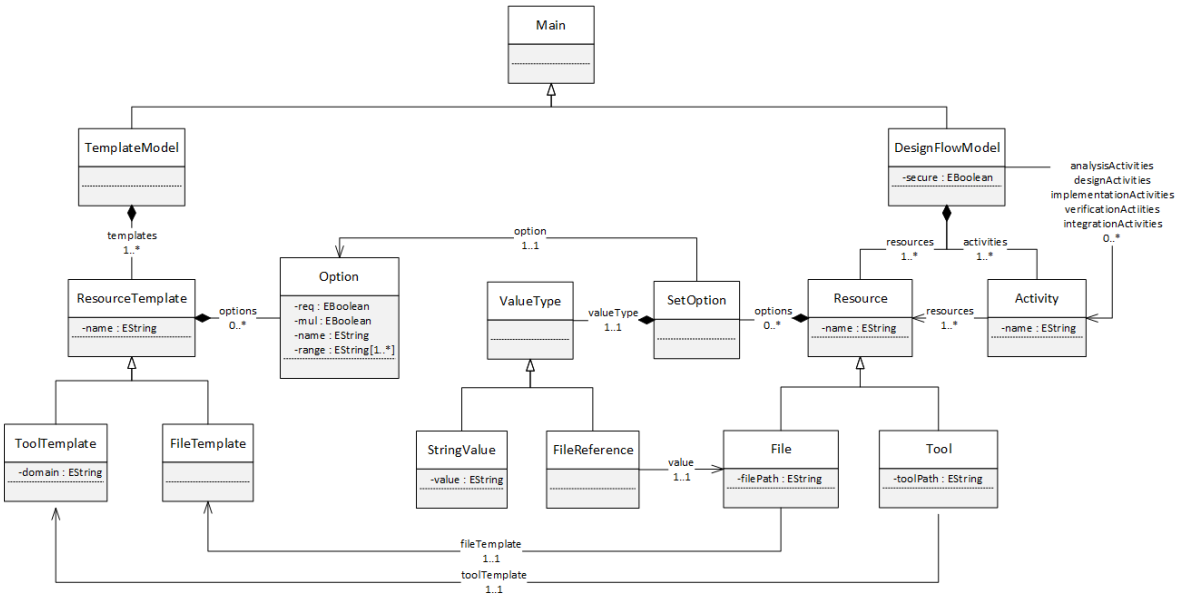


Figure 7.: The DFML class diagram.

3.3.2 *Rules*

In order for a DFML program to be considered valid, it has to follow a set of rules as failing to do so would raise errors in the described model.

- A design flow must have at least one activity and one resource;
- A design flow must not have duplicate activities;
- An activity must have one and only one tool associated to it;
- The design flow must use an activity;
- An activity must use a resource;
- A required option must be used;
- The range of an option must be respected;
- A tool can only reference files if both, the tool and referenced files, belong to the same activity;
- Activities, tools, files and options must have different names;

3.3.3 *Keywords*

The available keywords provided by the DFML are represented in Table 4.

Table 4.: The DFML keywords.

Keyword	**Description**
Activity	Defines an activity
Analysis	Defines the set of activities for the analysis phase
Design	Defines the set of activities for the design phase
domain	Declares the tool domain
File	Defines a file
filename	Declares the file name
FileTemplate	Defines a file template
Implementation	Defines the set of activities for the implementation phase
Integration	Defines the set of activities for the integration phase
multiple	Declares an option as multiple (used multiple times with different values)
option	Defines an option
range	Declares the range of values an option can take
required	Declares an option as required
SecureDesignFlow	Declares a secure design flow
Tool	Defines a tool
toolpath	Declares the tool location
ToolTemplate	Defines a tool template
Verification	Defines the set of activities for the verification phase

3.4 SUMMARY

This chapter started with the presentation and analysis of the system architecture for the DFML framework with the detailed view of every major system component. Then followed the description of all the analysis process for the ontology and language. On the ontology side, was briefly described the methodology used followed by its execution, starting with the elicitation of concepts and ending with the SWRL rules. On the language side, was given an overview with the help of an UML class diagram followed by a set of rules as well as a list of keywords to be part of the language that are close to the design flow domain.

4

IMPLEMENTATION

This chapter builds on the analysis efforts from the previous chapter, presenting the implementation process of both the ontology and language in addition to the DFML framework. The ontology is built with following the class hierarchy devised in the analysis chapter, the data and object properties are realized and parametrized, its individuals are instantiated and the SWRL rules are implemented. The language grammar is specified using *Extended Backus-Naur Form (EBNF)* notation and validators are implemented to enforce the previously defined rules of the language. Lastly, is detailed the framework development, emphasizing some of the major classes responsible for the semantic integration with the language and some of the editor customizations.

4.1 DESIGN FLOW ONTOLOGY

After the analysis exercise, the implementation of the DFO was a very straightforward process. The ontology for this dissertation was implemented in OWL2 using the Protégé 5 ontology editor. First were created the classes, then were declared the object and data properties followed by the instantiation of individuals and, finally, the creation of semantic rules.

4.1.1 *Classes and Individuals*

Figure 8 is a screen shot of the OntoGraf tab in Protégé where are displayed the DFO classes, represented by the yellow circle, and individuals, represented by the purple diamond.

4.1.2 *Properties*

Declaration of object properties in Protégé simply involve the attribution of the range and domain, characteristics which allow for a more enriched meaning of a property, and inverse properties that contribute to a more complete ontology. Figure 9 contains all the object

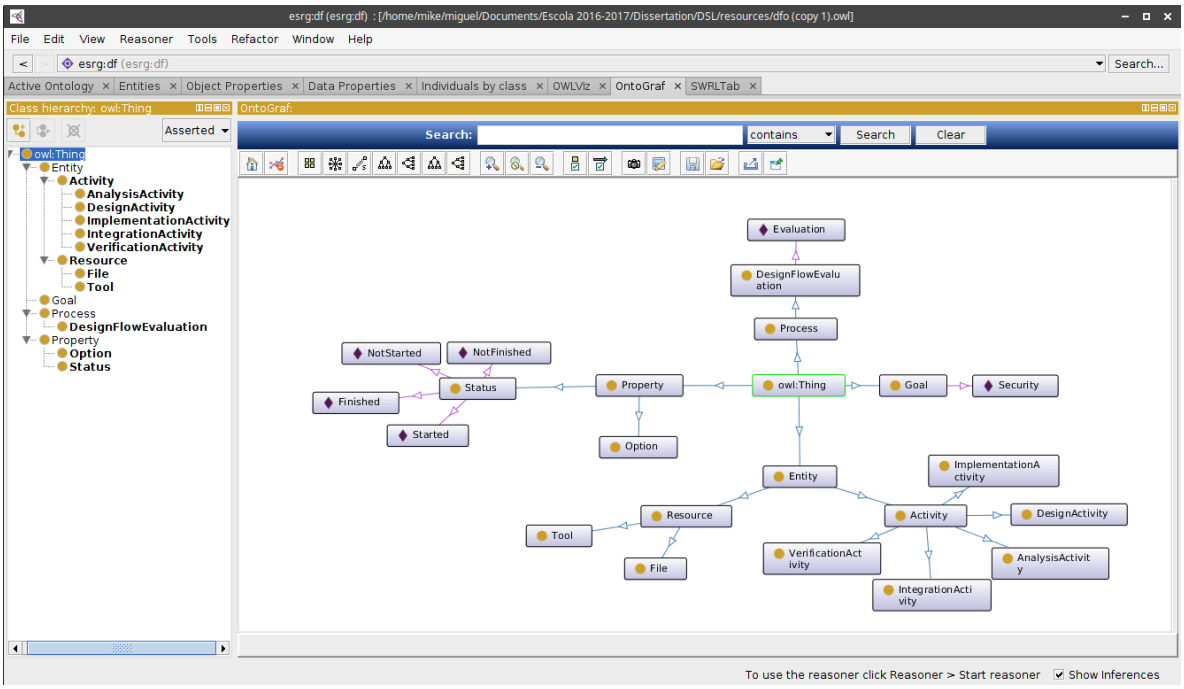


Figure 8.: Class hierarchy and individuals.

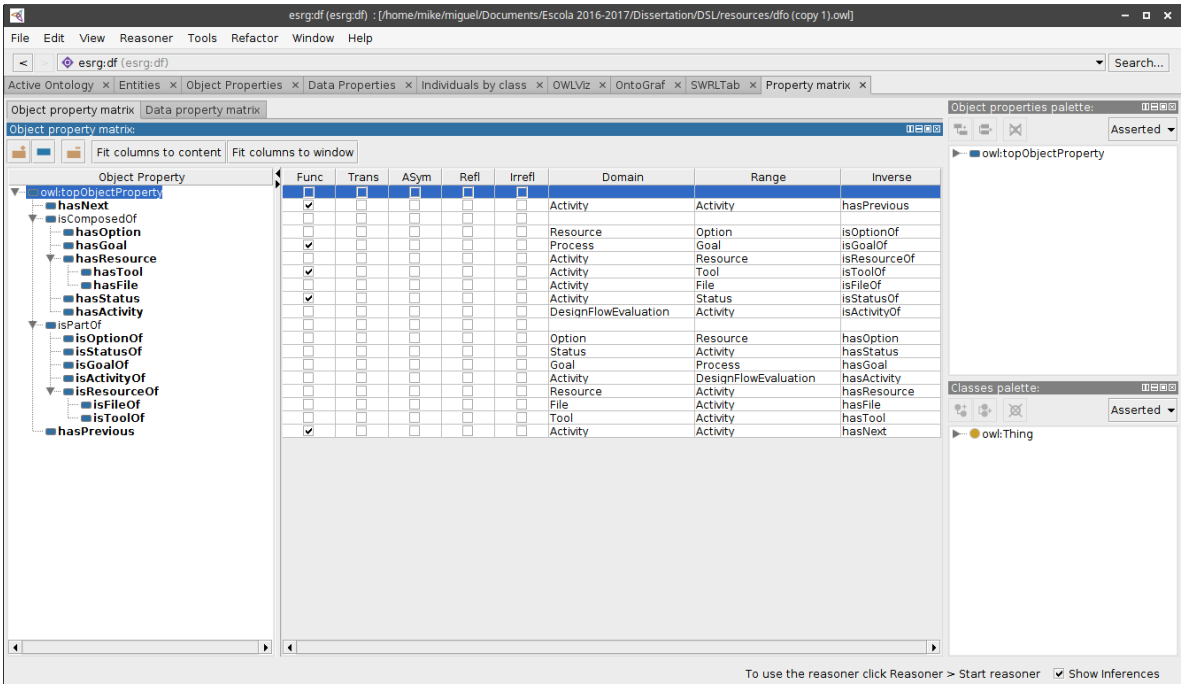


Figure 9.: Object properties matrix.

properties declared in the ontology. Some of these properties, like *hasTool* for instance, were declared as functional properties. If a property is functional, for a given individual, there can be at most one individual that is related to the individual via the property. This means if

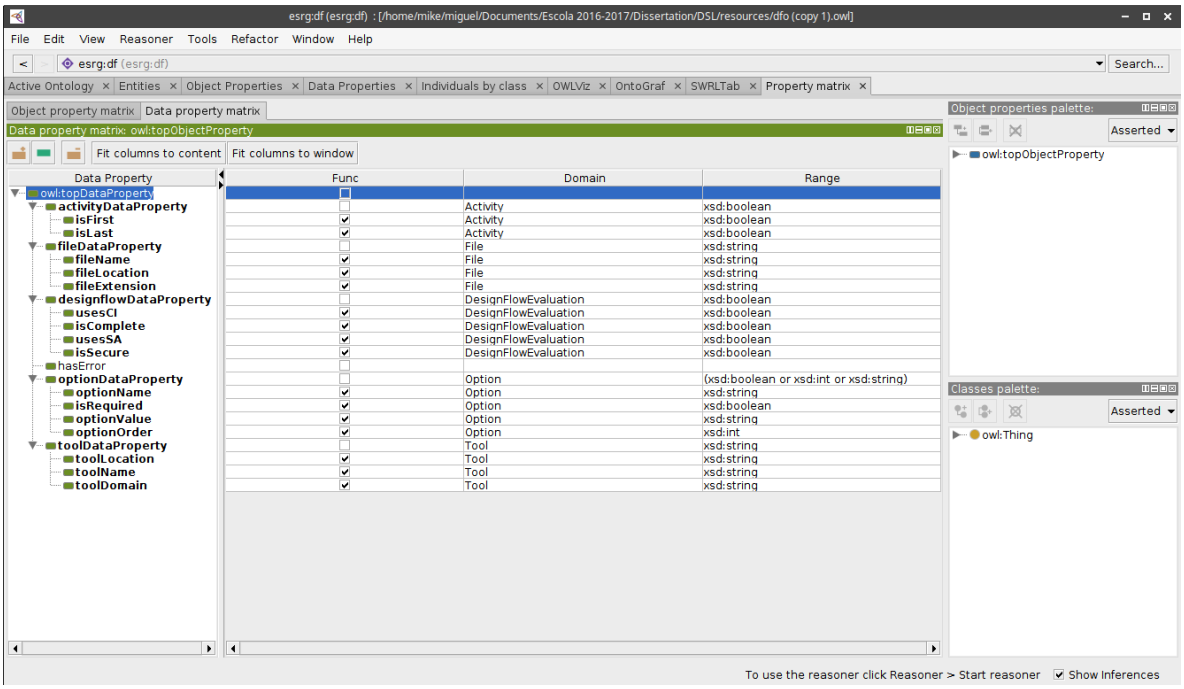


Figure 10.: Data properties matrix.

an individual Compile *hasTool* Compiler and also Compile *hasTool* CCompiler, then because *hasTool* is a functional property, the reasoner can infer that Compiler and CCompiler are the same individual. Nevertheless, if Compiler and CCompiler are explicitly declared as different individuals, the previous inference by the reasoner would lead the ontology to a state of inconsistency. Because an activity can have one and only one tool, *hasTool* was therefore declared as a functional property.

Regarding data properties, showed in Figure 10, all of them were declared as functional properties apart from *hasError*, as this property can have multiple values, for instance, the evaluation of a design flow can be negative with one or more errors.

4.1.3 *SWRL Rules*

The SWRL rules were inserted in the ontology with the help of the SWRLTab plug in, displayed in Figure 11. For each created rule was given a name as well as a short description of its purpose.

4.2 Design Flow Modeling Language

As stated before, the DSL was implemented using the Xtext framework with Eclipse IDE. For the creation of the language support infrastructure, Xtext requires only a grammar

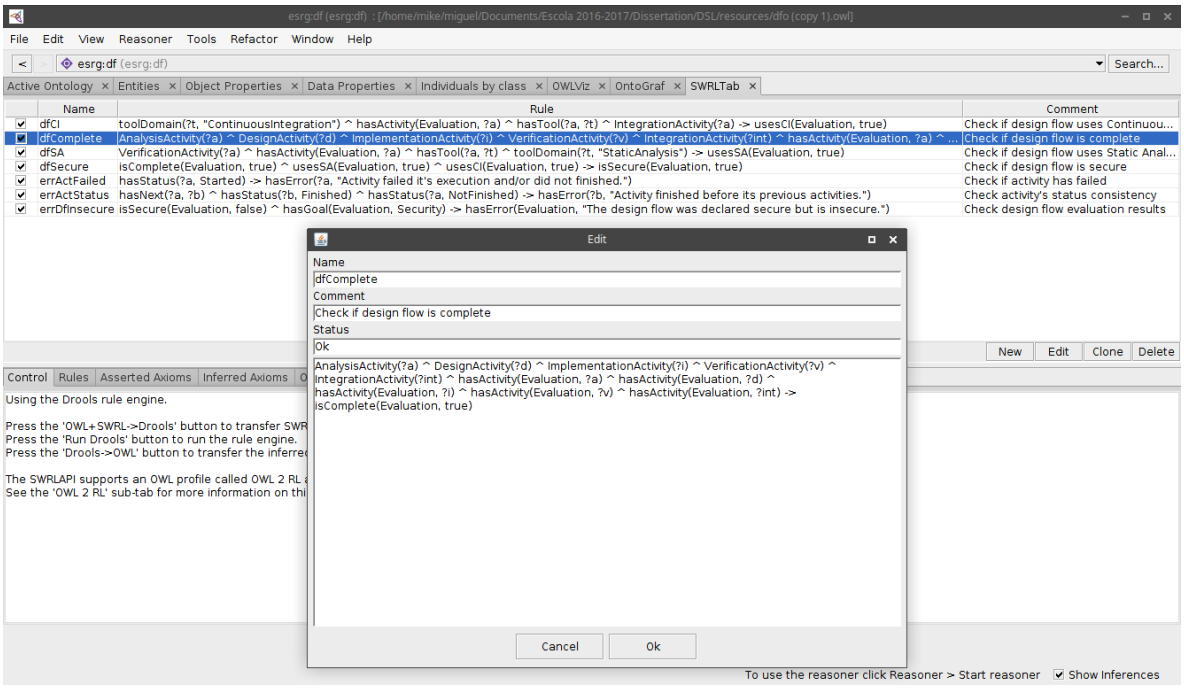


Figure 11.: SWRL rules.

specification of the language. Some aspects of the language can be directly described in the grammar while other have to rely on language validators, simply by choice or because they cannot be expressed in the grammar specification. Validators also work as a mechanism for providing additional constraint checks to the language that cannot be done at parsing time.

4.2.1 *Grammar*

The DFML grammar is described in over 100 lines of Xtext and displaying its full length would compromise the readability of the main text. Therefore smaller snippets of the grammar were selected to be detailed next while the complete grammar listing is provided in Appendix A.1.

```
1 Main:
    DesignFlowModel | TemplateModel
3 ;
```

Listing 4.1: Main rule snippet.

Listing 4.1 contains the *Main* rule which essentially delegates any model into two different rules each corresponding to those two types of dfml files, either a resource template or a design flow description.

```
1 TemplateModel:
    templates+=ResourceTemplate+
3 ;
```

Listing 4.2: TemplateModel rule snippet.

In the case of the *TemplateModel* rule, it creates a set of resource templates, denoted by the use of the '+=' signs, and it must contain at least one element, as is characterized be the usage of the rightmost '+' sign in the grammar rule (see Listing 4.2 line 2).

```
1 DesignFlowModel:
    (secure?='SecureDesignFlow')?
3   'Analysis:'
      analysisActivities+=[Activity]*
5   'Design:'
      designActivities+=[Activity]*
7   'Implementation:'
      implementationActivities+=[Activity]*
9   'Verification:'
      verificationActivities+=[Activity]*
11  'Integration:'
      integrationActivities+=[Activity]*
13  &activities+=Activity+
    &resources+=Resource+
15 ;
```

Listing 4.3: DesignFlowModel rule snippet.

As for the *DesignFlowModel* rule, displayed on Listing 4.3, it specifies the layout, attributes and elements of a design flow. First, it is declared a boolean attribute, given by the use of the '?=' characters, named 'secure' which is used to indicate whether the model being described requires security constraint checks or not. In Xtext, grammar keywords are denoted by being enclosed in single quotes ('), so for a design flow to be declared secure, the keyword 'SecureDesignFlow' must be used prior to its description. Then, follows the layout of the activities through the different phases of the design flow, with each phase having its own set of activities. The use of the '*' symbol, which means zero or more in Xtext, indicates that the set of any phase can contain zero or more elements. Each set's element is a cross-reference to an element of type Activity, as denoted by the use of the square brackets ([]). This means that a user can reference an activity by its name and Xtext will resolve that cross-reference by searching in the program for an element of the Activity type with the

given name. If no element is found and error is automatically issued. For this mechanism to work, the referred element must have an attribute called 'name'.

```
1 Activity:
    'Activity' name=ID '{'
3     resources+=[Resource]+
    '}'
5 ;
```

Listing 4.4: Activity rule snippet.

As can be seen in the *Activity* rule in line 2 of Listing 4.4, this indeed possesses the 'name' attribute which is preceded by the 'Activity' keyword. The activity's resources are enclosed in brackets ({}), and the resources set must contain at least one or more cross-references to a resource. This definition enforces the semantic value of the ontology's Activity concept and is an example of how semantic support is provided even from the grammar specification.

```
1 ResourceTemplate:
    ToolTemplate | FileTemplate
3 ;

5 Resource:
    Tool | File
7 ;
```

Listing 4.5: ResourceTemplate and Resource rules snippet.

In Listing 4.5 are shown two very similar rules that delegate the general resource into one of two types, either a tool or a file. Because the rules that handle each resource are very similar only the tool type will be described next.

```
1 ToolTemplate:
    'ToolTemplate' name=ID ('domain' domain=STRING)?'{'
3     options+=Option*
    '}'
5 ;
```

Listing 4.6: ToolTemplate rule snippet.

In the case of the *ToolTemplate* rule, the definition of a tool template begins with the use of the keyword 'ToolTemplate' followed by the name of the template. Optionally as indicated

by the use of the '?' symbol, the domain of the tool template is specified and the options available are enclosed in brackets, as shown in Listing 4.6.

```
Tool:
  'Tool' ('('toolTemplate=[ToolTemplate]')')? name=ID '{'
    'toolpath' toolPath=STRING
    options+=SetOption*
  '}'
;
```

Listing 4.7: Tool rule snippet.

The tool definition begins with the 'Tool' keyword followed by the template that its options are inherited from, which is an optional attribute, and its name, as can be seen in Listing 4.7. Enclosed in brackets are the 'toolPath' attribute which unequivocally references the tool's executable and a set of options, being the latter inherited from a tool template.

```
SetOption:
  option=[Option] '=' valueType=ValueType
;
```

Listing 4.8: SetOption rule snippet.

In Listing 4.8 is displayed the *SetOption* rule which is used to set options in resource definitions. It simply cross-references an option and attributes it a value.

```
ValueType:
  StringValue | FileReference
;

StringValue:
  value=STRING
;

FileReference:
  value=[File]
;
```

Listing 4.9: ValueType, StringValue and FileReference rules snippet.

This value can be one of two types, either a string literal (*StringValue* rule) or a cross-reference to a file (*FileReference* rule), that is why the *ValueType* rule was folded into two, as seen on Listing 4.9.

```
Option:
  (req?='required')? (mul?='multiple')? 'option' name=ID ('range' '{'
   range+=STRING (',' range+=STRING)* '}')?
;
```

Listing 4.10: Option rule snippet.

Finally, the definition of an option is declared in the *Option* rule in Listing 4.10. It contains two boolean attributes, 'req' and 'mul', which are used to characterize an option as required and/or multiple, respectively. Following those attributes is the keyword 'option', the name of the option and, optionally, the range of possible values the option can take, which are enclosed in brackets and preceded by the keyword 'range'.

Ecore Model

Xtext uses the *Eclipse Modeling Framework (EMF)*, which provides modeling and code generation facilities, for the creation of the model of the parsed program. From the grammar specification Xtext automatically infers the EMF metamodel for the language, that is defined in the Ecore format. Ecore format is a subset of UML class diagrams, as can be seen on Figure 12. During parsing, an EMF model is created from the parsed program, which is extensively used in tasks such as validation and code generation.

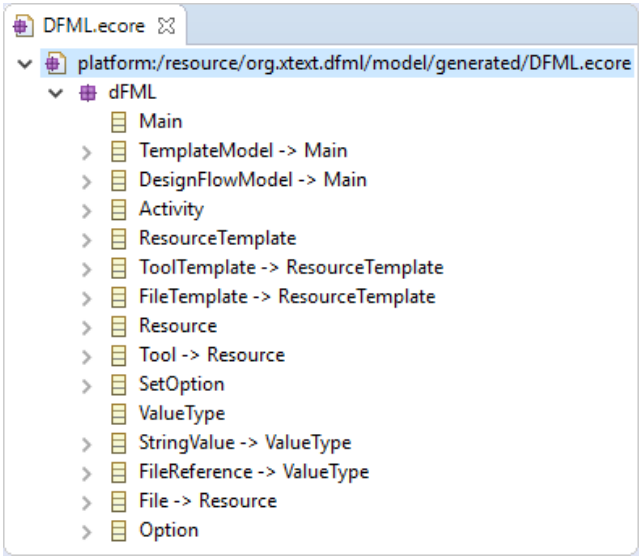


Figure 12.: DFML Ecore-based metamodel.

4.2.2 *Validators*

Validators are a mechanism used in Xtext to perform some checks to the language that cannot be expressed in the grammar or cannot be done at parsing time. For instance, a default validator that is automatically created by Xtext is one that validates that the name of every element in an EMF model is unique. Validators are written using the Xtend programming language in a very declarative way. They simply provide the errors or warnings to Xtext which takes care of generating the error markers in the DSL's IDE.

Xtext performs validation by invoking each method annotated with '@Check', passing all of the EMF model instances that have a compatible runtime type to each such method. The type of the single parameter is important because the error method can only be called on variables of the same type as the parameter.

```
@Check
def checkActivityHasTool(Activity activity) {
  var toolCounter = 0
  for (resource : activity.resources) {
    if (resource instanceof Tool) {
      toolCounter++
    }
  }
  if (toolCounter < 1) {
    error("Missing tool on Activity '" + activity.name + "'",
   DFMLPackage.Literals.ACTIVITY__NAME)
  }
  else if (toolCounter > 1) {
    error("Multiple tools on Activity '" + activity.name + "'",
   DFMLPackage.Literals.ACTIVITY__NAME)
  }
}
```

Listing 4.11: checkActivityHasTool validator.

The first validator checks if an activity contains a tool. As can be seen in Listing 4.11, this validator starts by creating a variable to store the number of tools of an activity, then it iterates through each element of an activity's resource set and for each element that is an instance of type 'Tool' it increments the tool counter. In the end two conditions are verified. If the number of tools is less than one is emitted an error stating there is a missing tool for the corresponding activity. If the number is higher than one is emitted an error stating there are multiple tools in the corresponding activity.

```
@Check
def checkOptionRange(SetOption setOption) {
  if (setOption.valueType instanceof StringValue) {
    if (!setOption.option.range.isEmpty && !setOption.option.range.
   contains((setOption.valueType as StringValue).value)) {
      error("Value does not belong to the option's range", DFMLPackage
   .Literals.SET_OPTION__VALUE_TYPE)
    }
  }
}
```

Listing 4.12: checkOptionRange validator.

In Listing 4.12 it is first checked if the 'valueType' is an instance of the 'StringValue'. If true, is then checked if the option's range is not empty and if it does not contain the value attributed to the option. If both these conditions are true is issued an error informing of the incompatibility between the option's value and the range of accepted values.

```
@Check
def checkActivityToolOptions(Activity activity) {
  var Tool tool
  val EList<File> toolRefFileList = new BasicEList<File>

  // Get activity's tool
  for (resource : activity.eCrossReferences) {
    if (resource instanceof Tool) {
      tool = resource as Tool
    }
  }

  // Get tool referenced files
  for (option : tool.options) {
    for (reference : option.valueType.eCrossReferences) {
      if (reference instanceof File) {
        toolRefFileList.add(reference)
      }
    }
  }

  for (file : toolRefFileList) {
    if (!activity.resources.contains(file)) {
```

```
24        error("Tool '" + tool.name + "' references file '" + file.name +
      "' that does not belong to the activity", DFMLPackage.Literals.
     ACTIVITY__RESOURCES)
      }
26   }
}
```

Listing 4.13: checkActivityToolOptions validator.

The validator on Listing 4.13 checks if the files referenced by the tool belong to the same activity as the tool itself. First is created a variable that will store the activity's tool and a list to store the referenced files followed by the identification of the activity's tool as well as the referenced files, which are each stored in their respective variables. Then the validator iterates through each element of the referenced files list and for each element that is not part of the activity's resources is issued an error.

```
1  @Check
   def checkUnusedActivity(Activity activity) {
3    //List of used activities
     val EList<Activity> usedActivitiesList = new BasicEList<Activity>
5    val model = activity.eContainer as DesignFlowModel

7    usedActivitiesList.addAll(model.analysisActivities)
     usedActivitiesList.addAll(model.designActivities)
9    usedActivitiesList.addAll(model.implementationActivities)
     usedActivitiesList.addAll(model.verificationActivities)
11   usedActivitiesList.addAll(model.integrationActivities)

13   //Traverse declared activities
     for (currentActivity : model.activities) {
15     if (!(usedActivitiesList.contains(currentActivity)) && (
      currentActivity.name == activity.name)) {
         error("Unused Activity '" + activity.name + "'", DFMLPackage.
      Literals.ACTIVITY__NAME)
17     }
     }
19 }
```

Listing 4.14: checkUnusedActivity validator.

To prevent unused activities from cluttering the design flow description, the validator from Listing 4.14 was implemented. First are created two variables, one is a list to store used

activities and the other stores the entire design flow model. Then, all the activities from the different phases are collected in the used activities list and, finally, the model is used to traverse through each activity to verify if each belongs to the used activities list. If not, an error is emitted stating the which activities are not being used.

In order to avoid extending this subject too much, the rest of the validators are detailed in the Appendix A.2.

4.2.3 *Generators*

After the program is parsed and its representational model is validated, code generation takes place. The DSL Xtext editor is already integrated in the automatic building infrastructure of Eclipse and so the generator is automatically called when a source file written in DFML is saved. To implement a code generator in Xtext is only necessary to override the `void doGenerate(Resource res, IFileSystemAccess fsa)` method, which receives an EMF `Resource` with the model of the program, and the `IFileSystemAccess` with the path location where generated code will be written to. In this method is only necessary to specify the relative path where the generated file will be created and its content as a string.

Generally, code generators are used in DSLs to generate code for GPLs, however in this case, the DFML framework uses them to provide support for the generation of CMakeLists.txt files according to Kitwares build system platform, CMake (Kitware, 2017).

The current CMake supported commands are present in the CMakeLists file template, written in DFML, in Listing 4.15. In a CMakeLists file, only the `cmake_minimum_required` command is required but other common commands are `project`, `add_executable` and `target_include_directories`.

```
1  FileTemplate CMakeLists {
     required option cmake_minimum_required
3    option enable_language
     option project
5    multiple option add_custom_command
     multiple option add_custom_target
7    multiple option add_executable
     multiple option add_library
9    multiple option add_subdirectory
     multiple option aux_source_directory
11   multiple option file
     multiple option include_directories
13   multiple option link_directories
     multiple option link_libraries
15   multiple option list
```

```
  multiple option set
  multiple option target_include_directories
  multiple option target_link_libraries
}
```

Listing 4.15: CMakeLists file template.

Listing 4.16 contains a snippet of the CMakeLists generator. First are retrieved from the model all the `File` objects that use the CMakeLists file template (line 4). Then, the list is iterated and for each object is generated a file using the `filePath` attribute which has its name. Regarding the content of the file, it is generated using the multi-line template expressions offered by Xtend. This type of expressions is enclosed in triple single quotes, can span multiple lines and a newline in the expression corresponds to a newline in the final output. The variable parts can be inserted in the expression using guillemets(≪≫), and valid Xtend code can be used inside them to perform loops, conditions or call methods. For instance, every file that is generated starts with a comment stating that it was generated by the DFML framework (line 8). Then, in the first set of guillemets (line 10), the model's File object is parsed for the first option, in this case `cmake_minimum_required`, after that, the name of the option is printed followed by the "(VERSION " set of characters and then the option's value followed by the closing parenthesis ')'. The rest of the generator works in a very similar way, parsing a command and then printing its name followed by its value enclosed in parenthesis, but for all the different supported commands. The complete listing of the generator is available in Appendix A.3.

```
override doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
  val model = resource.contents.get(0) as DesignFlowModel
  if (model !== null) {
    val cmakeFiles = model.getFilesFromTemplate("CMakeLists")
    for (cmakefile : cmakeFiles) {
      fsa.generateFile(cmakefile.filePath,
        '''
        # DFML generated file

        ≪val o1 = cmakefile.parseOption("cmake_minimum_required").
    iterator.next≫
        ≪o1.option.name≫(VERSION ≪(o1.valueType as StringValue).
    value≫)
        ≪IF !cmakefile.parseOption("project").empty≫

        ≪FOR o : cmakefile.parseOption("project")≫
```

```
15          ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
            ≪ENDFOR≫
17          ≪ENDIF≫
            ...
19          ≪IF !cmakefile.parseOption("add_custom_command").empty≫

21          ≪FOR o : cmakefile.parseOption("add_custom_command")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
23          ≪ENDFOR≫
            ≪ENDIF≫
25          '''
          )
27      }
    }
29  }
```

Listing 4.16: CMakeLists generator snippet.

4.3 DFML FRAMEWORK

Most of the DFML's IDE features were provided directly from Xtext, as it already provides a fully featured editor for the DSL being developed. Smaller customizations to the editor were introduced, but the major development effort was focused on its integration with semantic technology. Java was the language used for the development of the framework and the OWLAPI which provides an interface for manipulating OWL ontologies using Java (Horridge and Bechhofer, 2011).

4.3.1 *Inference Engine*

As was said previously in the Section 3.1, the Inference Engine is the core component of the DFML framework and, therefore, a specific Java class for this component was created.

```
1  public class InferenceEngine {
     public static class RTOntology {
3      public OWLOntology ontology = null;
       public boolean isInsecure = false;
5    }

7    private static ExecutorService executorService = Executors.
      newSingleThreadExecutor();
```

```
9   private static final boolean debug = false;
    private static final boolean verbose = true;
11
    private static final String ontIRI = "esrg:df";
13  private static final String rtOntIRI = "dfml:df";

15  private static boolean isInitialized = false;
    private static OWLOntologyManager manager = OWLManager.
     createOWLOntologyManager();
17  private static OWLDataFactory factory = OWLManager.getOWLDataFactory
     ();
    private static OWLOntology ontology = null;
19  private static RTOntology runtimeOntology = new RTOntology();
    private static PelletReasoner reasoner = null;
21  private static final String CONSOLE_NAME = "InferenceEngine";
    private static MessageConsole console = null;
23  private static MessageConsoleStream out = null;
    private static MessageConsoleStream err = null;
25
    private static void createConsole();
27  private static boolean checkConsistency(OWLOntology ontology);
    private static void saveOntology(String exportOnt) throws
     IOException;
29  private static void loadOntology(String importOnt) throws
     IOException;
    private static void parseDesignFlowModel(DesignFlowModel model);
31  private static void parseResources(EList<org.xtext.dFML.Resource>
     resources);
    private static void parseOptions(org.xtext.dFML.Resource resource,
     Set<OWLAxiom> axioms);
33  private static void parseActivities(EList<org.xtext.dFML.Activity>
     activities);

35  public static boolean init();
    public static void parseDSL(Resource resource);
37  public static void runDesignFlow(Resource resource);
}
```

Listing 4.17: InferenceEngine interface.

Listing 4.17 contains all the variables and functions implemented in this class which is detailed next. Starting in line 2, the `RTOntology` serves as a data structure for represent-

ing the runtime ontology of the InferenceEngine. It is composed of an ontology and a state variable which indicates its state regarding security. Since this class will be responsible for executing a design flow, that means it will need to be able to execute each step asynchronously. Java provides a series of utility classes commonly useful in concurrent programming from which the `ExecutorService` was selected for providing a complete asynchronous task execution framework, managing queuing and scheduling of tasks. More specifically, the `SingleThreadExecutor` (line 7) was used because, as read from the documentation, it uses a single worker thread operating off an unbounded queue, that means the number of tasks to be executed has no limits and they are executed sequentially one at a time. It perfectly fits to the situation of the design flow execution since, likewise, only one task will be executed at a time. Lines 9 and 10 contain two boolean flags, `debug` and `verbose`, used during development for controlling some debug features and the amount of verbose or information presented during the framework execution, respectively. On lines 12 and 13 are two strings which represent the *Internationalized Resource Identifiers (IRIs)* of the DFO and the *Runtime Ontology (RTO)*. These simply serve as a mean for identifying and referencing the ontologies. The `isInitialized` flag is used to indicate whether the InferenceEngine is initialized (line 15). For the InferenceEngine to be able to manipulate ontologies, an ontology manager is required. The `OWLOntologyManager` (line 16) is declared by the OWLAPI and provides methods for creating and editing ontologies. Simultaneously, the engine will also handle with OWL data types, thus requiring the `OWLDataFactory` (line 17) also declared by the OWLAPI. Lines 18 and 19 have the declaration of two variables for holding two ontologies, the first will be a reference to the DFO which will serve as a base for the construction of the second, the RTO. The `PelletReasoner` (line 20) is also necessary to be able to do inferences on the RTO. Lastly, for being able to display some information to the user, the framework requires a console. On line 21 is declared a string which contains the name of the console, line 22 contains the reference to the actual `MessageConsole` and lines 23 and 24 have the two streams of data to the console, one for general output and other for errors specifically.

Lines 26 to 33 contain several auxiliary methods used in the InferenceEngine. The `createConsole()` is used when initializing the engine for creating the console and assigning its output streams. For performing ontology consistency analysis, the `checkConsistency()` method is used which returns the result using a boolean data type. Besides checking for the logical consistency of the ontology, this method also provides information with regards to the security evaluation that is performed on the design flow, identifying and clarifying to the user the reason for the security evaluation results as well as providing guidelines on their correction, if so. The methods `loadOntology()` and `saveOntology()` are used for loading ontologies during the initialization of the engine and saving them various times throughout the execution of the framework. The last four methods are used in the first

major step of the InferenceEngine, the parsing of the model, with each method focusing on a specific element of the model. Generally, the parsing involves the translation of the information of the model into OWL axioms that are then added to the RTO.

The last three methods were declared public and correspond to the initialization of the engine and each of the two main steps it executes.

```
public static boolean init() {
  if (!isInitialized) {
    // Setup InferenceEngine console
    createConsole();

    // Load design flow ontology
    try {
      LoadOntology("resources" + File.separator + "dfo.owl");
    } catch (IOException e) {
      e.printStackTrace();
    }

    // Create a reasoner for the InferenceEngine
    if (ontology != null) {
      reasoner = PelletReasonerFactory.getInstance().
   createNonBufferingReasoner(ontology);

      // Check ontology consistency
      isInitialized = checkConsistency(ontology);
    }
  }
  return isInitialized;
}
```

Listing 4.18: init() method.

Starting with the `init()` method represented on Listing 4.18, it begins by verifying if the engine was already initialized. If not, then is created the engine's console with the auxiliary method `createConsole` (line 4) followed by the loading of the DFO ontology. After the load, comes the creation of an instance of the reasoner (line 15). First is verified if the ontology was in fact loaded successfully and, if so, the reasoner is created and the consistency of the ontology is immediately checked with the result being used for setting the state of the initialization of the engine. If the consistency is false, so is the initialization state and if it is true, which tends to always be the case since the ontology being loaded is a tested oc-

currence of the DFO, then the initialization is also true, finishing with the return statement of this value.

```
public static void parseDSL(Resource resource) {
  final String local_log = "[ParseDSL] ";

  // Check if the DSL model is valid
  IResourceValidator validator = ((XtextResource)resource).
   getResourceServiceProvider().getResourceValidator();
  List<Issue> issues = validator.validate(resource, CheckMode.
   FAST_ONLY, CancelIndicator.NullImpl);
  if (!issues.isEmpty()) {
    err.println(local_log + "There are errors in the model.");
    issues.forEach(error -> err.println(local_log + error.getMessage()
   ));
    out.println(local_log + "Aborting InferenceEngine run...");
  }
  else {
    out.println(local_log + "Parsing " + resource.getURI().lastSegment
   () + "...");

    try {
      // Create new ontology for runtime manipulation
      if (runtimeOntology.ontology != null) {
        if (manager.contains(runtimeOntology.ontology)) {
          manager.removeOntology(runtimeOntology.ontology);
        }
      }
      runtimeOntology.ontology = manager.createOntology(IRI.create(
   rtOntIRI), manager.getOntologies());
    } catch (OWLOntologyCreationException e1) {
      e1.printStackTrace();
    }

    final DesignFlowModel model = ((DesignFlowModel) resource.
   getContents().get(0));

    // Create all instances in the runtime ontology
    parseDesignFlowModel(model);
    parseResources(model.getResources());
    parseActivities(model.getActivities());
```

```
34      out.println(local_log + "Parse complete: " + runtimeOntology.
        ontology.getAxiomCount() + " axioms were inferred." );

36      // Check consistency
        checkConsistency(runtimeOntology.ontology);
38
        // Save ontology
40      try {
          SaveOntology(ResourcesPlugin.getWorkspace().getRoot().
        getLocation() + resource.getURI().path().replaceFirst("/resource",
        "").replace(".dfml", "") + ".owl");
42      } catch (IOException e) {
          e.printStackTrace();
44      }
    }
46  return;
}
```

Listing 4.19: parseDSL() method.

The `parseDSL()` method is in charge of performing the translation of the DFML model, that resulted from the parsed program, into an ontology so that this can be manipulated by the framework during the design flow execution, as shown in Listing 4.19.

The first step consists in obtaining an instance of the `IResourceValidator` to perform validation of the extracted model. If there are any issues those are displayed in the console to the user and the functions returns, otherwise the parsing process can begin. Before going straight to the parsing, it is first created the new ontology that would be used to hold the information derived from the model.

First is confirmed that there is not already an existing runtime ontology, if one does exist it is removed from the ontology manager, after that a clean runtime ontology is created and inserted with axioms copied from the initially loaded DFO (line 22). Concluding that procedure, the parsing begins with the declaration of the `model` variable which stores the model of the program (line 27) followed by the execution of the auxiliary functions that effectively parse the model elements.

Starting with the general design flow parsing, here the *hasGoal* data property may be asserted depending on the state of the `isSecure` attribute that is set in the DFML program. Then each activity has its individual created and assigned to the specific phase as well as have the order of execution established by the assertion of the *hasNext* object property, according to the model. The *hasStatus* object property is also asserted between each activity individual and the *NotStarted* individual, which essentially serves to set every activity in the not started state. Simultaneously for each activity are asserted the *isFirst* and *isLast* data

properties, accordingly, and the method ends with the insertion of all the asserted axioms in the RTO ontology.

The `parseResources()` method receives the list of resources from the model and iterates each of them, first identifying the type of the resource, file or tool, and following a series of procedures accordingly. For instance, in case of a tool, an individual of class *Tool* is created and, after the information of the model goes through some processing, the *toolName*, *toolLocation* and *toolDomain* data properties are asserted. Then, for each resource is executed the `parseOptions()` method which takes care of creating the individuals of the *Option* class and asserting their object and data properties.

The last auxiliary method executed is the `parseActivities()` which receives the list of activities from the model and asserts the *hasTool* and *hasFile* object properties for each activity individual as well as the *hasActivity* object property with the *DesignFlowEvaluation* individual, finishing with the insertion of all the axioms in the ontology.

After printing a short message indicating the successful parsing procedure, the just created RTO ontology is checked for logical consistency in addition to the design flow security evaluation and saved. The `parseDSL()` finishes with the return statement and with it the first step of the InferenceEngine is concluded.

4.3.2 *Task Executor*

During the design flow execution step, the framework needs to be able to handle the execution of the activities' commands and so, the `TaskExecutor` class was created.

```
public class TaskExecutor implements Callable<Boolean> {
  private String[] command;
  private static String workingDirectory = null;
  private static final String CONSOLE_NAME = "TaskExecutor";
  private static IOConsole console = null;
  private static BufferedReader in = null;
  private static IOConsoleOutputStream out = null;
  private static IOConsoleOutputStream err = null;

  private IOConsole findConsole(String name);

  TaskExecutor(String[] command, String workingDir);
  @Override public Boolean call();
}
```

Listing 4.20: TaskExecutor interface.

As can be seen from the interface shown in Listing 4.20, `TaskExecutor` implements the `Callable` Java interface since its instances will be executed by another thread, separated from the main thread. The *Runnable* interface was initially considered but because it was necessary to return the results of the command's execution, and since this does not allow any type of return, it ended up being rejected in favor of the `Callable` interface.

In terms of internal variables, the `TaskExecutor` class has an array of strings to store the command and its parameters (line 2) as well as a string for the working directory from which the command is executed. Similarly to the `InferenceEngine`, the `TaskExecutor` also uses a console which not only serves the purpose of presenting information to the user regarding the commands execution, but it is also used to prompt the user regarding its successful execution when such cannot be inferred automatically. Therefore, the class contains a string which stores the name of the console (line 4), a reference to the console itself (line 5) and three data streams, for input, output and errors.

Regarding its internal methods, it contains only the `findConsole()` method which is useful for each instance to be able to find and use the same console instead of having each spawning its own console, which would end up being very unpractical. If the console is not found then one is created.

The constructor (line 12) is called when each instance is created and it simply initializes all the internal variables.

Lastly, the `call()` method, which is an override method per the `Callable` Java interface, is where the command execution is effectively carried out. Listing 4.21 contains a snippet of its implementation.

```
@Override
public Boolean call()
{
  ...

  ProcessBuilder pb = new ProcessBuilder().redirectErrorStream(true);
  pb.directory(new File(workingDirectory));

  // Command processing
  ...

  pb.command(cmd);
  p = pb.start();

  Thread outputGobbler = new Thread(new StreamGobbler(p.getInputStream
    (), out));
  outputGobbler.start();
```

```
18    if (p.waitFor(INTERVAL, TimeUnit.SECONDS)) {
        retVal = p.waitFor();
20    }
      else {
22      throw new InterruptedException("Process became unresponsive\n");
      }
24    outputGobbler.join();

26    // Process return value
      ...
28  }
```

Listing 4.21: call() method code snippet.

After initializing some local variables and displaying in the console the working directory and command to be executed, a `ProcessBuilder` object is created to manage the command execution. First is set the working directory and, after some command processing, the command is also set and its execution started.

Because some commands might generate a lot of output, care was taken to have a dedicated separated thread just to handle it to prevent the main thread from failing to be able to process it, thus hanging the main application. This thread is created on line 15 and it starts executing on line 16.

Then, in the main thread is waited the command execution termination, but in order to prevent the command from remaining in execution indefinitely, the waiting lasts only for a determined interval, after which an exception is thrown to forcibly destroy the processing.

Normally, upon the termination of the command execution, all that is left to do is the processing of its return value. When this value is zero, which indicates a successful execution, the methods terminates, otherwise the user is informed of its value and prompted to either stop the design flow from proceeding or allow it to continue, if the return value is the expected one.

4.3.3 *Editor Customizations*

As stated in the beginning of this section, Xtext already provides most of the DFML editor features but some customizations were necessary to add some required functionality into the framework.

Menus and Keyboard Shortcuts

To begin with, the InferenceEngine execution needs to be embedded with the editor and to do so, some methods were implemented to allow its execution to be triggered via context menus or the use of keyboard shortcuts. When creating the DSL project for the first time, Xtext creates a dedicated project just for the *User Interface (UI)* portion of the DSL's Eclipse editor. In this project resides the plugin.xml file which contains information on how to display icons and menu items and so on in the editor's UI. Adding a new menu item to be displayed in the UI required the modification of this file and the actions associated with it were defined in a Java class. In total were developed two pairs of context-specific menu entries, one for each step of the `InferenceEngine`, for the project explorer context and another for the editor context, together with a keyboard binding for each menu entry. Because the implementation shares very similar procedures, only the menu entry for the execution of the first step of the `InferenceEngine` in the editor context will be detailed.

Starting with the menu handler which codes the behavior of the menu item, first is identified, from the event that is received from the UI, the instance of the active editor and from this is obtained a reference to the file currently displayed in the editor, which should be a DFML file. Then, a thread is spawned and verifies if the `InferenceEngine` is initialized before executing the `parseDSL()` method, as can be seen in Listing 4.22.

```
@Override
public Boolean exec(XtextResource file) throws Exception
{
  if (InferenceEngine.Init()) {
    InferenceEngine.ParseDSL(file);
  }
  return Boolean.TRUE;
}
```

Listing 4.22: Handler code snippet.

In order for a menu entry to display the item that would trigger this action, some entries were added in three extension points on the plugin.xml of the UI project, as shown in Figure 13. First was created a new handler, in the `org.eclipse.ui.handlers` extension point, that matches a `commandId` with the Java class of the given behavior which was described before. In the `org.eclipse.ui.commands`, was added an entry that refers to the `commandId` and provides the name to be displayed in the menu and a description of the command. Lastly, was added a new entry on the `org.eclipse.ui.menus` corresponding to the same `commandId` and a condition for when that menu entry is visible, which only happens when the DFML editor is opened.

Regarding the keyboard shortcuts, in the `org.eclipse.ui.bindings` extension point entries were added to bind the `commandId` with a sequence of key presses. The 'F1' key was bound to the first step of the engine while the 'F9' was bound to the second.

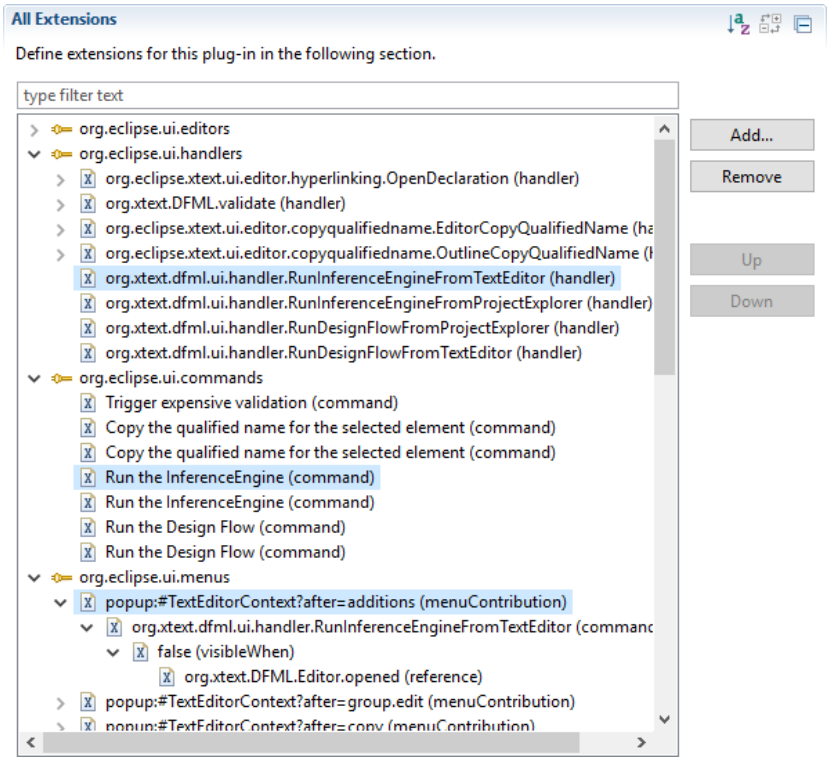


Figure 13.: UI plugin extension points.

DFML Project Wizard

Another customization introduced in the editor was a specific DFML project wizard that guides the user into creating and setting up a new DFML project. Once again, the Xtext framework proves its usefulness and flexibility by generating a lot of the boiler plate classes required to support such a wizard being only necessary to complete a few stubs. One of this stubs is the `generateInitialContents()` procedure which is automatically called when creating a new project using the project wizard. Like the name indicates, this method generates the initial content for the new project and, in this case, the project has available a series of '.dfml' files which provide support for some development tools like CMake, Make, PRQA's static code analyzer and Jenkins continuous integration system. Beyond that it also provides a sample design flow description, so the user can avoid starting one from scratch.

Example Project Wizard

One last customization implemented was the addition of another wizard, for the creation of a complete working example of a DFML project to help the user better understand the use of the framework. Once again, this only took the addition of a few entries on the `org.eclipse.emf.common.ui.examples` extension point of a plugin.xml file, displayed in Figure 14. These entries contain a small description of the example and a reference to a '.zip' file with all the project's content.

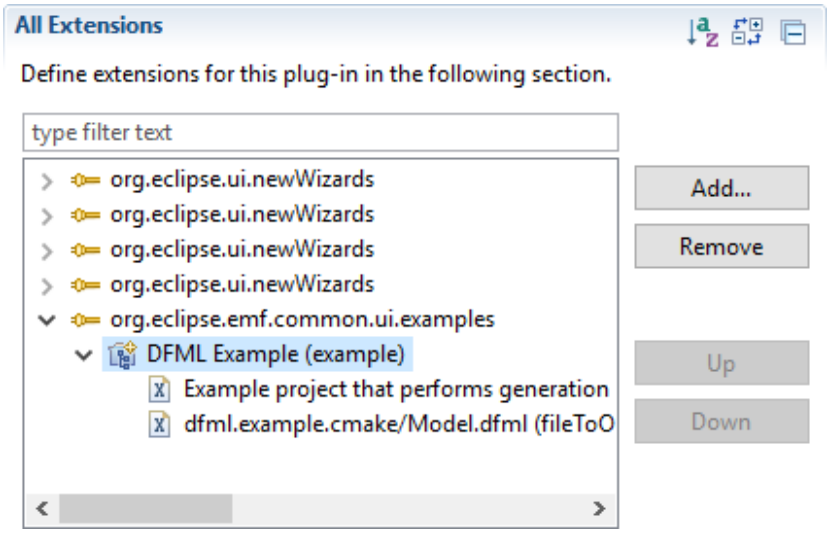


Figure 14.: Example plugin extension points.

4.4 SUMMARY

Presenting the implementation details of this dissertation was the focus of this chapter. First was explained the creation of the DFO which started with the creation of the classes and properties and the instantiation of individuals followed by the addition of the SWRL rules. Then, regarding the DFML language was described the grammar rules implemented in Xtext, the validators which perform additional checks that cannot be done at parsing time and the CMakeLists code generator. Finally, was demonstrated the implementation of the DFML framework. The two major classes, `InferenceEngine` and `TaskExecutor`, which make the core of the framework were explained in detail as well as some of the UI customizations done to the editor.

5

RESULTS

In this chapter are evaluated and presented the results of the work of this dissertation. First will be described the tests done regarding the ontology and the language followed by the exhibition of some use case scenarios to showcase the DFML framework and finishing with a discussion of the obtained results.

5.1 TESTS

The testing of the DFML framework was split according to its two main parts. Regarding the ontology, it was tested for its full integrity and usability as well as its semantic rules, which were tested separately. As for the DSL, this was tested using the Junit (Junit, 2017) framework and some additional utility classes provided by Xtext. Lastly, as part of a collaborative work, was conducted a code review on the hypervisor developed by some colleagues to assess its compliance with the MISRA standard.

5.1.1 *Design Flow Ontology*

During its elaboration, the DFO ontology was tested to avoid commonly known errors and validated to assess its usability using OOPS (Poveda-Villalón et al., 2009) which is available as a web service at `http://oops.linkeddata.es/`. As can be seen from Figure 15, three pitfalls were flagged in the DFO, one from the minor category and two belonging to the important category. The minor pitfall refers to 53 ontology elements from DFO which do not have any human readable annotations tied to them. The other two pitfalls refer to the lack of declaration of ontology metadata, such as version information, creation date and so on, and the lack of license information that applies to the ontology. Because the resolution of these pitfalls did not seem to directly impact the functionality of the DFML framework they were left to be addressed in a future iteration.

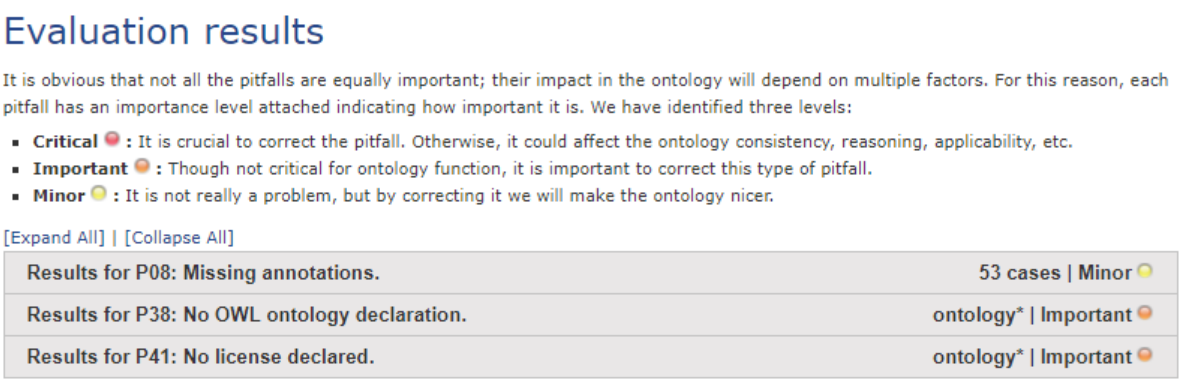


Figure 15.: OOPS evaluation results.

SWRL Rules

Contrary to the overall ontology evaluation, the evaluation of the SWRL rules were rather laborious. For each rule, the necessary conditions were manually asserted in a clean instance of the ontology, then the reasoner would be used to execute the rules and the inferences would be evaluated against the expected result.

The semantic rules were divided across three test scenarios, the first tested Rule-1, Rule-2, Rule-3 and Rule-4 in a scenario of a secure design flow being successfully evaluated, the second tested both Rule-5 and Rule-6 as involved errors related to activities, and the last tested Rule-7 which involved errors in the evaluation. For example, to test Rule-7, the object property *hasGoal* and the data property *isSecure* were both manually asserted for the *Evaluation* individual and after the activation of the reasoner, the inferred properties showed up, denoted by the colored background, as can be seen in Figure 16. As the inferred properties correspond to the expected, the test was deemed successful.

5.1.2 *Design Flow Modeling Language*

Xtext is integrated with Junit, which is a unit test framework for Java, and provides a series of classes that were used to perform tests on certain components of the DFML language during its implementation.

One of the first components tested was the language parser and the test consisted simply in providing a sample program to the parser and verify the constructed EMF model. This is accomplished using an assertion, which is a way of specifying the desired outcome and comparing it with the actual outcome. If both outcomes are the same, the assertion succeeds, otherwise, it fails.

All the developed language validators were also tested, each with its own unit test using Junit. For each test was provided a sample code that specifically caused the validator under test to fail and an assertion would be used to test that it failed.

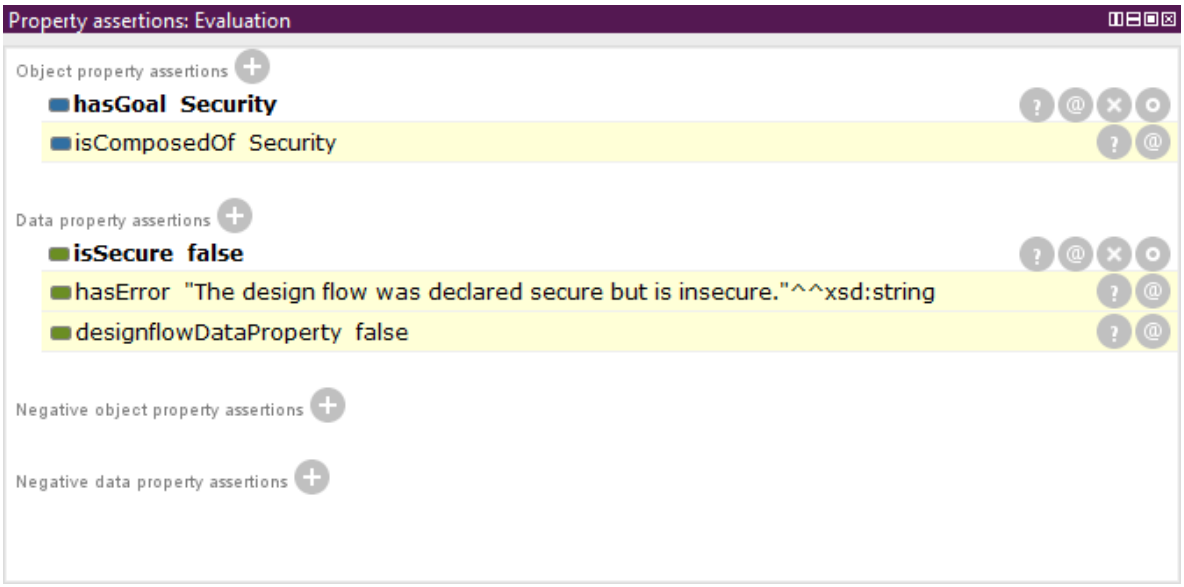


Figure 16.: SWRL Rule-7 test.

The Figure 17 shows the results of the execution of the Junit tests for both the language validators and the parser.

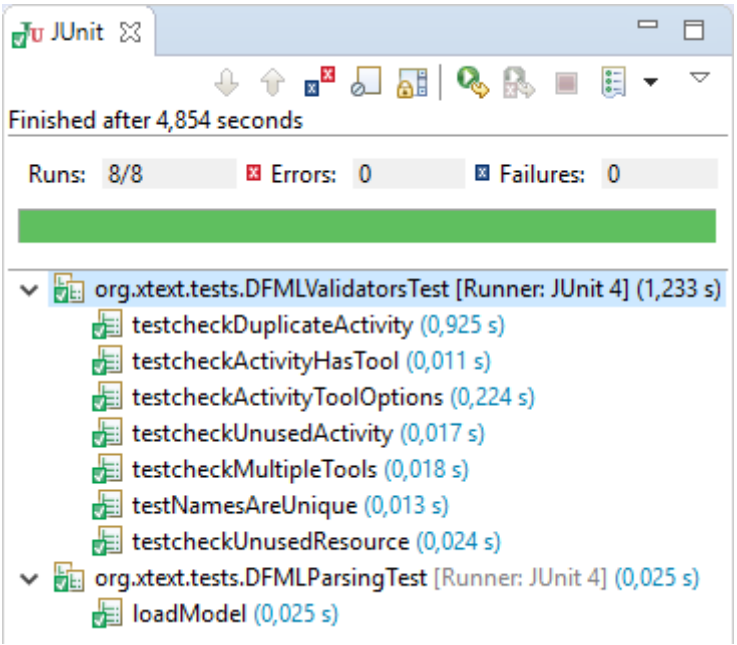


Figure 17.: Junit tests' results.

Table 5.: Hypervisor MISRA compliance before and after refactoring.

	Before	After
Number of files	24	21
Lines of code	2188	2185
Total preprocessed code lines	617	573
Diagnostic count	1074	261
Rule violation count	1151	273
Violated rules	58	22
Compliant rules	160	196
File compliance index	97.20%	98.90%
Project compliance index	73.39%	89.91%

5.1.3 *Hypervisor MISRA Compliance*

In order to assess the hypervisor's code compliance with the MISRA standard was used QA-C++, a static analysis tool used by industry-leading companies and developed by Programming Research. The tool requires the use of the MISRA compliance module, as by itself QA-C++ does not provide any MISRA checks. The module integrates with the static analyzer and provides coverage of 92% of all the subsets enforceable rules.

Table 5 contains data extracted from two rule compliance reports that were generated by the QA-C++ tool, one upon the first analysis of the hypervisor and another taken after some iterative refactoring of the hypervisor in order to comply with the MISRA guidelines. In the table, the number of files represents the source code files that make the code base of the hypervisor, counting with the header files included by those source code files. The lines of code metric is the sum of lines of code in source code files, not counting header files. The number of diagnostics represents all the diagnostics appearing in all files with diagnostics in header files being counted only once, even if included in multiple source files. The rule violation count is the sum off all the rule violations and is different from the count of diagnostics since a single diagnostic may refer to more than one rule violation. The last two metrics represent the overall compliance with the standard with the file compliance index representing the mean of all the individual files compliance and the project is calculated in a similar way but considering violations across all the files.

Summing up the data from the table, the total count of rule violations decreased from 1151 to 273, while the total number of compliant rules increased from 160 to 196, out of 219 rules that are enforced by the tool. These results are reflected in the principal metric given by the compliance report, the project compliance index, which increased 17%. The fact that both the file and project compliance indices are close indicates that each file in the project is violating the same rules.

5.2 RESULTS

In this section are presented the results achieved with the developed work, focusing mainly on the use of the DFML framework.

5.2.1 *Sample Project*

Starting with a simple and common task done in any IDE, this first demonstration covers the creation of a new DFML project using the custom developed wizard. Upon the creation and setup of a new DFML project, the user is greeted with the DFML editor as shown in Figure 18. In the Project Explorer view is displayed the structure of the DFML project, in the Outline view is displayed a different representation of the program consisting in a model build of its main elements, and on the right side is the DFML file editor which provides the user with content assists, like auto completion and references suggestions, when describing a design flow. By default, the project provides to the user a series of templates for tools and files as well as the DFML file for the design flow definition, which simply provides a stub for an activity and a tool.

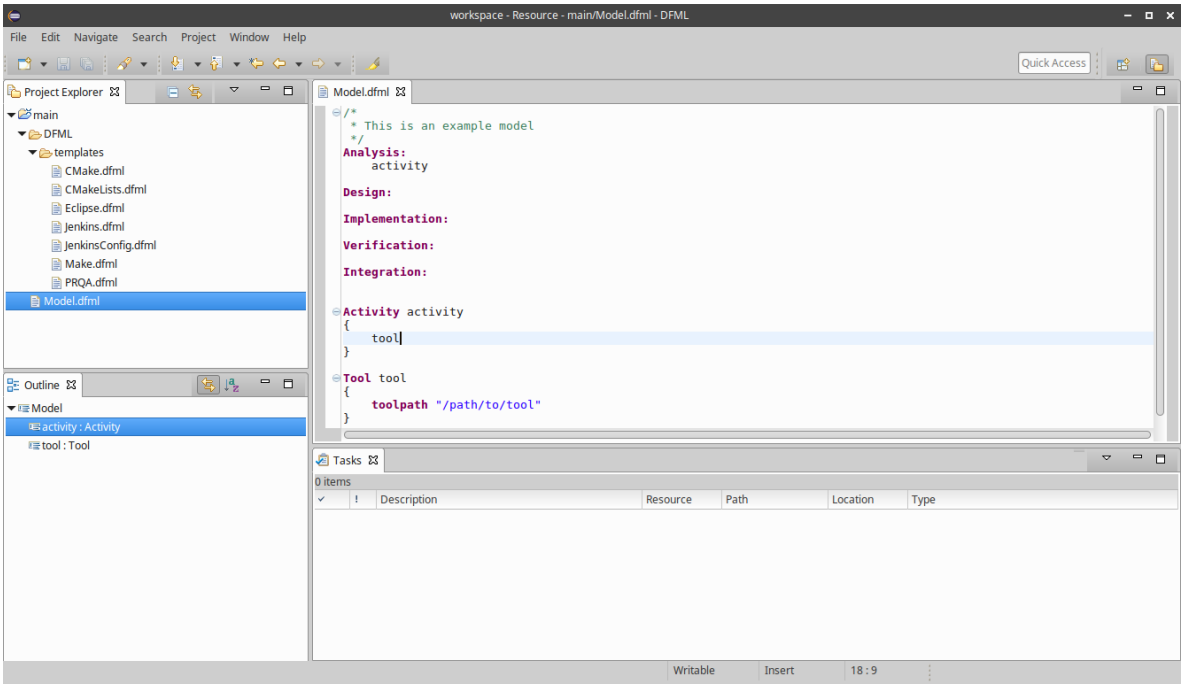


Figure 18.: DFML editor with sample project.

Figure 19 serves to showcase how errors are conveyed to the user in the framework. In this case, there are two errors in the program, one is in the multiple references of the same activity throughout the design flow and the other is in the reference of more than one

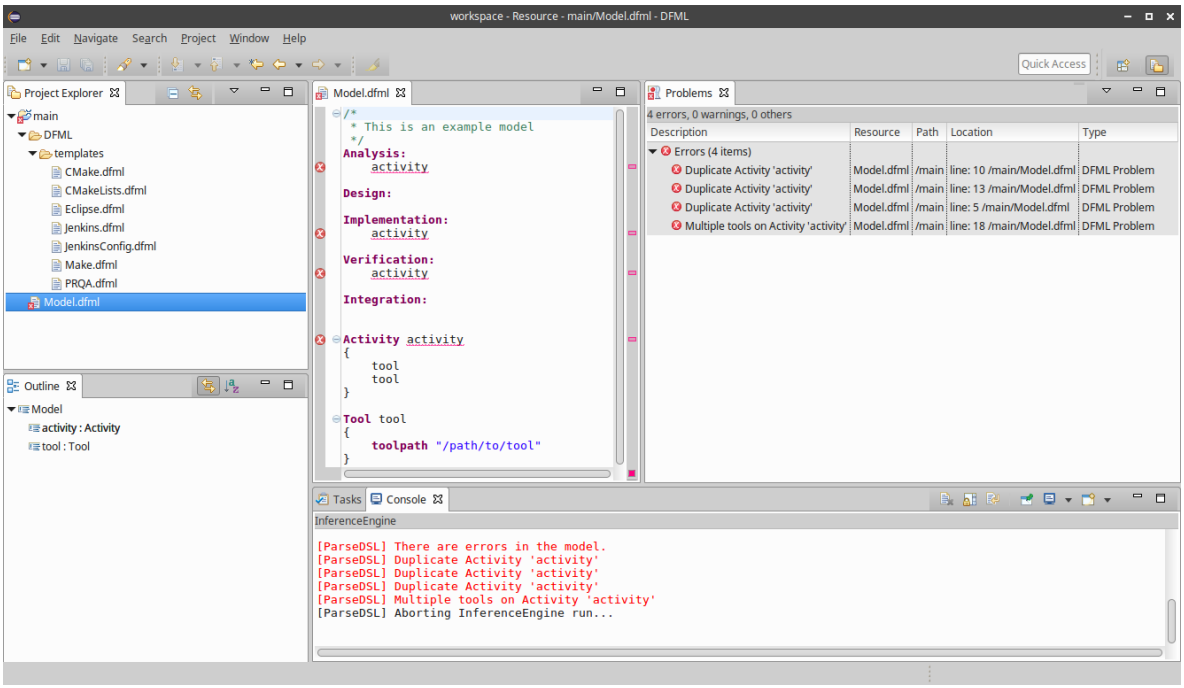


Figure 19.: DFML program with errors.

tool in the activity. In the editor are displayed error markers on the corresponding lines in addition to the squiggly lines shown under the name of the elements at fault. Beyond that, there is the Problems view which displays all the errors in addition to the location of their occurrence. Lastly, if the user attempts to execute the InferenceEngine, its execution is aborted, as the model is not valid, and the errors are printed in the console.

5.2.2 *Example Project*

The added example project consists of a simple design flow that includes three very common activities of embedded software development. Its goal is to generate a build system, compile and execute an application written in C++, which in this case consists of a classic printing of the 'Hello World' statement. Beyond the design flow program, the project includes all the DFML templates for the used tools, CMake and Make, the file template for the generation of the CMakeLists file and the C++ source code.

Figure 20 displays the wizard views. On the left is the New Project view that is presented to the user when creating a new project. The user has the option to create a new DFML project or create an already setup project using the provided example. After selecting the example option, on the right side is presented the New Example view that is showed to the user and displays a brief description about the project.

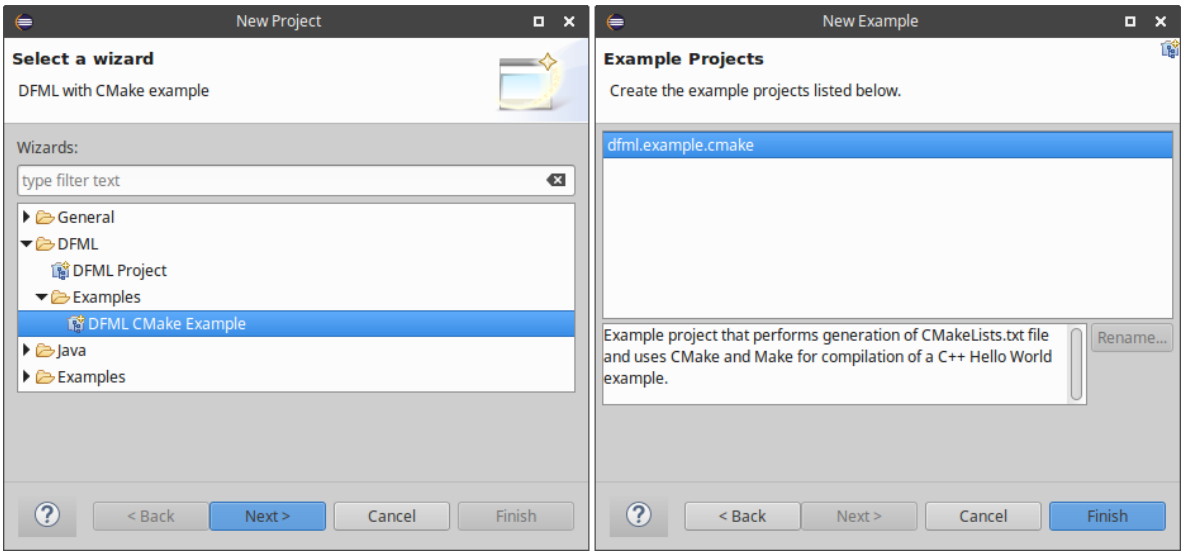


Figure 20.: Example wizard.

When the setup of the new example is finished the user is presented with the editor in the state displayed in Figure 21. On the Project Explorer view we have the content of the project in a tree view, there is the generated CMakeLists.txt file, some templates for tools and files used in the example, the source code of the C++ program, and the DFML file with the design flow description. Then there is the Outline view which displays the model of the program containing some condensed information and on the right side is the DFML program.

In Figure 22 can be seen, in the Console view, the output of the TaskExecutor console which resulted from the execution of the design flow in the example. In the Project Explorer view can be seen the added files as a result of the successful compilation of the program in addition to its equally successful execution, as evidenced by the printed 'Hello World'.

5.2.3 *Security Evaluation*

In order to show how security is effectively addressed, two examples were developed. In the first example, the design flow is declared as a secure design flow, but it only contains activities in the integration phase, concerning the Continuous Integration domain, and will, therefore, violate some of the SWRL-based rules of the ontology. As can be seen in Figure 23, the design flow evaluation results in the output seen on the InferenceEngine console, which specifies the reasons for the design flow not being considered secure and offers suggestions on how to make it so.

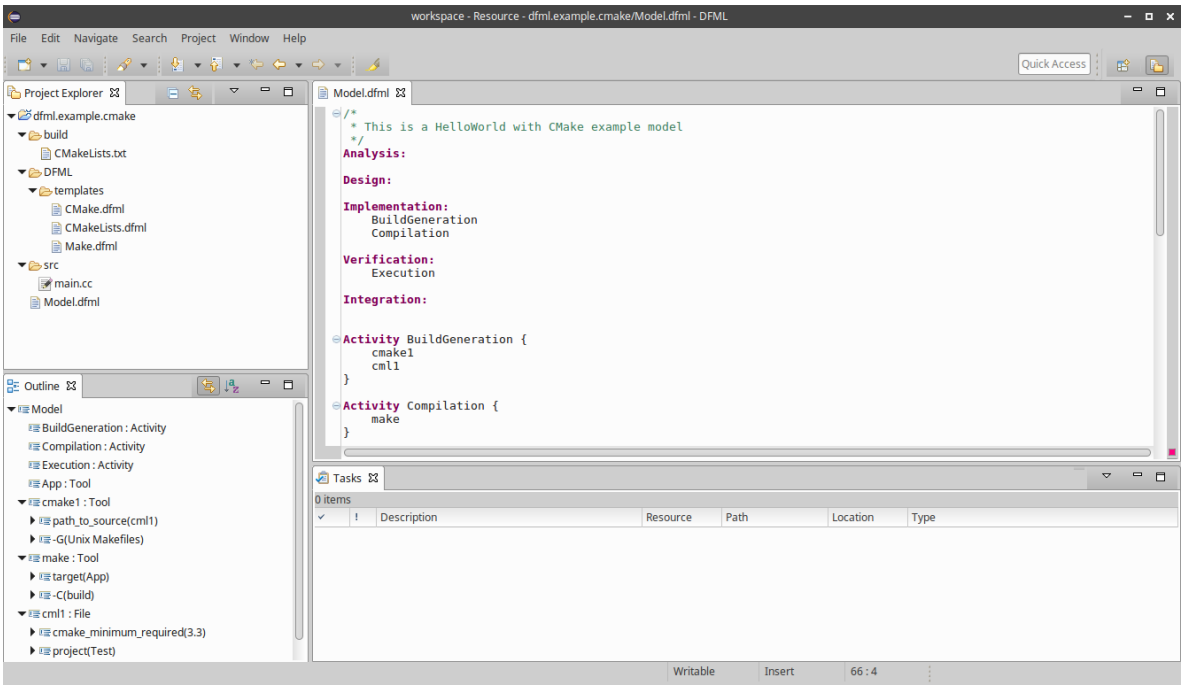


Figure 21.: DFML editor with example.

The second example, much like the previous, is also declared as secure design flow, and because it complies with all the criteria of the SWRL-based rules for a secure design flow, its execution proceeds without any errors, as is shown in Figure 24.

5.3 DISCUSSION

The examples presented previously demonstrate the flexibility provided by the DFML framework in the automation, security assessment and tool integration involved in the execution of a design flow. The example project demonstrated the automation and tool integration achieved with the framework for typical embedded software development tasks. It included the generation of the CMakeLists.txt file which is then used by an external tool to generate an entire build system. The last two examples showed how security is assessed by the DFML framework with the use of semantic rules applied over the ontology representation of the design flow.

The adoption of certain types of tools can help increase the productivity, quality and security, but according to Grammatech (2015), the best approach is the one that automates the use of a combination of tools from all categories and, as the obtained results show, the DFML framework can be a good proof of this approach.

One of the current downsides of the framework is the lack of flexibility during the design flow execution. Once its execution is initiated, this becomes immutable, and only stopped

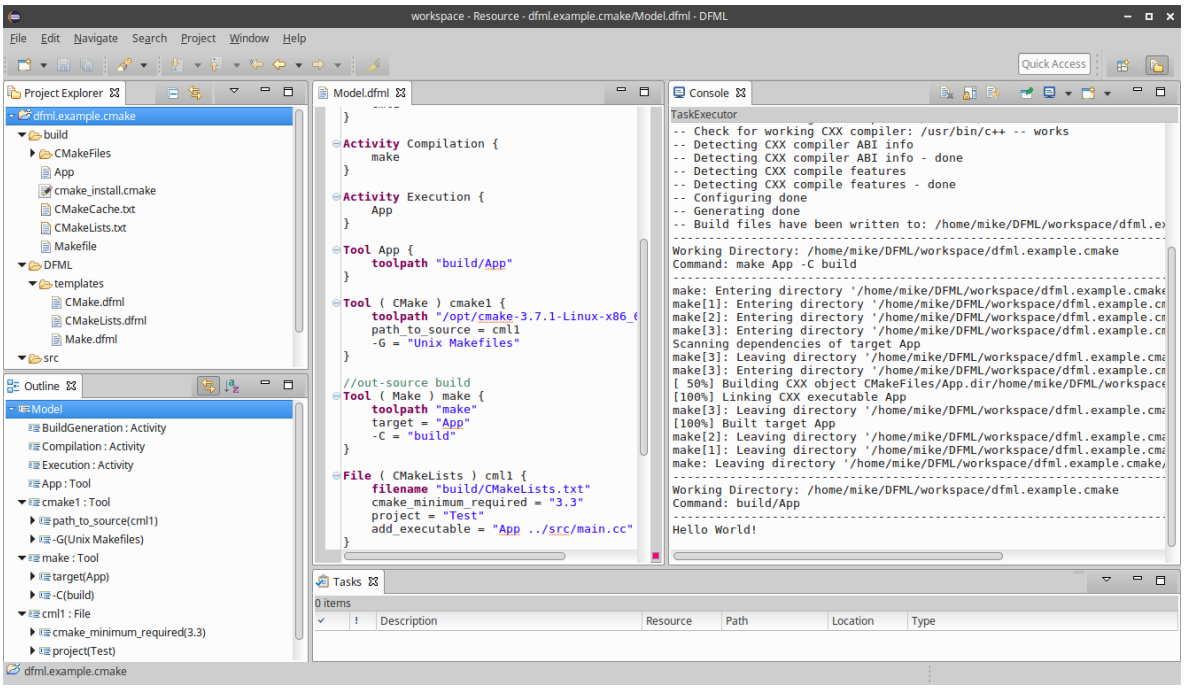


Figure 22.: Example design flow execution.

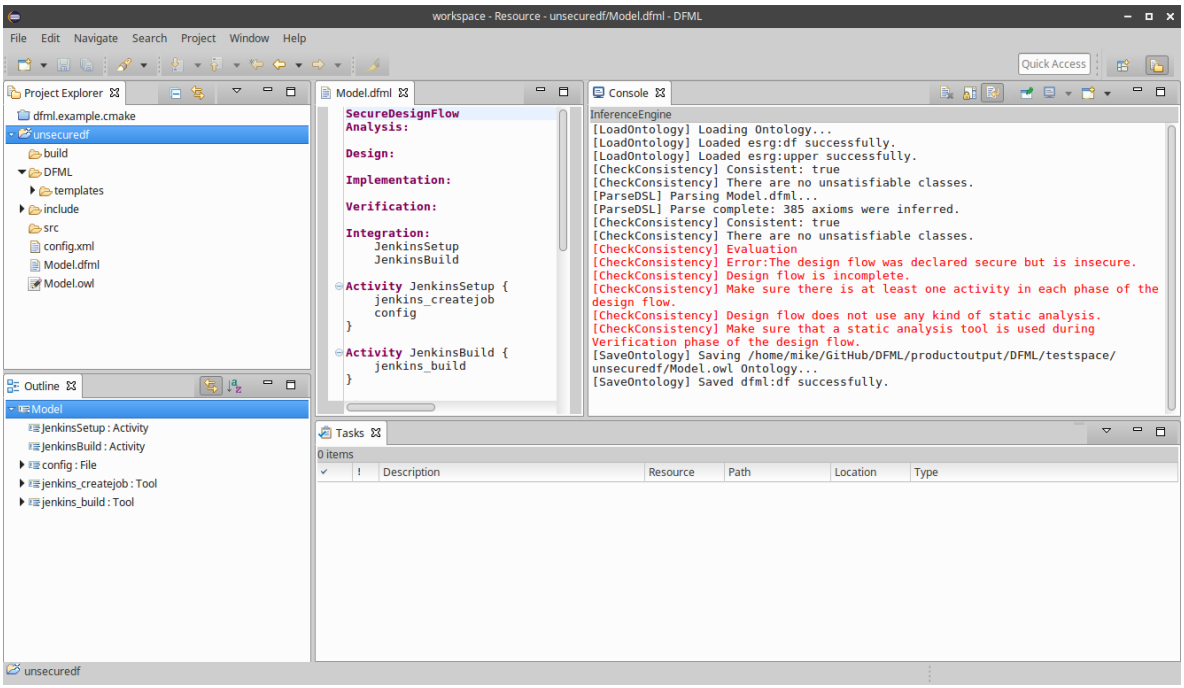


Figure 23.: Unsecure design flow example.

if an error occurs during the execution of one of its activities or it ultimately finishes. If any alteration to the design flow is deemed necessary, it cannot be done on-the-fly, instead the design flow must be stopped, updated and then started over from the first activity again.

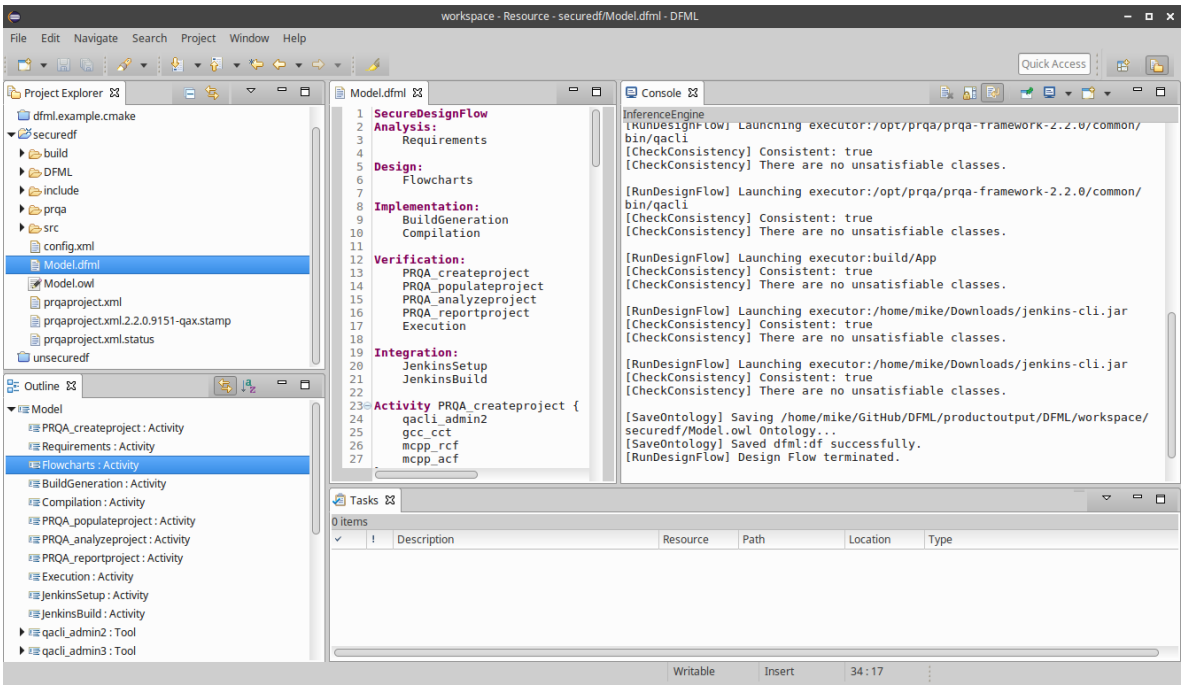


Figure 24.: Secure design flow example.

Another drawback is the inability to describe design flows with multiple paths. In a more sophisticated design flow, multiple paths of execution, and therefore decision-making, are certainly necessary, but it is currently unsupported by the DFML framework.

5.4 SUMMARY

This chapter focused mainly on the presentation of the accomplished results with the work of this dissertation. First were explained and clarified the tests conceived for both the ontology and the language in addition to the tools and methodologies employed. Then were demonstrated the results ultimately achieved in the development of the DFML framework with a series of examples and pictures of the DFML editor, finishing with a discussion on the obtained results.

6

CONCLUSION

In this chapter are presented the conclusions drawn from the work of this dissertation and are discussed some prospects for future work.

6.1 CONCLUSIONS

This dissertation described the effort taken in the development of the DFML framework, an ontology enhanced modeling DSL for secure design flow targeting embedded software development. The project was based on the integration of two modeling technologies, ontologies and DSLs, which were used to promote design flow automation and tool integration while addressing security from the outset.

The use of a DSL provides guidance to the software developer through the definition of the design flow and makes it easier to customize according to the product being developed.

Regarding the use of semantic technology, its integration with the DSL allowed the specification of additional constraints and semantics, ultimately providing better validation mechanisms and knowledge consistency of the design flow domain.

The different design flow examples were used as a proof of concept and its results demonstrate how security is enforced through semantic rules and how easy development tools integration and design flow automation can be using the DFML framework.

With the compliance with safety and security standards becoming increasingly important every day and likely to become mandatory in the future, tools like the DFML framework can make an impact by effectively reducing the development time, enforcing security from the outset and helping lower the efforts of certification.

6.2 PROSPECT FOR FUTURE WORK

In terms of the future developments, series of features were identified that could be implemented in order to further enhance the framework, not only regarding functionality but also to improve its maturity. One of the next steps of improvement would be the addition

of more semantic rules to the DFO ontology that can model the complete set of requirements of safety and security standards, such as CC, IEC 61508 and others. Another area with room for improvement is the framework's code generation feature. Currently, the only way to provide code generation is to implement the code generator within the framework development, which requires the recompilation of the entire tool in order to be available for the user. By taking advantage of the flexibility that Xtext provides, is possible to make the framework extendable with custom code generators provided by the users at runtime. That would require the development of an interface for code generators that could be plugged into the DFML framework, the creation of an extension point in the framework to accept the user defined code generators using the developed interface and, of course, any required refactoring of the framework to accommodate such feature. One last thing that should be addressed by the framework is traceability, which can be defined as the ability to help stakeholders understand the associations and dependencies that exist among entities created or used during a software development process (Pan et al., 2013). In its current state, the framework only allows the results of the activities to be viewed in the console as long as the current instance of the framework is being executed, and although it is possible to integrate a third-party tool that provides traceability in the design flow it makes sense that this feature should be part of the framework.

6.3 Final Considerations

The elaboration of this dissertation contributed to the knowledge of the author and the scientific community through the authorship and co-authorship of a scientific article in the field of embedded systems. Below is presented the article already submitted.

S. Pinto, M. Macedo, J. Martins, J. Alves, A. Tavares, "DFML: An Ontology Enhanced Domain-Specific Language for Modeling Secure Design Flow", submitted to IEEE "Embedded Systems Letters" on November 2017.

BIBLIOGRAPHY

ANSYS. Qualified Code Generation Greatly Reduces Cost of Safety-Critical Automotive Software. *White Paper*, pages 1–4, 2016.

Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2016. ISBN 9781782160304.

Andrzej Bialas. Intelligent sensors security. *Sensors*, 10(1):822–859, 2010. ISSN 14248220. doi: 10.3390/s100100822.

Andrzej Bialas. Common criteria related security design patterns for intelligent sensors-knowledge engineering-based implementation. *Sensors*, 11(8):8085–8114, 2011. ISSN 14248220. doi: 10.3390/s110808085.

Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL reasoners. In *CEUR Workshop Proceedings*, volume 350, 2008. ISBN 9781595936493.

David F. C. Brewer. Ten Years On: Doesn't the World of Security have anything to offer the World of Safety? In *Lessons in System Safety, Proceedings of the Eighth Safety-critical Systems Symposium, Southampton, UK 2000*, pages 246–268. Springer-Verlag, 2000. ISBN 9781852332495.

CCMODE. Common criteria compliant, modular, open it security de-velopment environment, 2017. URL http://www.commoncriteria.pl/.

Jean-Louis Colao, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017. Invited paper.

Sven Efftinge and Sebastian Zarnekow. Xtend documentation, 2011. URL http://www.eclipse.org/xtend/documentation/.

Moritz Eysholdt and Heiko Behrens. Xtext. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, page 307, 2010. ISBN 9781450302401. doi: 10.1145/1869542.1869625. URL http://portal.acm.org/citation.cfm?doid=1869542.1869625.

Martin Fowler. Continuous Integration, 2006. URL http://martinfowler.com/articles/continuousIntegration.html.

Gartner. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016, 2017. URL https://www.gartner.com/newsroom/id/3598917.

Grammatech. A Four-Step Guide to Security Assurance for IoT Devices. Technical report, Grammatech, 2015.

Tom Gruber. Definition of Ontology, 2007.

Peter B. Gutgarts and Aaron Temin. Security-critical versus safety-critical software. In *2010 IEEE International Conference on Technologies for Homeland Security, HST 2010*, pages 507–511, 2010. ISBN 9781424460472. doi: 10.1109/THS.2010.5654973.

Matthew Horridge and Sean Bechhofer. The OWLAPI: a Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011. ISSN 15700844. doi: 10.3233/SW-2011-0025.

Ian Horrocks, Peter F Patel-schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL : A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member submission 21*, pages 1–20, 2004.

IEC. IEC 61508 : Functional safety of electrical/electronic/ programmable electronic safety-related systems, 2005.

ISO. ISO/IEC 15408-1:2009 - Evaluation Criteria for IT Security, 2009. URL https://www.iso.org/standard/50341.html.

ISO. ISO 26262 Road Vehicles Functional safety, 2011.

Stephen C Johnson. Yacc : Yet Another Compiler-Compiler. *Computing Science Technical Report No. 32*, page 33, 1975.

Junit. Junit4, 2017. URL http://junit.org/junit4/.

Kitware. Cmake, 2017. URL https://cmake.org/.

M.E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. *Computing science technical report*, 39:12–9, 1975. ISSN 19447973. doi: 10.1029/WR004i005p01115. URL http://ken-cc.googlecode.com/svn/trunk/doc/lex.pdf.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. ISSN 03600300. doi: 10.1145/1118890.1118892. URL http://portal.acm.org/citation.cfm?doid=1118890.1118892.

MISRA. *MISRA-C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems*. MISRA, 2008. ISBN 9780952415626.

Anders Møller and Michael I. Schwartzbach. Static program analysis. *Elektronische Rechenanlagen*, 27(2):89–95, 2015. URL http://cs.au.dk/{~}amoeller/spa/spa.pdf.

Ernest Mougoue. Ssdlc 101: What is the secure software development life cycle?, 2016. URL https://www.synopsys.com/blogs/software-security/secure-sdlc/.

Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. *Stanford Knowledge Systems Laboratory*, page 25, 2001. ISSN 09333657. doi: 10.1016/j.artmed.2004.01.014.

Natalya F Noy, Monica Crubézy, Ray W Fergerson, Holger Knublauch, Samson W Tu, Jennifer Vendetti, and Mark A Musen. Protégé-2000: An Open-Source Ontology-Development and Knowledge-Acquisition Environment. *AMIA Annu Symp Proc*, 953: 953, 2003. ISSN 1942-597X. doi: D030003158[pii]. URL http://protege.stanford.edu.

Jeff Z. Pan, Steffen Staab, Uwe Aßmann, Jürgen Ebert, and Yuting Zhao. *Ontology-driven software development*. Springer-Verlag Berlin Heidelberg, 2013. ISBN 9783642312267. doi: 10.1007/978-3-642-31226-7.

Terrence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. URL http://www3.interscience.wiley.com/journal/113446166/abstract.

María Poveda-Villalón, Mari Carmen Suárez-Figueroa, Miguel Ángel García-Delgado, and Asunción Gómez-Pérez. OOPS! (OntOlogy Pitfall Scanner!): supporting ontology evaluation on-line. *Undefined*, 1:1–5, 2009. ISSN 1552-6283. doi: 10.4018/ijswis.2014040102. URL http://www.semantic-web-journal.net/system/files/swj989.pdf.

PRQA. Succeding with Static Code Analysis: An Implementation Guide, 2016.

Oscar Slotosch, Martin Wildmoser, Jan Philipps, Reinhard Jeschull, and Rafael Zalman. ISO 26262 - Tool chain analysis reduces tool qualification costs. *Automotive - Safety & Security*, 210:27–38, 2012. ISSN 16175468.

W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. *OWL 2 Web Ontology Language*, pages 1–7, 2012. URL http://www.w3.org/TR/owl2-overview/.

Tobias Walter. Combining Domain-Specific Languages and Ontology Technologies. In *Proceedings of the Doctoral Symposium at MODELS 2009*, 2009.

A

LISTINGS

This appendix provides several code listings whose length would compromise readability of the main text.

A.1 DFML GRAMMAR

Listing A.1 contains the entire DFML grammar specification in Xtext.

```
grammar org.xtext.DFML hidden(WS, ML_COMMENT, SL_COMMENT)
generate dFML "http://www.xtext.org/DFML"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Main:
  DesignFlowModel | TemplateModel
;

TemplateModel:
  templates+=ResourceTemplate+
;

DesignFlowModel:
  (secure?='SecureDesignFlow')?
  'Analysis:'
    analysisActivities+=[Activity]*
  'Design:'
    designActivities+=[Activity]*
  'Implementation:'
    implementationActivities+=[Activity]*
  'Verification:'
    verificationActivities+=[Activity]*
  'Integration:'
    integrationActivities+=[Activity]*
```

```
  &activities+=Activity+
  &resources+=Resource+
;

Activity:
  'Activity' name=ID '{'
    resources+=[Resource]+
  '}'
;

ResourceTemplate:
  ToolTemplate | FileTemplate
;

ToolTemplate:
  'ToolTemplate' name=ID ('domain' domain=STRING)?'{'
    options+=Option*
  '}'
;

FileTemplate:
  'FileTemplate' name=ID '{'
    options+=Option*
  '}'
;

Resource:
  Tool | File
;

Tool:
  'Tool' ('('toolTemplate=[ToolTemplate]')')? name=ID '{'
    'toolpath' toolPath=STRING
    options+=SetOption*
  '}'
;

SetOption:
  option=[Option] '=' valueType=ValueType
;

ValueType:
  StringValue | FileReference
```

```
68 ;

70 StringValue:
     value=STRING
72 ;

74 FileReference:
     value=[File]
76 ;

78 File:
     'File' ('('fileTemplate=[FileTemplate]')')? name=ID '{'
80     'filename' filePath=STRING
       options+=SetOption*
82   '}'
;
84
Option:
86   (req?='required')? (mul?='multiple')? 'option' name=ID ('range' '{'
      range+=STRING (',' range+=STRING)* '}')?
;
88
/*
90  * Terminal rules
 */
92
terminal FLOAT returns ecore::EFloat:
94 /*  ('-'|'+')?*/ (INT '.' INT) (('e' | 'E') INT)?;

96 terminal BOOL returns ecore::EBoolean:
     'true' | 'false';
98
terminal ID:
100  '^'? ('a'..'z' | 'A'..'Z' | '_' | '-') ('a'..'z' | 'A'..'Z' | '_' |
      '-' | '0'..'9')*;

102 terminal INT returns ecore::EInt:
     /*('-' | '+')?*/ ('0'..'9')+;
104
terminal STRING:
106  '"' ('\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\' | '"
      '))* '"' | "'" ('\\' .
     /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\' | "'"))* "'";
```

```
108
terminal ML_COMMENT:
110   '/*'->'*/';

112 terminal SL_COMMENT:
   '//' !('\n' | '\r')* ('\r'? '\n')?;
114
terminal IRI:
116   ('#' | '&'..'+' | '-'..'.'  | '0'..';' | '?'..'Z' | 'a'..'z' | '_')
     +;

118 terminal WS:
   (' ' | '\t' | '\r' | '\n')+;
120
terminal ANY_OTHER:
122   .;
```

Listing A.1: DFML grammar.

A.2 VALIDATORS

```
@Check
2 def checkDuplicateActivity(DesignFlowModel model) {
   // Collect all activities
4   val allActivities = <Integer, Activity>newHashMap()
   allActivities.putAll(activityIndex(model.analysisActivities,
     ANALYSIS_ID))
6   allActivities.putAll(activityIndex(model.designActivities, DESIGN_ID
     ))
   allActivities.putAll(activityIndex(model.implementationActivities,
     IMPLEMENTATION_ID))
8   allActivities.putAll(activityIndex(model.verificationActivities,
     VERIFICATION_ID))
   allActivities.putAll(activityIndex(model.integrationActivities,
     INTEGRATION_ID))
10  findDuplicateActivities(model, allActivities)
}
12
def activityIndex(EList<Activity> activityList, Integer phaseID) {
14  val activityIndexMap = <Integer, Activity>newHashMap()
   var index = 0
```

```
16
   //Traverse activities list
18   for (activity : activityList) {
       activityIndexMap.put(index+phaseID, activity)
20     index++
     }
22   return activityIndexMap
}
24
def findDuplicateActivities(DesignFlowModel model, HashMap<Integer,
    Activity> activityList) {
26   val visitedActivities = <Activity, Integer>newHashMap()
   val duplicateActivities = <Integer, Activity>newHashMap()
28
   //Traverse activities list
30   val Iterator<Integer> iter = activityList.keySet.iterator
   while (iter.hasNext) {
32     val index = iter.next
     var activity = activityList.get(index)
34
     //Found duplicate
36     if (visitedActivities.containsKey(activity)) {
       duplicateActivities.put(visitedActivities.get(activity),
    activity)
38       duplicateActivities.put(index, activity)
     }
40     else {
       visitedActivities.put(activity, index)
42     }
   }
44
   //Throw error to every duplicate found
46   val Iterator<Integer> iter2 = duplicateActivities.keySet.iterator
   while (iter2.hasNext) {
48     var i = iter2.next
     val activity = duplicateActivities.get(i)
50     var EStructuralFeature feature
     if (ANALYSIS_ID <= i && i < DESIGN_ID) {
52       feature = DFMLPackage.Literals.
    DESIGN_FLOW_MODEL__ANALYSIS_ACTIVITIES
       i = i - ANALYSIS_ID
54     }
     else if (DESIGN_ID <= i && i < IMPLEMENTATION_ID) {
```

```
56        feature = DFMLPackage.Literals.
    DESIGN_FLOW_MODEL__DESIGN_ACTIVITIES
        i = i - DESIGN_ID
58      }
      else if (IMPLEMENTATION_ID <= i && i < VERIFICATION_ID) {
60        feature = DFMLPackage.Literals.
    DESIGN_FLOW_MODEL__IMPLEMENTATION_ACTIVITIES
        i = i - IMPLEMENTATION_ID
62      }
      else if (VERIFICATION_ID <= i && i < INTEGRATION_ID) {
64        feature = DFMLPackage.Literals.
    DESIGN_FLOW_MODEL__VERIFICATION_ACTIVITIES
        i = i - VERIFICATION_ID
66      }
      else if (INTEGRATION_ID <= i) {
68        feature = DFMLPackage.Literals.
    DESIGN_FLOW_MODEL__INTEGRATION_ACTIVITIES
        i = i - INTEGRATION_ID
70      }
      error("Duplicate Activity '" + activity.name + "'", feature, i)
72    }
  }
```

Listing A.2: checkDuplicateActivity validator and auxiliary methods.

The validator in Listing A.2 is responsible for verifying there are no duplicate activities being used in the design flow. First is created a hash map that maps an integer number to an activity followed by the collection of all the activities from every phase into this hash map. During the collection the `activityIndex()` method is used to perform some conversion between the activity index and the phase it belongs to. Lastly, is called the `findDuplicateActivities()` method which issues an error for each duplicate activity. These methods were created to avoid complexity in the validator since was necessary to introduce some extra logic to traverse hash maps. In essence, the method iterates through every activity and each duplicate found is added to a second hash map of duplicated activities. Finally, it iterates through every element of the hash map and issues an error. The need to use a hash map is what made this validator's implementation a little more complex and came from the fact that because the activities are stored in a set for each phase, in order for the error marker to be placed in the correct activity within a set is necessary to have the index of the activity in the set.

```
1 @Check
```

```
def checkUnusedResource(Resource resource) {
  //List of used resources
  val EList<Resource> usedResourcesList = new BasicEList<Resource>
  val model = resource.eContainer as DesignFlowModel

  //Traverse activities
  for (activity : model.activities) {
    //Traverse resources
    for (currentResource : activity.resources) {
      usedResourcesList.add(currentResource)
    }
  }

  //Traverse used resources
  for (currentResource : model.resources) {
    if (!(usedResourcesList.contains(resource)) && (currentResource.
   name == resource.name)) {
      if (resource instanceof Tool) {
        error("Unused Tool '" + resource.name + "'", DFMLPackage.
   Literals.RESOURCE__NAME)
      }
      else {
        error("Unused File '" + resource.name + "'", DFMLPackage.
   Literals.RESOURCE__NAME)
      }
    }
  }
}
```

Listing A.3: checkUnusedResource validator.

In order to prevent unused resources from cluttering the design flow description the validator from Listing A.3 was implemented. First is created a list to store used resources and a variable to store the entire design flow model. The model is then traversed in order to collect every resource used by every activity of the design flow in the used resources list. Finally, every resource definition in the model is verified against the used resources list, if it is not in the list that means that it is not used by any activity and an error message is issued depending if the resource is either a file or a tool.

```
@Check
def checkMultipleOption(SetOption setOption) {
  val List<String> multipleOptionsList = newArrayList()
```

```
4
   (setOption.eContainer as Resource).options.forEach[SetOption so |
    multipleOptionsList.add(so.option.name)]
6
   if ((Collections.frequency(multipleOptionsList, setOption.option.
    name) > 1) && (!setOption.option.mul)) {
8    error("Option '" + setOption.option.name + "' is not allowed to
    have multiple occurrences", DFMLPackage.Literals.SET_OPTION__OPTION
    )
   }
10 }
```

Listing A.4: checkMultipleOption validator.

This validator checks if an option's multiplicity is incorrectly applied. First is created a list to store every option's name followed by the collection of all the options of a resource into this list. If an option's name appears with a frequency higher than one in the list and if that option's 'mul' attribute is false, that means the option is not allowed to have multiple values and a corresponding error is issued, as can be seen in Listing A.4.

A.3 GENERATORS

Listing A.5 shows the full implementation of the CMakeLists generator.

```
override doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
2  val model = resource.contents.get(0) as DesignFlowModel
   if (model !== null) {
4    val cmakeFiles = model.getFilesFromTemplate("CMakeLists")
     for (cmakefile : cmakeFiles) {
6      fsa.generateFile(cmakefile.filePath,
         '''
8        # DFML generated file

10       ≪val o1 = cmakefile.parseOption("cmake_minimum_required").
    iterator.next≫
         ≪o1.option.name≫(VERSION ≪(o1.valueType as StringValue).
    value≫)
12       ≪IF !cmakefile.parseOption("project").empty≫

14       ≪FOR o : cmakefile.parseOption("project")≫
         ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
```

```
16          ≪ENDFOR≫
            ≪ENDIF≫
18          ≪IF !cmakefile.parseOption("enable_language").empty≫

20          ≪FOR o : cmakefile.parseOption("enable_language")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
22          ≪ENDFOR≫
            ≪ENDIF≫
24          ≪IF !cmakefile.parseOption("set").empty≫

26          ≪FOR o : cmakefile.parseOption("set")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
28          ≪ENDFOR≫
            ≪ENDIF≫
30          ≪IF !cmakefile.parseOption("file").empty≫

32          ≪FOR o : cmakefile.parseOption("file")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
34          ≪ENDFOR≫
            ≪ENDIF≫
36          ≪IF !cmakefile.parseOption("list").empty≫

38          ≪FOR o : cmakefile.parseOption("list")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
40          ≪ENDFOR≫
            ≪ENDIF≫
42          ≪IF !cmakefile.parseOption("add_subdirectory").empty≫

44          ≪FOR o : cmakefile.parseOption("add_subdirectory")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
46          ≪ENDFOR≫
            ≪ENDIF≫
48          ≪IF !cmakefile.parseOption("aux_source_directory").empty≫

50          ≪FOR o : cmakefile.parseOption("aux_source_directory")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
52          ≪ENDFOR≫
            ≪ENDIF≫
54          ≪IF !cmakefile.parseOption("include_directories").empty≫

56          ≪FOR o : cmakefile.parseOption("include_directories")≫
            ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
58          ≪ENDFOR≫
```

```
        ≪ENDIF≫
60      ≪IF !cmakefile.parseOption("add_library").empty≫

62      ≪FOR o : cmakefile.parseOption("add_library")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
64      ≪ENDFOR≫
        ≪ENDIF≫
66      ≪IF !cmakefile.parseOption("link_directories").empty≫

68      ≪FOR o : cmakefile.parseOption("link_directories")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
70      ≪ENDFOR≫
        ≪ENDIF≫
72      ≪IF !cmakefile.parseOption("link_libraries").empty≫

74      ≪FOR o : cmakefile.parseOption("link_libraries")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
76      ≪ENDFOR≫
        ≪ENDIF≫
78      ≪IF !cmakefile.parseOption("add_executable").empty≫

80      ≪FOR o : cmakefile.parseOption("add_executable")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
82      ≪ENDFOR≫
        ≪ENDIF≫
84      ≪IF !cmakefile.parseOption("target_include_directories").
    empty≫

86      ≪FOR o : cmakefile.parseOption("target_include_directories")
    ≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
88      ≪ENDFOR≫
        ≪ENDIF≫
90      ≪IF !cmakefile.parseOption("target_link_libraries").empty≫

92      ≪FOR o : cmakefile.parseOption("target_link_libraries")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
94      ≪ENDFOR≫
        ≪ENDIF≫
96      ≪IF !cmakefile.parseOption("add_custom_target").empty≫

98      ≪FOR o : cmakefile.parseOption("add_custom_target")≫
        ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
```

```
100             ≪ENDFOR≫
                ≪ENDIF≫
102             ≪IF !cmakefile.parseOption("add_custom_command").empty≫

104             ≪FOR o : cmakefile.parseOption("add_custom_command")≫
                ≪o.option.name≫(≪(o.valueType as StringValue).value≫)
106             ≪ENDFOR≫
                ≪ENDIF≫
108             ''')
            )
110     }
    }
112 }
```

Listing A.5: CMakeLists generator.